UNIVERSIDADE FEDERAL DO RIO DE JANEIRO

ESCOLA POLITÉCNICA

DEPARTAMENTO DE ENGENHARIA ELETRÔNICA

E DE COMPUTAÇÃO

SISTEMA DE CONTROLE EM TEMPO REAL VIA BLUETOOTH

Autor: _____

Edson Hiroshi Watanabe

Orientador: _____

Fernando Cesar Lizarralde

Examinador: _____

José Ferreira de Rezende

Examinador: _____

Liu Hsu

DEL

Maio de 2005

AGRADECIMENTOS

Agradeço aos meus pais e a minha irmã pelo suporte que me deram em todos os momentos.

Agradeço a engenheira Viviane Medeiros, pela pessoa amiga que me ajudou nesses 5 anos de faculdade.

Agradeço ao Professor e Orientador Fernando Lizarralde, pela orientação e motivação ao longo do curso de graduação.

RESUMO

As tecnologias de comunicação sem fio trazem uma maior flexibilidade para a troca de dados entre dispositivos, mas também comprometem o determinismo de tempo para a entrega desses dados. Para investigar uma dessas tecnologias sem fio, a tecnologia ***Bluetooth***, e suas aplicações em sistemas de controle, foi criado um simulador em tempo real que permite a interação de um sistema de comunicação ***Bluetooth*** real com um sistema dinâmico que é virtual. O propósito desse simulador em tempo real é avaliar os efeitos do atraso de comunicação em sistemas de controle distribuído, ou avaliar sistemas de supervisão. São estudados dois sistemas de controle: um sistema de controle de um pêndulo rotacional invertido e um sistema de 2ª ordem. Os resultados das simulações em tempo real são comparados com uma reprodução dessas mesmas simulações "off-line", com métodos numéricos mais sofisticados, de forma a validar a estabilidade numérica do simulador e a sua correta execução no tempo.

PALAVRAS - CHAVE

- Simulação em tempo real
- Atraso de Comunicação
- Bluetooth
- Real Time Linux

ÍNDICE

CAPÍTULO 1

INTRODUÇÃO

Atualmente, muitas das tecnologias de comunicação sem fio tem sido impulsionadas pelos setores de telecomunicações e informática, o que contribui para: o surgimento de diversificados produtos, a redução de custos e o aprimoramento da tecnologia em si, que apresentam taxas de transmissão de dados cada vez mais elevadas.

Como exemplo podemos citar alguns padrões que especificam algumas dessas tecnologias: HomeRF (versões 1.0 e 2.0), IEEE802.11b/Wi/Fi, *Bluetooth* e IrDA (*Infrared Data Association*). Os 3 primeiros padrões utilizam a faixa de rádio freqüência, enquanto que o IrDA utiliza luz infravermelha. Nesse projeto vamos estudar a tecnologia sem fio *Bluetooth*. Para mais detalhes sobre essas tecnologias, ver a referência em [10].

Embora esses setores direcionem o desenvolvimento dessas tecnologias sem fio para um ambiente de escritório ou doméstico, já existem propostas em se aplicar estas mesmas tecnologias em ambiente industrial, como descrito em [1], [2] e [3].

A maior vantagem da comunicação sem fio é a flexibilidade e a facilidade com que se pode conectar com outro dispositivo e trocar dados. Em dispositivos na faixa de freqüência de rádio, com antenas omnidirecionais, basta que os dispositivos estejam dentro do alcance para que a comunicação possa ser estabelecida. Além disso, as ondas de rádio podem atravessar obstáculos como paredes. Tudo isso diminui a quantidade de planejamento necessário para a implementação do sistema de comunicação, permite a redução do cabeamento e uma maior mobilidade.

A desvantagem do uso de comunicação sem fio é a menor confiabilidade, dado que a probabilidade de erro na transmissão é algumas ordens de grandeza superior à probabilidade de erro nos sistemas cabeados. Em protocolos de comunicação que asseguram a integridade dos dados, isso significa que retransmissões freqüentes podem ocorrer, o que diminui o determinismo de tempo para a entrega dos dados de comunicação e traz restrições quanto à sua utilização em aplicações de tempo real.

A seguir vamos descrever os objetivos desse projeto.

1.1 – Objetivo

O objetivo desse projeto é estudar a tecnologia *Bluetooth* e algumas de suas possíveis aplicações em sistemas de controle. Para isso, serão criados um par de aplicações (cliente e servidor), que vão ser utilizadas para a troca de dados entre computadores, utilizando-se da tecnologia *Bluetooth*. Além disso, vamos simular em tempo real um sistema dinâmico, que representa um processo físico que estamos monitorando ou controlando. Uma simulação em tempo real é uma simulação mais realística, que nos permite avaliar o desempenho de um sistema de comunicação *Bluetooth*, que é real

e está interagindo com um sistema de controle de um processo físico, sendo este processo físico virtual.

Serão estudados os efeitos do atraso de comunicação em aplicações de tempo real e como compensar esses efeitos. Este projeto vai ser implementado em dois computadores pessoais utilizando os sistemas operacionais *Linux* e *Real Time Linux*.

1.2 – Motivação

As motivações para este trabalho podem ser divididas em 2 tópicos:

- utilização da tecnologia *Bluetooth* em sistemas de controle, para supervisão (monitoramento e ajuste de parâmetros), ou para a comunicação sem fio em sistemas de controle distribuído;
- desafio em se analisar um sistema de controle distribuído com atraso de transporte na malha de realimentação. Este atraso é resultado de: retransmissões de pacotes corrompidos; limitações de taxa de transmissão; utilização de sistema operacional não preemptivo[1] (Linux), que não garante um atraso máximo e é afetado pela carga do sistema;

preemptivo[1]: Embora não haja uma tradução correta para a palavra inglesa "preemptive", muitos livros técnicos de sistemas operacionais utilizam a palavra preemptivo para a tradução deste termo. Um exemplo é a referência [46]. Nesse contexto, um sistema operacional preemptivo indica que uma tarefa de maior prioridade sempre pode interromper uma de menor prioridade que esteja em execução, e começar a ser executada pelo sistema operacional.

A parte crítica deste sistema de comunicação, que é o rádio Bluetooth, o *hardware* controlador (que constituem um módulo *Bluetooth*) e o *firmware*, todos são adquiridos como produtos comerciais. Tudo isso aliado ao fato de que esta tecnologia pode ser integrada a computadores pessoais ou PDAs através da obtenção de *software* livre adequado, propicia a rápida criação e desenvolvimento de um sistema de teste para aplicações desta tecnologia.

A aplicação de tecnologias de Bluetooth em ambiente industrial já vem sendo investigada. As referências [1], [2] e [3] apresentam aplicações de dispositivos *Bluetooth* em ambiente industrial para substituir parte do cabeamento. Em [1], concluiu-se que pode haver benefícios na sua utilização em sensores localizados em partes móveis ou rotativas de máquinas e para a comunicação de veículos guiados autonomamente. Em [2] e [3], a combinação de tecnologias de comunicação sem fio e de *Internet* (como por exemplo a *World Wide Web* – WWW e *Wireless Application Protocol* - WAP), pode resultar em interfaces HMI (*Human Machine Interfaces*) portáveis, de forma que um equipamento a ser supervisionado pode ser acessado pelo operador com um dispositivo portátil, como um aparelho de telefone ou um PDA (*Personal Digital Assistant*). Por exemplo, com isso seria possível:

- apresentar, em tempo real, dados recebidos de um nó sensor em um gráfico;
- carregar um arquivo de dados, com o registro de eventos ocorridos ao longo de um período de tempo;

- receber um sinal de alarme. Ex. um alarme é enviado direto a seu aparelho de telefone, para alertar sobre algum evento;
- ajustar ou verificar parâmetros de controle;

Em uma outra aplicação da tecnologia *Bluetooth*, temos um sistema de controle onde os sensores, o controlador e o atuador estão em locais físicos diferentes, conectados por uma rede de comunicação. Nesse caso, temos um sistema de controle distribuído, ver referências [4] e [5]. A análise desse sistema pode ser reduzida ao problema de se analisar um sistema de controle, com atraso de transporte variante no tempo entre um nó sensor e um nó controlador e de um nó controlador para um nó atuador.

A seguir estudaremos os conceitos de Laboratórios Virtuais e Laboratórios Remotos, que representam um ponto de partida para o estudo realizado nesse trabalho.

1.3 – Laboratórios Virtuais e Laboratórios Remotos

Muitas das idéias utilizadas para a implementação deste trabalho são semelhantes a utilizadas na implementação de um Laboratório Virtual ou um Laboratório Remoto. Esses tipos de laboratórios são utilizados para a realização de experimentos (reais ou virtuais) à distância, o que pode beneficiar o aprendizado, constituindo-se uma forma de se auxiliar no ensino. Alguns exemplos podem ser vistos em [12] e [13]. Vamos então analisar alguns aspectos de implementação de um laboratório virtual/remoto e relacioná-los ao trabalho desenvolvido.

Laboratórios Virtuais são descritos em [12] como uma simulação em software de sistemas físicos, e são disponibilizados para serem acessados via uma rede local ou pela *Internet*. O usuário deve poder interagir com o sistema simulado, alterando parâmetros e verificando os resultados. Esses sistemas são utilizados por Instituições de Ensino para complementar atividades de laboratório, permitindo aos alunos acesso a qualquer instante para verificar algum experimento (que será resolvido numericamente).

Os laboratórios virtuais funcionam de acordo com um modelo de cliente/servidor, onde um usuário (cliente) tenta acesso a uma máquina que hospeda um servidor (por exemplo um servidor WWW) e se for bem sucedido deverá poder utilizar uma interface gráfica para alterar parâmetros, iniciar uma simulação, parar a execução, gerar gráficos para visualizar os resultados, entre outros comandos.

Já o conceito de laboratório remoto difere do conceito de laboratório virtual. No laboratório remoto, os usuários também necessitam acessar uma máquina que roda um aplicativo servidor, via rede local ou Internet, mas o experimento é real. O servidor deve estar conectado a um *Hardware* especializado em aquisição de dados, com conversores A/D (analógico/digital) e D/A (digital/analógico). Esse aparato é necessário para a coleta de dados provenientes de sensores e para o envio de sinais de controle para uma planta. Como exemplo, temos o trabalho desenvolvido em [13], em que um usuário pode acessar via rede local ou Internet, um servidor que permitirá que se alterem

parâmetros de um experimento de controle de velocidade de um motor de corrente direta, tais como: o período de amostragem, a lei de controle utilizada pelo servidor e os sinais de referência para controlar a velocidade. Outra possibilidade interessante é que o cálculo do sinal de controle pode ser feito no lado do cliente. Nesse caso, o servidor envia os sinais amostrados para o cliente e espera receber do cliente os sinais de controle discreto. A figura 1.1 ilustra o sistema implementado em [13], para o caso em que o controle é calculado no lado do cliente.

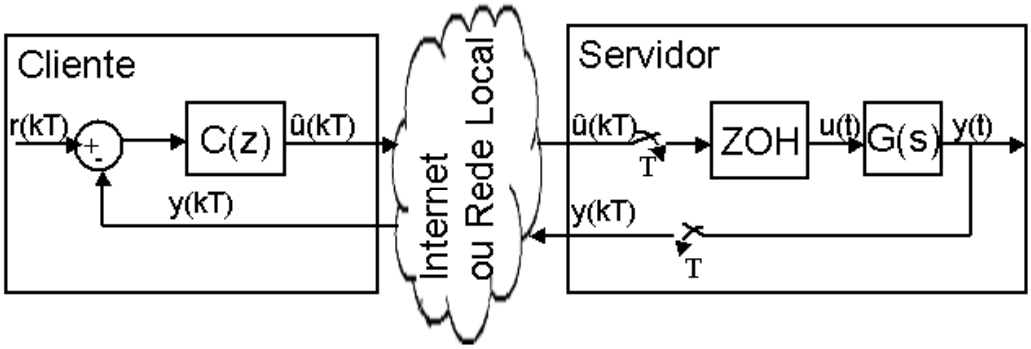


Figura 1.1 – Laboratório Remoto apresentado em [13], com o cálculo do controle calculado no lado do cliente.

Na ilustração da figura 1.2, se por hipótese o atraso da rede fosse estático e conhecido, poderíamos selecionar entre as opções a) e b) da figura 1.2, caso o atraso de ida e volta de um pacote de dados (ou o *Round Trip Time* - RTT) não exceder um limiar máximo, que faz parte da especificação do sistema de controle. Nesse caso, a figura a) apresenta uma configuração de controle distribuído, com o controlador no lado do cliente e em b) temos uma configuração em que o cliente apenas monitora os dados gerados pelo experimento. Note que RTT sempre denota o atraso de ida e volta de um pacote de dados, e no caso da figura 1.2b) é este atraso que foi medido.

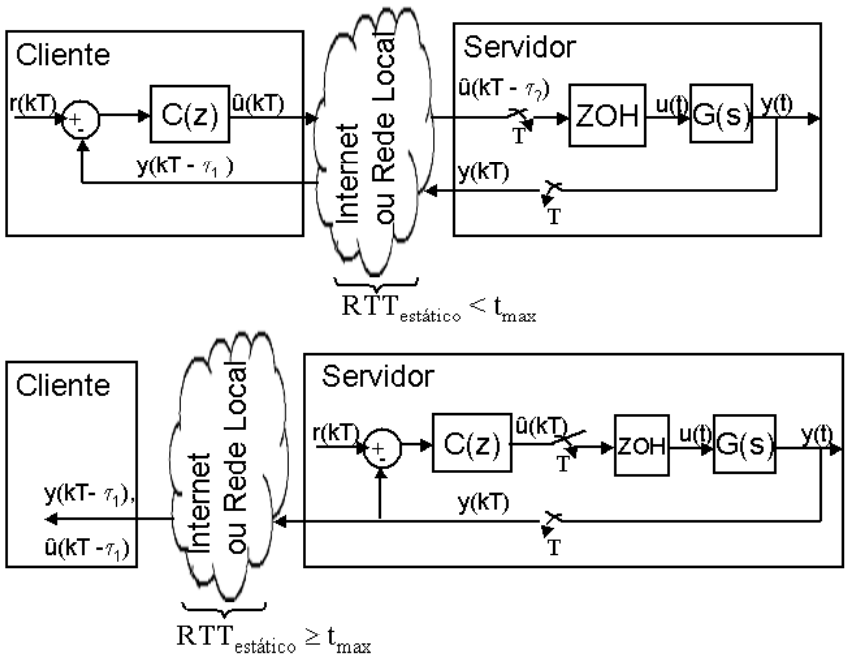


Figura 1.2 – Configurações do sistema Laboratório Remoto/Virtual,: a)controle distribuído; b) monitoramento.

As hipóteses utilizadas nos sistemas da figura 1.2 não são razoáveis, pois o atraso RTT não é estático. No entanto, esse exemplo da figura 1.2 serve para ressaltar que se um sistema de comunicação fornecer serviços de transporte de dados para um sistema de controle, este sistema de comunicação deve ser avaliado em relação ao atraso no transporte desses dados de controle.

A seguir será apresentado o conceito de sistemas de controle distribuído.

1.4 – Sistemas de Controle Distribuído

No item anterior, apresentamos um exemplo em que a malha de controle é fechada através de uma rede de comunicação. A este tipo de sistema, onde o controlador, sensores e o atuador não estão centralizados num mesmo dispositivo e a comunicação é feita através de uma rede, chamamos a isto de sistema de controle distribuído em tempo real. Estudaremos apenas os casos em que os sinais que trafegam pela rede de comunicação são sinais digitais (e não sinais analógicos).

Esses sistemas de controle distribuído começaram a ser utilizados durante a década de 70, sendo a indústria automobilística a principal área de aplicação desse tipo de abordagem de controle. As principais redes de comunicação são os de barramento de campo, como o FIP (*Factory Instrumentation Protocol)* e o Profibus (*Process Fieldbus*), e barramentos automobilísticos como o CAN (Controller Area Network).

Uma breve introdução a esse sistema de controle e as redes de comunicação pode ser encontrada em [15], entretanto são considerados apenas casos em que os dispositivos estão ou conectados diretamente ou compartilhando o mesmo meio com outros dispositivos, não sendo, portanto, uma análise que abranja o caso descrito em [13], onde a Internet é utilizada

Considerando apenas que os dispositivos estão ou diretamente conectados ou compartilham de um mesmo meio físico com outros dispositivos, temos que um dos principais problemas encontrados nesse tipo de sistema é o atraso de comunicação introduzido por:

- limitações de taxa de transmissão;
- controle de acesso ao meio não garante um atraso máximo para a transmissão de dados (MAC – Medium Access Control);
- perda de pacotes e retransmissões comprometem a estimativa de pior caso;

A limitação na taxa de transmissão introduz retardo, pois é necessário mais tempo para a transmissão de um pacote de dados.

Em relação ao controle de acesso ao meio, é desejável que todo dispositivo que necessite transmitir dados possa acessar ao meio (compartilhado com outros dispositivos) em um tempo limitado e conhecido, e que transmissões prioritárias ganhem acesso antes das transmissões de menor prioridade. Nesses casos, pode-se apenas garantir que o atraso para acessar ao meio é limitado, mas o atraso introduzido é aleatório.

Outro fator que introduz atrasos aleatórios é a perda de pacotes de dados. Em redes cabeadas, a taxa de erros é considerada pequena o suficiente para não ser considerada um problema restritivo,

mas em redes sem fio a taxa de erros é algumas ordens de grandeza superior que nas redes cabeadas, e é considerado um problema para poder prever o pior caso de atraso de comunicação.

No caso de utilizar a Internet ou a Rede Local, como descrito em [13], outros problemas ocorrem, como:

- tráfego concorrente e congestionamento, o que leva a uma espera em fila nos roteadores e comutadores
- perda de pacotes nos roteadores devido a *buffer overflow*, no qual pacotes são descartados por não encontrar espaço na fila de espera do roteador.

De maneira geral, a figura 1.3 ilustra o que a rede de comunicação deve representar para um sistema de controle distribuído. T_{SC} é o atraso de comunicação do sensor para o controlador e T_{CA} é o atraso de comunicação do controlador para o atuador.

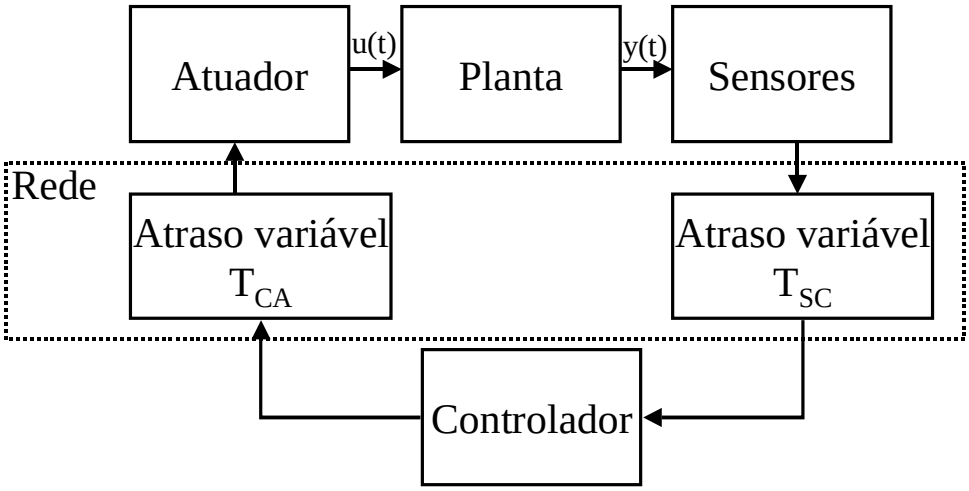


Figura 1.3 – Sistema de Controle Distribuído em tempo real.

A partir da figura 1.3, vamos voltar ao artigo [13] e utilizar o modelo da planta que se deseja controlar. Assumindo que $T_{SC} = 0$ e $RTT = T_{CA}$ = constante, podemos analisar o sistema apresentado na figura 1.5, para RTT = 0 ms, 200 ms, 400 ms e 600 ms. O período de amostragem é T = 200 ms. Novamente essas condições ($T_{SC} = 0$ e RTT constante) não são razoáveis para o problema em questão, mas são utilizadas aqui para ilustrar o efeito de atraso na malha de realimentação.

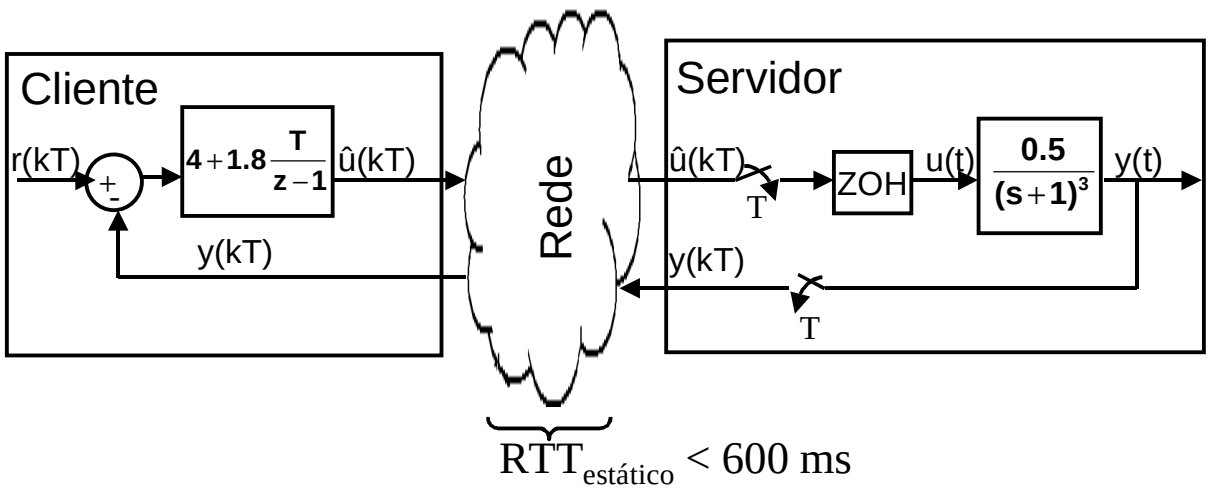


Figura 1.4 – Sistema de controle distribuído, com controlador PI (Proporcional Integral).

Na figura 1.4, a planta que se deseja controlar tem função de transferência:

$$G(s) = \frac{0.5}{(s+1)^3}$$

e o controlador utilizado nesse exemplo é um PI (Proporcional Integral) $C(s) = 4 + \frac{1.8}{s}$.

Para exemplificar os efeitos de um atraso de comunicação estático nesse sistema de controle, vamos apresentar na figura 1.5 a resposta ao degrau do sistema da figura 1.4, para diferentes atrasos estáticos.

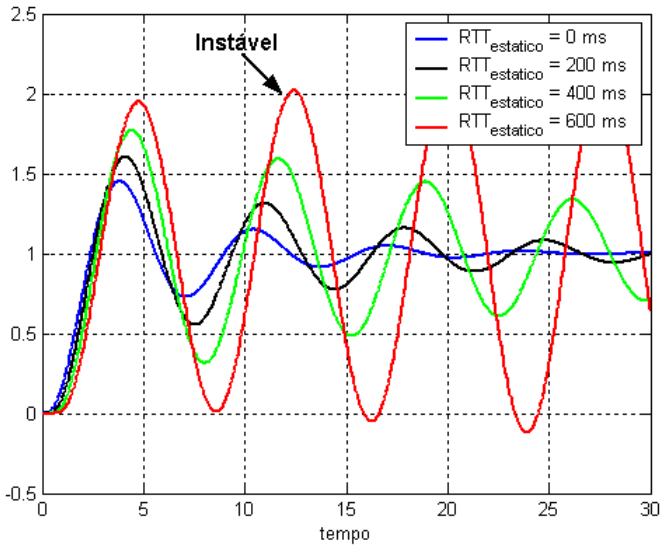


Figura 1.5 – Curva de y(t), para o sistema da figura 1.4, quando RTT = 0 ms, 200 ms, 400 ms e 600 ms. Para RTT = 600 ms, o sistema é instável.

Por fim, este item mostrou e ilustrou um dos principais problemas encontrados em sistemas de controle distribuído. No próximo item será apresentado o sistema que utiliza o *Bluetooth* e no qual uma das aplicações é a sua utilização em controle distribuído.

1.5 – Estudo de caso

Um sistema de comunicação *Bluetooth* será implementado utilizando-se dois computadores pessoais, e dois dispositivos *Bluetooth* comercialmente vendidos pela 3Com. A figura 1.6 demonstra o cenário. Chamaremos um PC de “PC Cliente” e o outro de “PC Servidor”.

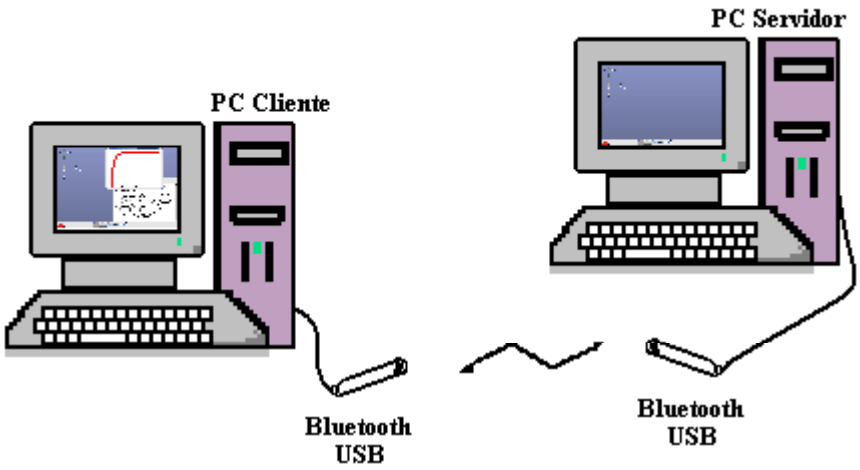


Figura 1.6 – Cenário do estudo de caso.

O PC Cliente utilizará o sistema operacional Linux, onde uma aplicação foi desenvolvida utilizando programação com interfaces de socket, em linguagem C, que possibilita acessar aos protocolos de camada superior do *Bluetooth* para a comunicação com dispositivos próximos.

O PC Servidor utilizará o sistema operacional Real Time Linux e o sistema operacional Linux. Assim, no PC Servidor estarão sendo executadas duas tarefas: a primeira é um aplicativo servidor, que espera por conexões *Bluetooth* provenientes de outros computadores, e a segunda tarefa é uma rotina que simula em tempo real uma planta (por exemplo um sistema de um pêndulo rotacional invertido) e transmite os dados para o aplicativo servidor como se fossem dados adquiridos de sensores.

Duas aplicações serão desenvolvidas. A primeira aplicação visa a utilização do *Bluetooth* apenas para a supervisão de sistemas, onde os dados gerados pelos sensores de uma planta poderiam ser monitorados por um ou vários clientes que receberão os dados do PC Servidor. A outra aplicação visa ao estudo de sistemas de controle distribuído em tempo real, onde se considera que a comunicação *Bluetooth* é rápida o suficiente para suportar aplicações de controle com restrições de tempo (*Hard Real Time*), e que o controlador poder ser projetado especialmente para compensar um atraso estático e conhecido.

Essa abordagem utiliza os conceitos de Laboratórios Virtuais e Laboratório Remoto, no que concerne a simulação de uma planta para a geração de dados e na supervisão e monitoramento desses dados, como descrito em [12] e [13], e de sistemas de controle distribuído em tempo real, como descrito em [15], onde os sensores, atuadores e o controlador estão em nós diferentes de uma rede.
Nesse estudo de caso, o enfoque está centrado em responder a 2 questões:

- Quais fatores podem contribuir para a variação do atraso de transporte das informações do sistema de controle;
- Como abordar o problema de atraso na malha de realimentação;

A seguir é apresentada a organização do conteúdo.

1.6 – Organização do Conteúdo

A organização do conteúdo é apresentada a seguir.

No capítulo 2, serão apresentados detalhes técnicos da tecnologia *Bluetooth.* O capítulo começa contando um pouco da história, de como a tecnologia foi concebida pela Ericsson Mobile, e em seguida apresenta os conceitos de pilhas de protocolo e perfis *Bluetooth*. Os protocolos *Baseband, Link Manager e* L2CAP e a interface HCI são detalhados. Por fim são descritos os módulos *Bluetooth* da 3Com adquiridos para este projeto e também é feita a descrição do Bluez, um pacote de softwares livres necessário para utilizar o Bluetooth no sistema operacional Linux. No apêndice B será descrito como adquirir e utilizar o Bluez no sistema operacional Linux, e como criar programas em linguagem C que utilizem a comunicação via *Bluetooth.*

No capítulo 3, é apresentado o sistema operacional Real Time Linux/Free (RTLinux), que constitui um sistema operacional em tempo real. Será definido o conceito de aplicações em tempo real e como o sistema operacional RTLinux se difere do sistema operacional Linux. Também são apresentados os histogramas que ilustram a precisão da execução das tarefas em tempo real no Linux e no RTLinux. No apêndice C será apresentado como adquirir e instalar o sistema operacional RTLinux/Free e como criar tarefas para serem executadas em tempo real.

No capítulo 4, vamos apresentar o conceito de simulação em tempo real. Como estamos interessados em simular sistemas dinâmicos e estes são descritos por modelos matemáticos, vamos apresentar métodos numéricos para a resolução de suas equações diferenciais ordinárias, e indicar qual são os métodos mais apropriados para uma simulação em tempo real.

No capítulo 5, vamos apresentar 2 estudos de casos, que servirão para validar a implementação do simulador em tempo real, e abordagens de controle linear que compensem ou tornem o sistema robusto a um atraso estático.

No capítulo 6, vamos apresentar uma visão geral da implementação do sistema, incluindo o Servidor e Cliente *Bluetooth*, a tarefa em tempo real que simula um sistema dinâmico, e os programas auxiliares que registram os dados gerados pela simulação.

No capítulo 7, temos a apresentação dos resultados de simulação em tempo real, histogramas do atraso de comunicação, e outras medidas obtidas através da simulação em tempo real.

No capítulo 8 é apresentada a conclusão e trabalhos futuros.

CAPÍTULO 2

Bluetooth

2.1 – Um pouco de História

Nas décadas de 1960s e 1970s, os sistemas de telecomunicações pertenciam a grandes e poucas organizações do setor público e privado, e os computadores (de grande porte) eram bens de custo elevado e de difícil operação, o que restringia o seu uso para as grandes corporações, universidades e agências de governo que se dispunham a adquiri-los e a treinar pessoal para operá-los.

Nas duas décadas passadas, a se especificar as décadas de 1980 e 1990, houve uma mudança dramática na forma em que as tecnologias de telecomunicações e de sistemas de informação se distribuíram entre a sociedade. Na década de 1980, a popularização dos computadores pessoais (PC – *Personal Computer*) de relativo menor custo, não somente permitiram que as pessoas tivessem acesso à geração e manipulação de conteúdo digital, como também distribuíram o conhecimento específico para operar os mesmos. Nessa mesma década começam os primeiros movimentos dos governos para a desregulamentação e para liberalizar o setor de telecomunicações, promovendo competição, inovação de serviços e produtos a menores custos. Nos anos 90, a popularização da Internet, o crescimento exponencial do mercado de telefonia celular, entre outros acontecimentos, marcam uma nova etapa na distribuição das tecnologias de informação e telecomunicação.

Nesse contexto dos anos 90, a tecnologia *Bluetooth* vai ser originada a partir das empresas líderes no setor de computação e telecomunicações. Em 1994, pesquisadores da Ericsson Mobile Communications buscavam uma maneira de conectar dispositivos como PDAs a rede de telefonia celular através de aparelhos celulares, sem a utilização de cabos seriais. Para realizar a comunicação sem fio, foi projetado um *link* de comunicação via rádio de curta distância que conectasse o PDA ao aparelho celular, cujos principais requisitos eram consumo reduzido de energia e baixo custo. Prevendo a potencialidade comercial dessa tecnologia, criada para conectar dispositivos eletrônicos, a Ericsson torna a tecnologia aberta e reúne mais 4 grandes empresas: a IBM Corporation, a Intel Corporation, Nokia Corporation e a Toshiba Corporation. Essas 5 grandes empresas criam o *Bluetooth Special Interest Group* em 1998. O intuito de se abrir a tecnologia é torná-la largamente aceita e utilizada pelas empresas, para torná-la um padrão *de facto* global. Com isso é inibido o surgimento de padrões concorrentes e garante-se o mercado para um produto amplamente suportado (pelo menos essa era a idéia). Em 1999 é lançado a versão 1.0 da especificação *Bluetooth* (ver [11] para obter a especificação) e mais 4 grandes empresas se juntam ao SIG: a 3Com Corporation, Lucent Technologies, Microsoft Corporation e a Motorola. No final do ano 2000 os primeiros produtos *Bluetooth* são lançados. Para outros detalhes, ver referências [10] e [18].

A especificação *Bluetooth* é continuamente revisada e atualizada. Em 2001 foi lançada a versão 1.1 da especificação, que é a versão estudada neste trabalho. Em 2003 é lançada a versão 1.2 e em 2004 a versão 2.0. Para obter as especificações, acesse a página da Internet na referência [11].

2.2 – A tecnologia *Bluetooth* (versão 1.1)

Nessa seção abordaremos alguns aspectos relevantes, para este trabalho, da especificação *Bluetooth* versão1.1, descrita em [9], o qual poderá ser obtida em [11].

Para se entender o *Bluetooth*, uma abordagem é começar a explicar os conceitos de protocolo, pilha de protocolo e perfis *Bluetooth*. Em seguida as funções das principais camadas de protocolo *Bluetooth* serão explicadas.

2.2.1 – Pilhas de Protocolo e Perfis

Um protocolo pode ser definido como um conjunto específico de mensagens que são trocadas e de ações tomadas em função de uma mensagem recebida ou a falta dela.

Uma pilha de protocolos (tradução do termo em inglês "*protocol stack*") se refere a um conjunto de protocolos que trabalham em conjunto. Por exemplo, as sete camadas do modelo OSI (*Open System Interconnection*) podem ser vistas como uma pilha de protocolo, para detalhes ver [20]. Esse termo também é um jargão utilizado por programadores para fazer referência a uma implementação em software que processa e gerência os protocolos. Por todo o texto daqui em diante, o termo "pilha de protocolo" se refere a protocolos que trabalham em conjunto em um sistema e não a uma implementação em software.

A especificação do *Bluetooth* descreve um conjunto com vários protocolos, mas para uma dada aplicação, um projetista vai utilizar apenas um subconjunto deste. Significa que nem todos os protocolos especificados são necessariamente utilizados, apenas dão maior flexibilidade e habilitam um maior número de serviços disponíveis. A figura 2.1 apresenta os protocolos que pertencem a tecnologia *Bluetooth*.

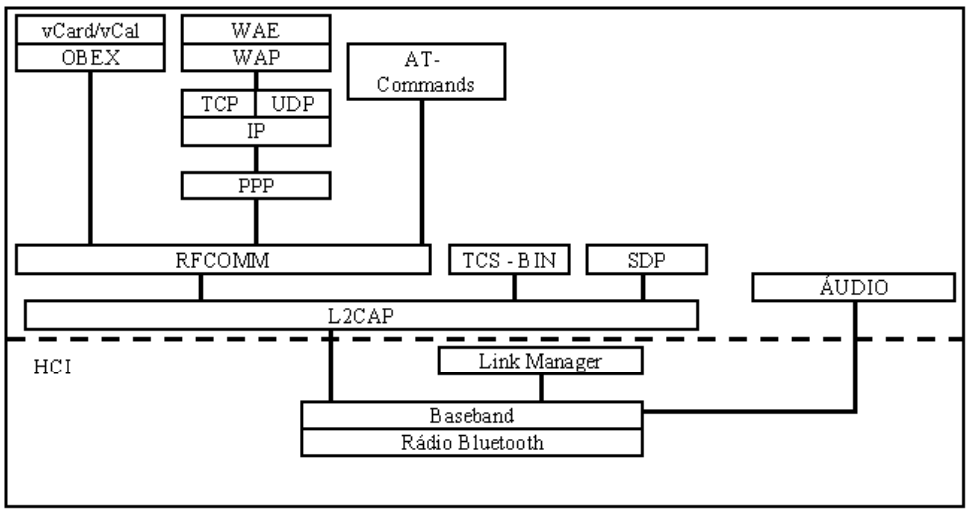


Figura 2.1 –Protocolos da tecnologia *Bluetooth*.

Para facilitar o desenvolvimento de produtos, os perfis *Bluetooth* nada mais são do que uma descrição técnica para implementar uma aplicação. Essa descrição técnica apresenta uma pilha de protocolo pré-definida e procedimentos necessários para que uma aplicação funcione.

Por exemplo, em um projeto envolvendo a utilização da tecnologia *Bluetooth* entre um aparelho celular e um *headset* (fone de ouvido), o projetista deverá verificar o perfil *Headset* (ver [9]) e considerar o conjunto de protocolos que deverão ser utilizados e como os parâmetros devem ser ajustados. Na figura 2.2 as pilhas de protocolos apresentadas são a maneira especificada em [9] para implementar uma aplicação que comunique o aparelho celular a um *headset* através do Bluetooth. As funcionalidades dos protocolos Bluetooth (indicado pela chave vermelha) serão descritas no item 2.2.2.

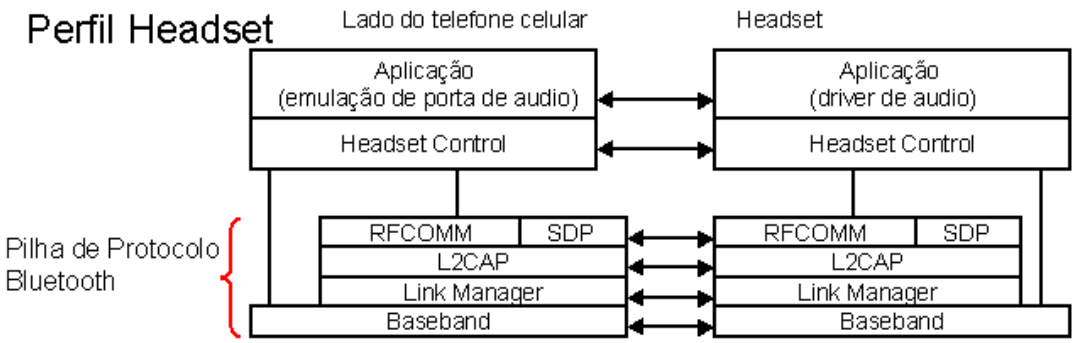


Figura 2.2 – O Perfil *Headset* especifica qual é a pilha de protocolo *Bluetooth* utilizada em cada dispositivo eletrônico (no caso o telefone celular e o *headset*);

O mesmo procedimento pode ser utilizado para o *Bluetooth* conectar um *pda* a um ponto de acesso (o ponto de acesso pode ser um PC com *Bluetooth*, conectado a uma rede local), onde o perfil de LAN (Local *Area Network*) é o perfil a ser consultado em [9]. A figura 2.3 ilustra a pilha correspondente ao lado do pda (não foi representado o PC). Para ver um trabalho que aborda o uso de pdas e pontos de acesso, veja [21].

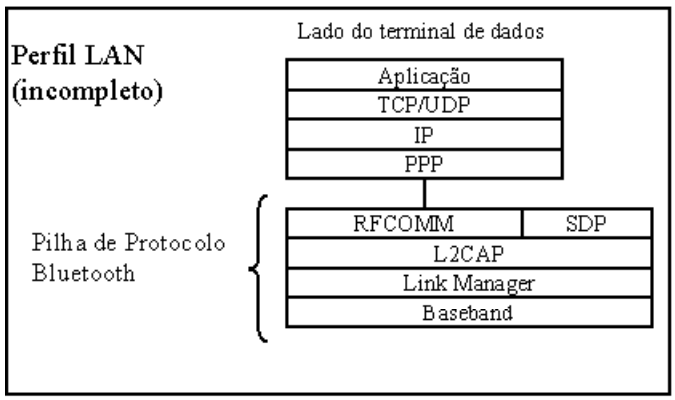


Figura 2.3 – O Perfil LAN, descreve qual a pilha de protocolos a ser utilizada no lado do *pda*.

Os perfis são criados a partir de grupos de trabalho (tradução do termo em inglês "*Working Groups*"), formados por representantes de empresas do *Bluetooth* SIG. Infelizmente até Janeiro de 2005, um grupo de estudo do Bluetooth SIG de nome *Industrial Automation*, que estuda a aplicações

do *Bluetooth* em automação, ainda não havia progredido do estado de *Study Group* para *Working Group*, e portanto não tem permissão para poder definir perfis.

2.2.2 – Protocolo Principais *Bluetooth* (*Core Protocols*)

Os protocolos Bluetooth podem ser divididos em 4 grupos:

- *Core Protocols*
- *Adopted Protocol*
- *Cable Replacement Protocol*
- *Telephony Control Protocols*

O grupo *Core Protocols* corresponde ao grupo de protocolos obrigatoriamente implementados nos dispositivo *Bluetooth*. Protocolos do *Adopted Protocols* são protocolos já existentes que foram adotados pelo Bluetooth, como por exemplo o *Point-to-Point Protocol* (PPP), o TCP (*Transmission Control Protocol*) e o IP (*Internet Protocol*). O *Cable Replacement Protocol* possui apenas um único protocolo chamado RFCOMM, que serve para emular comunicações seriais entre dispositivos. O *Telephony Control Protocols* é utilizado para permitir que o *Bluetooth* funcione como um telefone ou modem. A figura 2.4 ilustra esses grupos, que também estão representados na figura 2.1 e não devem ser confundidos com o conceito de pilha de protocolo.

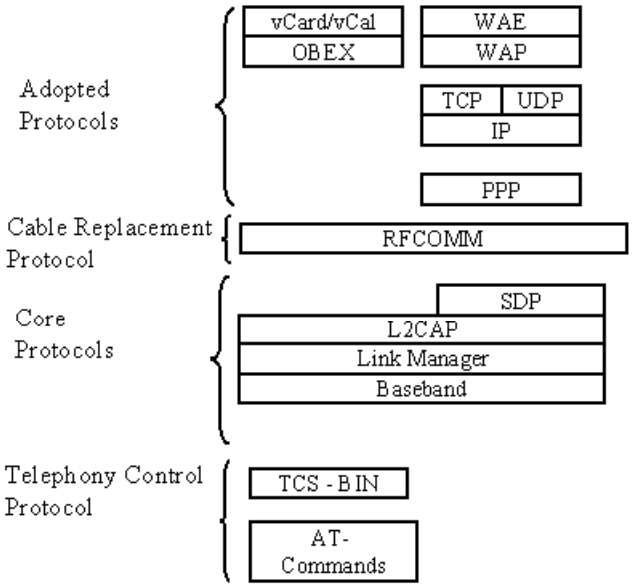


Figura 2.4 – Os quatro grupos de protocolos do *Bluetooth*

Neste trabalho, vamos utilizar apenas os protocolos pertencentes ao *Core Protocols* (a exceção é o protocolo *Service Discovery Protocol* que não será utilizado) e que estão listados a seguir:

- *Logical Link Control and Adoption (L2CAP)*
- *Link Manager Protocol (LMP)*
- *Baseband*

A seguir, cada um desses protocolos será descrito, no entanto devemos introduzir o *Bluetooth Radio* e o *Host Controller Interface (HCI).* Em um sistema *Bluetooth*, uma parte do sistema é implementada no *Bluetooth Host* (ex. PC ou PDA) e a outra parte é implementada no módulo *Bluetooth* (que contém apenas o rádio e *Hardware* controlador), também chamado de *Bluetooth Controller*. As camadas superiores, como o L2CAP e o PPP, são implementadas no Host. As camadas inferiores, como a *Baseband* e o LMP (Link *Manager Protocol*) são implementadas no módulo Bluetooth. Para a comunicação entre o módulo e o *Host*, é criada uma interface chamada *Host Controller Interface*, que permite ao Host acessar ao módulo Bluetooth, abstraindo-se de que tipo de barramento físico se está utilizando, como por exemplo o USB (*Universal Serial Bus*) ou PC-Card. Uma visão geral da arquitetura de um módulo *Bluetooth* pode ser vista na figura 2.5.

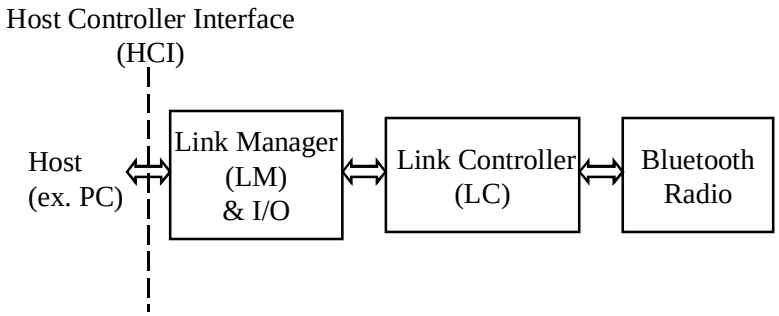


Figura 2.5 – Diagrama de blocos de um módulo *Bluetooth*

Os blocos apresentados na figura 2.5 do módulo *Bluetooth* são:

- *Bluetooth Radio*
- *Link Controller*
- *Link Manager*

A especificação do *Bluetooth Radio* é descrita na seção 2.2.3. O *Link Controller* é responsável pela execução do protocolo *Baseband*, que será descrito na seção 2.2.4. *O Link Manager* tem como função a configuração e controle de um enlace de comunicação. O *Link Manager (LM)* utiliza o *Link Manager Protocol* (LMP) para se comunicar com outros LM remotos, o que será descrito na seção 2.2.5. O *Host Controller Interface* será apresentado na seção 2.2.6, assim como uma breve descrição do USB (*Universal Serial Bus*) por ser o tipo de barramento físico utilizado neste trabalho. O *Host* nesse trabalho é um computador pessoal , como os descritos na seção 1.2.4. No *Host* apenas o protocolo L2CAP será utilizado, o que será descrito na seção 2.2.7. Para este trabalho, entender como o protocolo L2CAP funciona e como interage com as camadas inferiores é de maior importância, pois todas as medidas de interesse serão feitas na camada imediatamente acima deste protocolo (na camada de aplicação). A figura 2.6 indica qual seção corresponde a cada um dos itens que irão ser explicados.

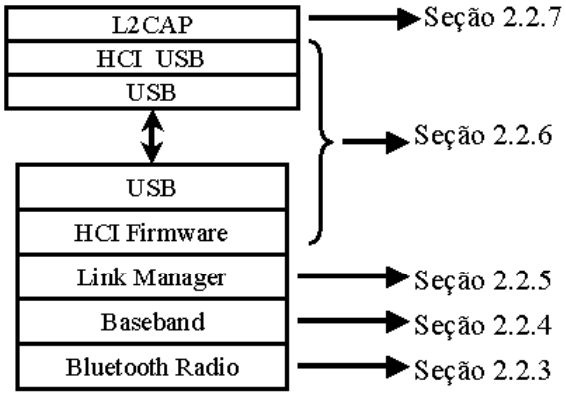


Figura 2.6 – Organização do conteúdo das próximas seções.

É importante avisar que nenhum aspecto relacionado a segurança na comunicação de dispositivos *Bluetooth* será analisada, como por exemplo a criptografia e administração de chaves nos protocolos *Baseband* e *Link Manager*. Quaisquer opções de segurança estarão desativas nesse projeto.

2.2.3 – A especificação do *Bluetooth Radio*

Todo módulo *Bluetooth* deve possuir um transceptor, ou seja, um transmissor e um receptor que são multiplexados no tempo para transmitir ou receber dados em um sistema de comunicação *half duplex*. A especificação *Bluetooth Radio* descreve os requerimentos para que um transceptor seja compatível com outros rádios *Bluetooth* e define a qualidade desejada do sistema, mas não define como uma implementação deve ser feita. A especificação do rádio se divide em três partes :

- a descrição da faixa de freqüência utilizada;
- a descrição dos requerimentos do transmissor;
- a descrição dos requerimentos do receptor;

Um transceptor *Bluetooth* deve operar na banda de freqüência de 2.4 GHz ISM (Industrial *Scientific Medicine*). A banda ISM é livre de licenciamento e equipamentos projetados para trabalhar nessa faixa de freqüência podem gerar e utilizar localmente energia de radiofreqüência, sem a necessidade de pagar por isto. Infelizmente as faixas ISM atribuídas na freqüência de 2.4 GHz podem variar de país para país, o que pode implicar na incompatibilidade de transceptores *Bluetooth* projetados para países diferentes, como por exemplo a França e os Estados Unidos. A seguir a tabela 2-1 apresenta algumas faixas ISM 2.4 GHz que o *Bluetooth* utiliza, dependendo da regulamentação de cada país. Essa tabela corresponde ao ano de 2001, quando foi lançada a versão 1.1 da especificação *Bluetoooth*.

Tabela 2-1 : Faixas ISM 2.4 GHz utilizadas pelo *Bluetooth* (versão 1.1) de alguns países

País	Faixa ISM 2.4 GHz
Brasil	2400 – 2483.5 MHz
Estados Unidos	2400 – 2483.5 MHz
França	2446.5 – 2483.5 MHz
Japão	2471 – 2497 MHz

No Brasil, a Anatel (Agência Nacional de Telecomunicações) atribui várias faixas às aplicações industriais, científicas e médicas, e algumas dessas faixas ISM são de: 902 – 928 MHz, 2400 – 2500 MHz e 5725 – 5875 MHz. No entanto estas aplicações não podem interferir em sistemas que operem fora da faixa ISM e também não se pode solicitar proteção a Anatel contra interferência de outros equipamentos que operem na mesma faixa ISM. O transceptor *Bluetooth* é projetado para operar na faixa de 2400 – 2483.5 MHz, utilizando a técnica de espalhamento espectral para aumentar a robustez do sistema em relação a interferência de outros dispositivos.

A técnica de espalhamento espectral consiste na utilização de uma largura de banda muito maior do que a necessária para transmitir um sinal que contém a informação. No caso do sistema *Bluetooth* a taxa nominal máxima é de 1 Mbps e um único canal de 1 MHz poderia ser utilizado nesse sistema. No entanto, com a técnica de espalhamento espectral, são utilizados 79 canais de 1 MHz, ou seja, há um total de 79 MHz disponíveis para a transmissão de dados. Para evitar interferência com as bandas adjacentes da ISM devido à emissões não desejadas, são utilizadas bandas de guarda de 2 MHz no início da banda ISM e de 2.480 – 2.483.5 MHz. A figura 2.7 ilustra o que foi descrito. Tudo isso corresponde ao que é feito na maioria dos países, mas em alguns países o número de canais utilizados é menor que 79, como por exemplo na França, onde o número de canais utilizados é de 23 (de 1 MHz) para se adequar à regulamentação do país (nas versões acima da 1.1 da especificação Bluetooth, os canais de 23 não são mais usados). Transceptores *Bluetooth* que implementam a banda completa (79 canais de 1 MHz) não interoperam com transceptores com banda reduzida (< 79 canais de 1 MHz).

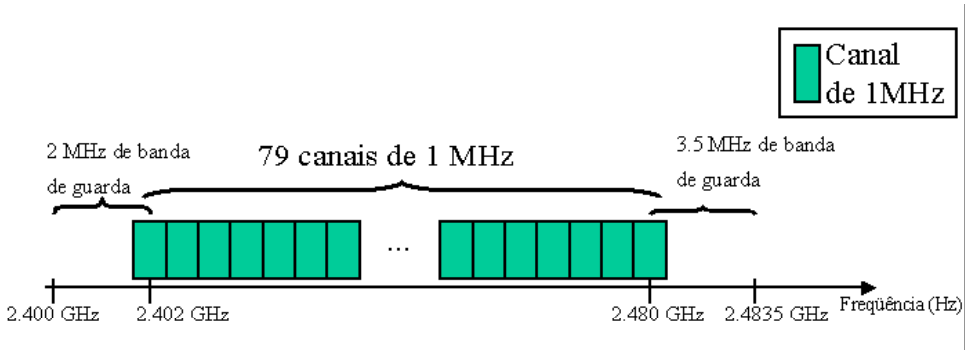


Figura 2.7 – Utilização da faixa ISM pelo *Bluetooth*, 79 canais de 1 MHz para dados e bandas de guarda

No entanto os 79 canais de 1 MHz não são utilizados para a transmissão ao mesmo tempo, e sim seqüencialmente. Os canais de 1 MHz são multiplexados no tempo, através da técnica de salto em freqüência (*frequency hopping*), no qual transmissões são feitas em canais de freqüência diferentes.

Cada transmissão pode ocupar 1, 3 ou 5 slots de tempo, sendo a duração de um slot de 625 μs. O esboço da figura 2.8 exemplifica o espalhamento espectral com saltos em freqüência (em inglês *Frequency Hopping Spread Spectrum* – FHSS).

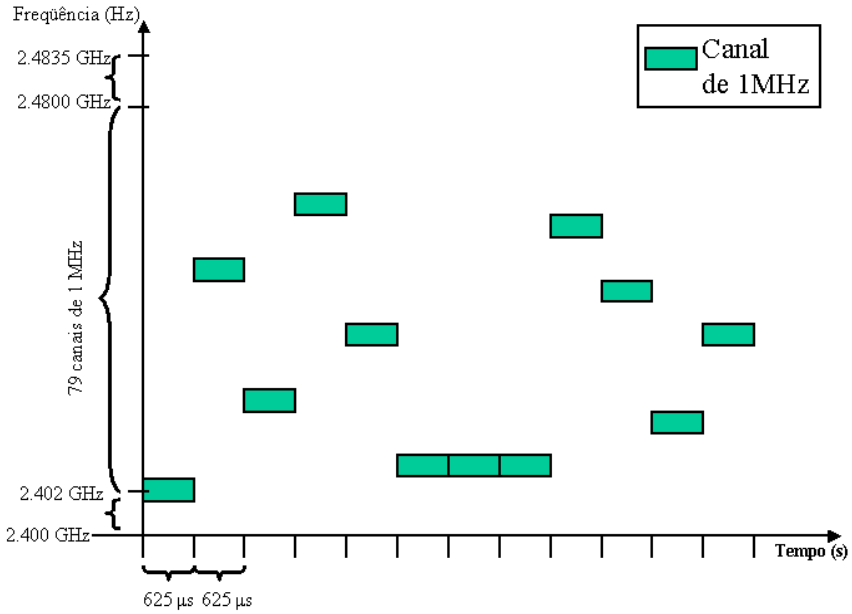


Figura 2.8 – Esboço de salto em freqüência, em verde o canal que está sendo utilizado em um determinado instante de tempo.

Uma outra técnica de espalhamento em freqüência é o chamado espalhamento espectral de seqüência direta (em inglês *Direct-Sequence Spread Spectrum* – DSSS), que utiliza instantaneamente uma largura de banda maior que a necessária. Equipamentos que implementem o IEEE802.11b (Wi-Fi) vão utilizar o DSSS e também devem operar na faixa ISM 2.4 GHz, o que se torna um bom exemplo de como ocorre a interferência de duas tecnologias que utilizem a mesma banda ISM. A figura 2.9 é um esboço de como o IEEE802.11b pode interferir quando há sobreposição da faixa de freqüência utilizada por uma transmissão *Bluetooth*. Embora haja interferência e o IEEE802.11b utilize uma potência muito maior para a transmissão (como visto na seção 1.1.3) do que o *Bluetooth*, deve ocorrer apenas uma redução da taxa média de transmissão e uma redução da qualidade de serviço para a comunicação via *Bluetooth*.

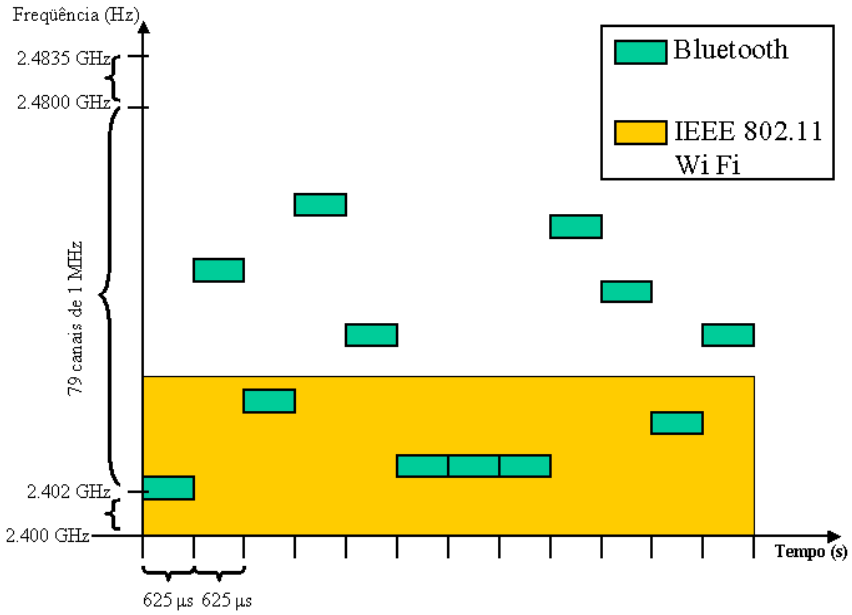


Figura 2.9 – Esboço de uma transmissão do IEEE802.11b (em laranja) interferindo numa transmissão Bluetooth (verde).

Assim terminamos de descrever o uso das faixas de freqüência ISM. A seguir serão descritos os requerimentos de um transmissor. A especificação indica quais os requisitos mínimos para a qualidade do transmissor e receptor, e indica também as condições para a realização de teste. Por exemplo, nos testes as temperaturas nominais estão entre 15 e 35 °C e são utilizados apenas 2 freqüências distintas (uma para a transmissão e outra para a recepção), ao invés das 79 possíveis.

No transmissor *Bluetooth*, a potência máxima permitida no conector da antena é especificada para três classes distintas. A tabela 2-2 introduz as classes de níveis de potência, que define a potência máxima permitida, além de sugerir valores para a potência mínima. Ainda é sugerido o uso de controle de potência para as classes 2 e 3. Para a classe 1 o controle de potência é obrigatório.

Tabela 2-2: Classes de potência:

Classe de Potência	Máxima Potência de Saída (valores obrigatórios)	Mínima Potência de Saída (valor sugerido)	Controle de Potência
1	100 mW	1 µW	Obrigatório
2	2.5 mW	1 µW	Opcional
3	1 mW	1 µW	Opcional

Transmissores de classe 1 obrigatoriamente utilizam o recurso de controle de potência. Isso pode ser feito se um receptor *Bluetooth* possuir o *Receiver Signal Strength Indicator* (RSSI), que é uma capacidade opcional nos receptores *Bluetooth*. O RSSI permite medir a potência recebida e enviar um pacote de controle avisando se o transmissor deve diminuir ou aumentar a potência. Se um receptor não tiver o RSSI implementado, um transmissor classe 1 deve reduzir a potência e operar como um rádio classe 2 ou classe 3 (que não requisitam de controle de potência).

A modulação utilizada por um transmissor *Bluetooth* deve ser a modulação GFSK – *Gaussian Frequency Shift Keying*. A modulação GFSK utilizada no *Bluetooth* é binária, na qual dois símbolos diferentes (o símbolo “0” e o símbolo “1”) são representados por sinais senoidais que diferem de um valor fixo de freqüência. Essa modulação é a mesma utilizada na modulação binária FSK (*Frequency Shift Keying*), com exceção de que o sinal de banda básica é filtrado por um filtro gaussiano (um filtro passa baixa), o que torna mais suave o sinal no domínio do tempo. Isso permite que a transmissão ocupe uma menor largura de banda em freqüência, reduzindo a interferência entre canais adjacentes. A taxa de símbolos é de 10^6 símbolos/s e cada símbolo corresponde a um bit, o que implica numa taxa de 10^6 bits/s.

Um receptor *Bluetooth* deve ter um nível de sensibilidade de –70 dBm ou melhor, e nesse nível deve ter uma taxa de erros de bit de 10^{-4}. A especificação em [6] também apresenta qual o desempenho mínimo que um receptor deve ter para uma dada potência de sinal desejado e interferente. O receptor deve receber em apenas um canal de freqüência a cada intervalo de tempo, acompanhando os saltos em freqüência do transmissor e rejeitando sinais fora da banda de freqüência do canal atual.

2.2.4 – *Baseband*

O protocolo *Baseband* descreve as funcionalidades do *Link Controller*, apresentado na figura 2.5 da seção 2.2.2. Essas funcionalidades incluem:

- controle de acesso ao meio (*Medium Access Control* - MAC);
- suporte ao tráfego de voz em tempo real entre dispositivos *Bluetooth;*
- *suporte ao tráfego de dados entre dispositivos Bluetooth;*
- comunicação *ad hoc* entre dispositivos *Bluetooth;*

Nesta seção, vários aspectos do protocolo *Baseband* serão abordados. Descreve-se como o controle de acesso ao meio é realizado, como uma rede de comunicação se estabelece entre dispositivos *Bluetooth*, os formatos de pacotes do protocolo *Baseband*, entre outros. No entanto não será feita nenhuma análise dos dispositivos *Bluetooth* em relação a segurança, como criptografia e administração de chaves de criptografia. Na implementação desse projeto, todas as opções de segurança do dispositivo *Bluetooth* são desativadas.

2.2.4.1 – Controle de Acesso ao Meio e Canais Físicos

O objetivo do controle de acesso é compartilhar um canal físico entre vários dispositivos Bluetooth. No entanto temos que definir primeiro o que é um canal físico. O sistema *Bluetooth* utiliza FHSS (*Frequency Hopping Spread Spectrum*), como visto na seção 2.2.3. No FHSS do *Bluetooth*, usualmente os saltos em freqüência ocorrem a cada intervalo de 625 μs (a exceção ocorre quando temos que transmitir pacotes que usam múltiplos de intervalos de 625 μs), sendo 79 canais disponíveis ao todo. Uma seqüência de saltos conhecida por vários dispositivos sincronizados é definida como um canal físico. Em um canal físico, apenas 8 dispositivos podem estar ativos (transmitindo e recebendo dados). Dispositivos próximos, que utilizem uma seqüência de saltos sincronizada e distinta, estão fazendo uso de um outro canal físico.

O controle de acesso ao meio é feito através de um protocolo de revezamento, chamado de protocolo de *polling*, no qual um dos dispositivos *Bluetooth* é estabelecido como mestre e os outros serão escravos. O dispositivo mestre envia uma mensagem para um dos escravos, dando a permissão para que este transmita um pacote *Baseband* no canal físico e terminada a transmissão desse escravo, o mestre escolhe outro escravo para permitir a transmissão e isso prossegue de maneira cíclica, de modo que todos os dispositivos tenham a possibilidade de transmitir dados. Nesse caso o controle de acesso ao meio é centralizado pelo mestre, que pode ser vir a ser qualquer um dos dispositivos *Bluetooth* no momento da configuração dessa rede de comunicação. Uma vez selecionado qual dispositivo é o mestre, os dispositivos escravos devem estar sincronizados com o mestre e a seqüência de saltos é obtida a partir do endereço *Bluetooth Device Address* – BD_ADDR do dispositivo mestre. O BD_ADDR é um endereço de 48 bits que é único para cada módulo *Bluetooth*. Essa seqüência de saltos é pseudo-aleatória, de longa duração (não repete um padrão em um curto espaço de tempo) e

utiliza os 79 canais de 1 MHz. Será visto posteriormente, que esse procedimento de se utilizar um endereço para determinar a seqüência de saltos, também é utilizado para buscar novos dispositivos e para realizar chamadas para a conexão. A busca e chamada serão abordadas quando forem descritos os canais físicos de *Inquiry Scan* e *Page Scan*, no item 2.2.4.5.

2.2.4.2 – *Piconets* e *Scatternet*

Quando existem apenas 2 dispositivos *Bluetooth* se comunicando, temos uma conexão ponto a ponto. Quando temos mais de 2 dispositivos, temos uma conexão de ponto a multiponto. A figura 2.10 ilustra esses tipos de conexões.

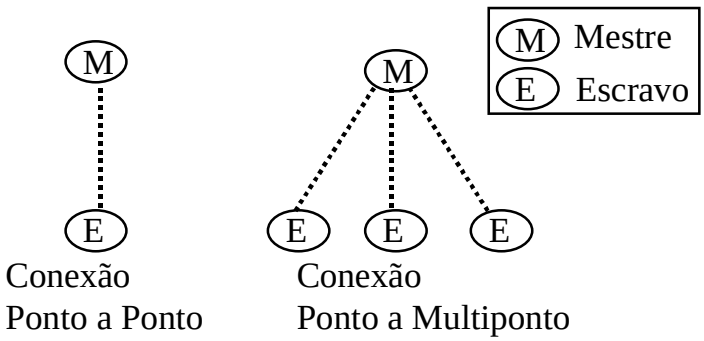


Figura 2.10 - Redes de comunicação entre dispositivos *Bluetooth*, conexões ponto a ponto e ponto a Multiponto.

Um mestre pode estar conectado a um número ilimitado de dispositivos, no entanto num mesmo canal físico apenas 7 escravos podem estar ativos (trocando dados). Um dispositivo conectado é definido como conectado no modo *Park* se não vai transmitir nem receber dados, mas se mantém sincronizado ao mestre. Dispositivos ativos em um mesmo canal físico formam o que se chama de *piconet*. Quando temos múltiplas *piconets* (cada uma usando um canal físico distinto) dentro de uma mesma área, com nós em comum, chamamos a isso de *scarttenet*. A figura 2.11 apresenta uma scartternet com 3 piconets.

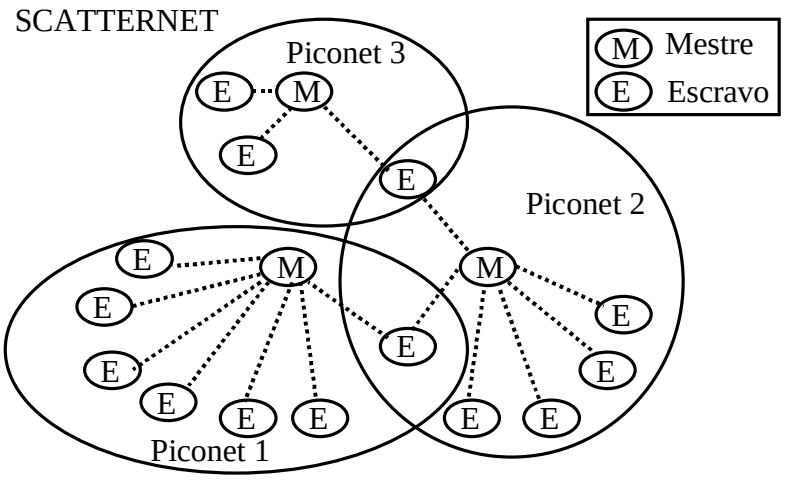


Figura 2.11 – Ilustrando 3 piconets que formam uma scatternet.

Uma pergunta que pode surgir é como um escravo se comunica com outro escravo dentro de uma piconet. A topologia de uma piconet permite que um escravo apenas se comunique diretamente com um mestre. Será visto na seção 2.2.4.7 que um pacote *Baseband* não tem uma estrutura de dados que permita que um escravo anexe o endereço de outro para que haja um roteamento do mestre para o outro escravo. Uma saída seria colocar essa informação na carga útil do pacote, que somente poderia ser utilizada para roteamento por um protocolo de camada superior. Outra solução é um escravo de uma piconet A se conectar diretamente com outro escravo que também é da piconet A, formando uma nova piconet B, onde o primeiro dispositivo seria o mestre e o segundo o escravo na piconet B.

2.2.4.3 – *Time Division Duplex- TDD*

Em um canal físico, os intervalos de tempo são enumerados e usualmente o mestre transmite nos intervalos pares, enquanto que um dispositivo escravo transmite nos intervalos ímpares. Nos intervalos pares, o mestre transmite dados e indica qual escravo pode transmitir no próximo intervalo. Esse tipo de comunicação multiplexada no tempo é chamado de *Time Division Duplex* (TDD). A cada intervalo de tempo (de 625 μs), uma nova freqüência é utilizada, caso o pacote *Baseband* transmitido ocupe apenas um único intervalo. No entanto, uma transmissão de um pacote *Baseband* pode ocupar 1, 3 ou 5 intervalos de 625 μs e durante toda a transmissão de um pacote, a freqüência não deve mudar. A figura 2.12 ilustra o TDD e a transmissão de pacotes de duração de 3 intervalos de tempo. A seqüência f(n), f(n+1), f(n+2), …, indicada na figura 2.12, representa a seqüência de freqüências utilizadas no canal físico.

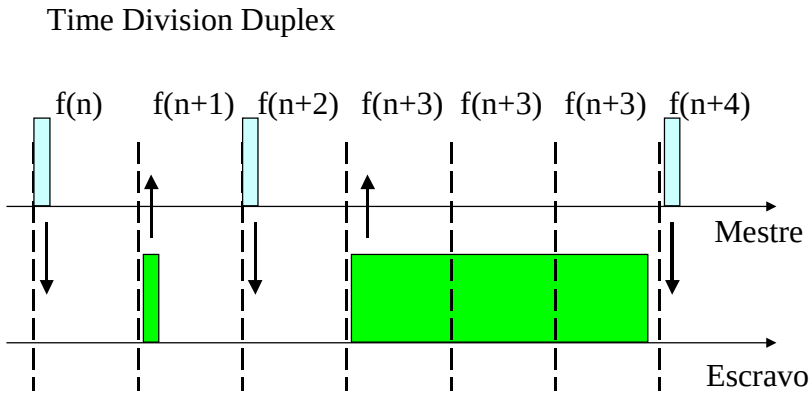


Figura 2.12 – Comunicação entre um Mestre e um Escravo, utilizando o Time Division Duplex. Note que na transmissão de pacotes de múltiplos intervalos de duração, a freqüência não muda.

2.2.4.4 – Estados e Subestados

Até a seção 2.2.4.3 somente foram apresentados cenários em que os dispositivos já estão conectados e sincronizados com o mestre da piconet. Nesses casos, os dispositivos conhecem o *Bluetooth Device Address* – BD_ADDR do mestre. No entanto, inicialmente um dispositivo Bluetooth não pertence a piconet alguma, nem compartilha um canal físico com um dispositivo e pode não conhecer um endereço BD_ADDR de novos dispositivos. Para enumerar as etapas necessárias para a

descoberta e a conexão com outros dispositivos, vamos introduzir os Estados e Subestados que o protocolo *Baseband* pode assumir.

No protocolo *Baseband* são definidos 2 estados, o estado *Connection* e o estado *Standby*. No estado Connection, o dispositivo está conectado e a outro e portanto pertence a uma piconet. No estado Standby, o dispositivo está em modo de baixo consumo e a única atividade é atualização do Relógio nativo, e não está conectado a nenhum dispositivo.

Além dos 2 estados, são também definidos 7 subestados: *Inquiry*, *Inquiry Scan*, *Inquiry Response*, *Page*, *Page Scan*, *Page Response*, *Slave Response* e *Master Response*. O que é de interesse é entender qual a finalidade desses subestados. Se tivéssemos 2 dispositivos *Bluetooth*, nomeados de disp1 e disp2, ambos no estado *Standby*, a seqüência de passos 1 e 2 devem ocorrer para que o disp1 se conecte ao disp2. As figuras 2.13 e 2.14 auxiliam o texto a seguir, indicando com os índices (em vermelho) a ordem descrita no texto:

- Passo 1: Pesquisa por novos dispositivos.

Objetivo: Descobrir se há um novo dispositivo próximo e qual o seu endereço BD_ADDR.

Descrição: (1) O disp1 sai do estado *Standby* e entra no subestado *Inquiry*, iniciando uma pesquisa por novos dispositivos. O disp1 deve permanecer nesse subestado por 10.24 segundos, transmitindo e escutando de forma alternada no tempo (usa o TDD). O canal físico (seqüência de saltos) utilizado é derivado de um BD_ADDR reservado e conhecido por todos dispositivos, o que será explicado na seção 2.2.4.5. Nas transmissões, um pacote *Baseband* de curta duração, chamado de ID (*Identity*), utiliza um campo de identificação derivado desse BD_ADDR reservado. (2) O disp2 entra no subestado *Inquiry Scan*. (3) O disp2 receberá a pesquisa (um pacote ID de *Inquiry*), mas não deverá responder imediatamente. Outros dispositivos também podem ter recebido a pesquisa e uma resposta imediata de vários dispositivos resultaria em colisão e perda da resposta, pois o meio é compartilhado (todos usariam a mesma freqüência para responder). (4) Então ao receber o pacote ID de pesquisa, o disp2 sai do subestado *Inquiry Scan* e depois de um tempo aleatório retorna a esse mesmo subestado. (5) Se escutar a pesquisa novamente, deve entrar no subestado *Inquiry Response* e enviar um pacote FHS (*Frequency Hop Synchronization*, descrito na seção 2.2.4.7) como resposta. (6) Mesmo que respostas sejam recebidas (pacotes FHS), o disp1 persiste no subestado *Inquiry* até o tempo de 10.25s expirar. (7) O disp2 volta a subestado Inquiry Scan e posteriormente, em (8) o disp2 volta ao estado Standby. (9) Expirado os 10.24 s, o disp1 cessa a pesquisa e volta ao estado Standby. A figura 2.13 ilustra o passo 1. Um dispositivo que receber um pacote ID com um identificador derivado do BD_ADDR reservado sabe que o outro dispositivo está no subestado de *Inquiry*. O pacote *Baseband* chamado de FHS (*Frequency Hop Synchronization*) contém informações como o BD_ADDR e o relógio do dispositivo que envia o FHS.

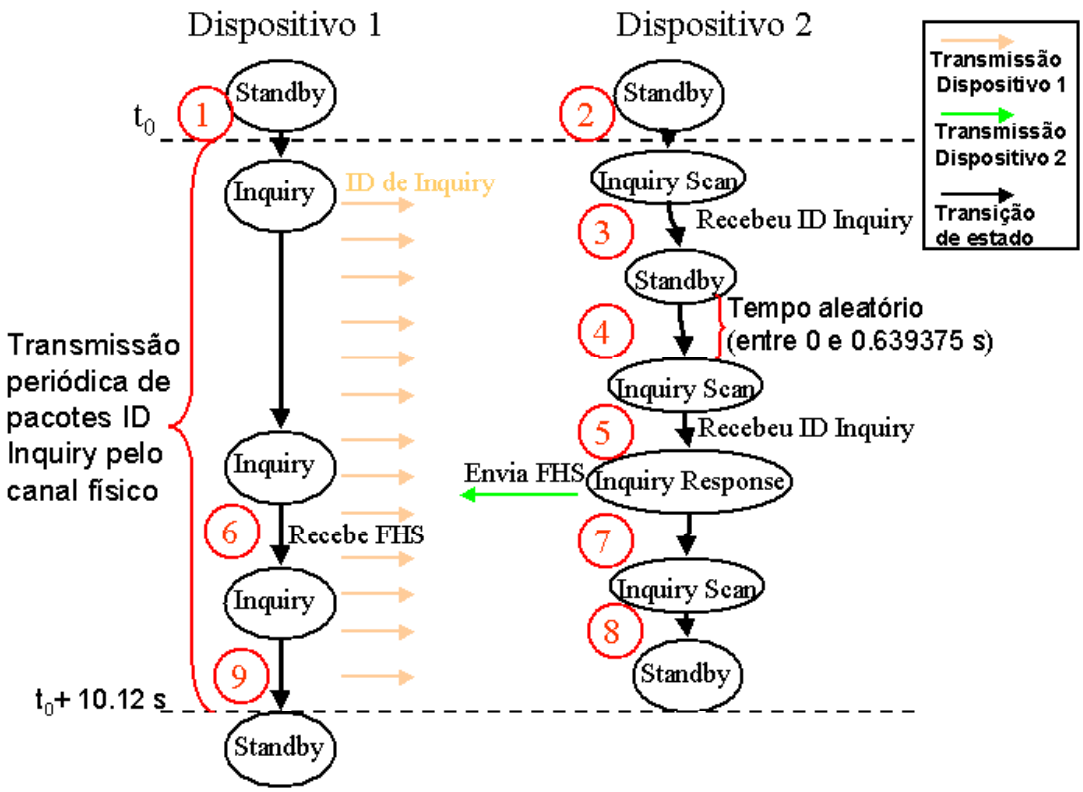


Figura 2.13 – Ilustração dos estados e subestados utilizados na pesquisa por novos dispositivos (Passo 1). O estado Inquiry deve durar pelo menos 10.12s, transmitindo periodicamente um mesmo tipo de pacote ID. Depois do dispositivo 2 receber o primeiro pacote ID, espera-se por um tempo aleatório para tentar responder a pesquisa com um pacote FHS.

- Passo 2: Chamada.

Objetivo: Utilizar o BD_ADDR do disp2, adquirido no passo 1, para realizar uma chamada. Durante a chamada são trocadas informações para a inicialização de uma conexão.

Descrição: (1) O disp1 sai do estado *Standby* e entra no subestado *Page*. Nesse subestado, o disp1 transmite periodicamente pacotes *Baseband* do tipo ID, como no caso descrito no passo 1 para o subestado *Inquiry*. A diferença é o que o campo identificador do pacote ID e o canal físico são derivados do BD_ADDR do disp2. (2) O disp2 estiver no subestado *Page Scan*. (3) O disp2 recebe o pacote ID de *Page*, entrando imediatamente no subestado *Slave Response*. A resposta do disp2 é dada com o mesmo pacote ID recebido. (4) Então, o disp1 entra no subestado *Master Response* e envia um pacote FHS . (5) O disp2 recebe o pacote FHS e responde de novo com o mesmo pacote ID. (6) O disp1 recebe o pacote ID e a partir daí os dispositivos estão conectados e entram no estado *Connection*, formando uma piconet. O dispositivo que inicia a chamada (no caso disp1) será o mestre da piconet. Durante os subestados de *Master Response e Slave Response* os dispositivos trocam informações importantes, como por exemplo informações de sincronização e um endereço temporário de 3 bits AM_ADDR (que é descrito na seção 2.2.4.7). A figura 2.14 ilustra o passo 2.

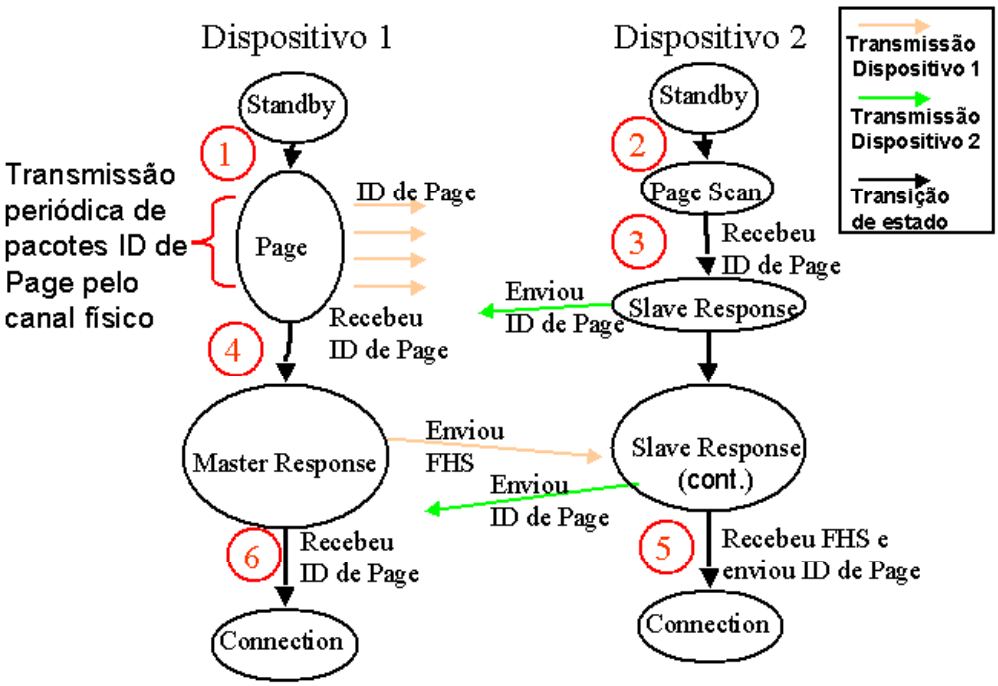


Figura 2.14 – Ilustração dos estados e subestados utilizados na chamada e conexão, para o exemplo citado.

Os procedimentos de pesquisa e chamada foram exemplificados para um caso possível. Também é possível que um dispositivo já conectado numa piconet (portanto já no estado *Connection*), deixe o estado *Connection* e entre nos subestados *Inquiry, Inquiry Scan, Page* ou *Page Scan*. Esse caso será abordado na seção 2.2.4.6.

Quando uma pesquisa ou uma chamada é feita, a especificação recomenda a liberação do máximo de recursos possíveis e isso afeta a transmissão de dados na piconet.

Outro cenário possível é iniciar uma conexão com um dispositivo sem realizar o *Inquiry*, por exemplo se já tivermos conhecimento do BD_ADDR do dispositivo.

As figuras 2.13 e 2.14 são uma representação bem simplificada. A seção 2.2.4.5 volta a abordar os subestados *Page* e *Inquiry,* e explica como derivar os canais físicos a partir de endereços BD_ADDR e como um dispositivo consegue realizar uma pesquisa e chamada se os dispositivos não estão sincronizados.

2.2.4.5 – Endereçamento e Canais Físicos *Piconet, Page Scan e Inquiry Scan*

Na seção 2.2.4.1, foi apresentada a definição de um canal físico e que a seqüência de saltos é obtida a partir do BD_ADDR (*Bluetooth Device Address*) do dispositivo mestre da *piconet*. Foram introduzidos também na seção 2.2.4.4, dois pacotes *Baseband* chamados de ID (*Identity*) e de FHS (*Frequency Hop Synchronization*). O pacote ID tem um campo de identificação derivado do BD_ADDR e o FHS é utilizado para informar outro dispositivo sobre o BD_ADDR e o relógio. Agora vamos apresentar de uma maneira mais geral para o uso do endereço BD_ADDR e como esses

endereçamentos auxiliam na descoberta de outros dispositivos (*Inquiry*) e na inicialização de uma conexão (*Page*).

A todo dispositivo *Bluetooth* deve ser atribuído um endereço único de 48 bits, chamado de BD_ADDR. A figura 2.15 apresenta o formato de um BD_ADDR, que é dividido em 3 campos: o campo LAP (*Lower Address Part*) com 24 bits, o campo UAP (*Upper Address Part*) com 8 bits e o campo NAP (*Non-significant Address Part*) de 16 bits. Apenas o LAP e o UAP têm importância para o endereçamento, o que significa que existem no total número 2^{32} endereços possíveis. Desses 2^{32} endereços, 64 são reservados e não podem ser atribuídos a nenhum dispositivo.

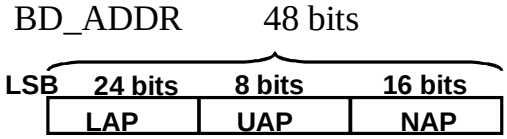


Figura 2.15 – Formato do BD_ADDR

O BD_ADDR é utilizado de 2 formas no protocolo *Baseband*: para gerar uma seqüência de saltos (um canal físico) e para identificação de um pacote *Baseband*. No primeiro caso o campo LAP de 24 bits e parte do campo do UAP são utilizados, no segundo caso, a identificação é derivada apenas do campo LAP.

Para gerar a seqüência de saltos em freqüência de uma piconet, os primeiros 28 bits do BD_ADDRESS (LAP + os 4 bits menos significativos do UAP) são processados junto com os 27 bits mais significativos do registrador do relógio, ilustrado na figura 2.16. O bit menos significativo do registrador de relógio (que funciona como um contador) inverte a cada 312.5 μs, e o segundo bit menos significativo inverte a cada 625 μs.

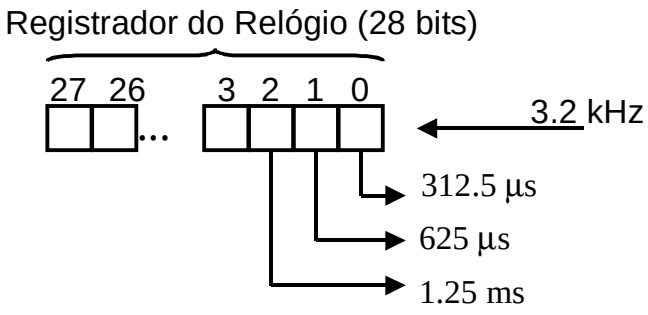


Figura 2.16 – Registrador de 28 bits do relógio

A figura 2.17 ilustra que os campos LAP e UAP (apenas os 4 bits menos significativos) são utilizados com os 27 bits mais significativos do registrador do relógio para gerar um número, que é um índice para um dos 79 canais de freqüência. Utilizando-se apenas os bits 1 a 27 do registrador de relógio da figura 2.16, temos que a cada 625 μs o conteúdo do registrador muda, mudando também o índice para a freqüência que deve ser utilizada no próximo salto. Na figura 2.17, o bloco chamado de "Caixa Seletora" faz operações lógicas com os 28 bits do BD_ADDR e os 27 bits do registrador do Relógio, para gerar a seqüência pseudo-aleatória de saltos.

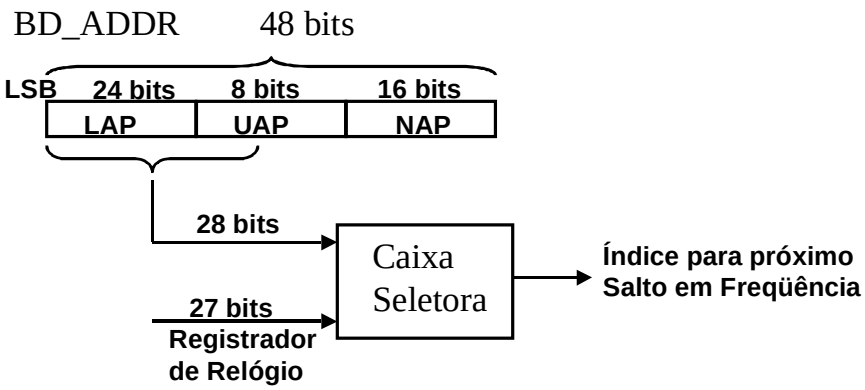


Figura 2.17 – A Caixa Seletora faz operações lógicas com os bits do BD_ADDR e do registrador do Relógio, para gerar a seqüência de saltos em freqüência.

São utilizados 27 bits do registrador do Relógio para gerar uma seqüência pseudo-aleatória de saltos de uma piconet. Mas nos canais físicos de *Page* e *Inquiry* são usados todos os 28 bits do registrador, pois os saltos em freqüência ocorrem a cada 312.5 μs, e não em intervalos de 625 μs (o número de saltos em freqüência por segundo é dobrado). O bloco Caixa Seletora da figura 2.17 também tem bits de controle que ajudam a selecionar a faixa de freqüência utilizada. Em um canal físico de uma piconet, são utilizadas todas as 79 freqüências disponíveis e as seqüências de saltos são de longa duração, ou seja, não repetem um padrão em um curto intervalo de tempo. Em canais físicos *Inquiry, Inquiry Scan, Page* e *Page Scan* são utilizadas apenas 32 freqüências e a cada 32 saltos a seqüência é repetida, o que facilita a sincronização.

Quando os dispositivos não estão sincronizados, mas utilizam a mesma seqüência de saltos, a seqüência de saltos utilizada por um dispositivo pode estar atrasada em relação a seqüência de saltos do outro dispositivo e nunca haveria a troca de dados. A solução utilizada no *Bluetooth* é: um dispositivo no subestado *Inquiry* ou *Page* irá saltar de freqüência 2 vezes em um mesmo intervalo de 625 μs e um outro dispositivo no subestado *Inquiry Scan* ou *Page Scan* deve estar escutando sempre numa mesma freqüência do canal físico correspondente por um período suficiente grande para ouvir por chamadas ou pesquisas. Isso permitiria que em um tempo limitado ambos os dispositivos estivessem transmitindo/recebendo na mesma freqüência e assim seja possível trocar dados de pesquisa (*Inquiry* ou *Page*).

O que resta determinar é qual endereço BD_ADDR utilizar para derivar a seqüência de saltos em cada caso. Isto é listado a seguir:

- no caso de pesquisa, nos subestados *Inquiry* e *Inquiry Scan*, os dispositivos utilizam um endereço BD_ADDR reservado e conhecido por todos, para determinar o canal físico. O BD_ADDR utilizado no *Inquiry* tem o campo LAP = 0x9E8B33 e o campo UAP = 0x00 (ver [22]);
- no caso de uma chamada para a conexão, no subestado *Page*, um dispositivo utiliza o BD_ADDR do dispositivo com o qual deseja iniciar uma conexão. O BD_ADDR provavelmente foi obtido durante uma pesquisa. Um dispositivo, no subestado *Page*

Scan, utiliza o próprio BD_ADDR para determinar o canal físico e procurar por chamadas;

- no caso da piconet, o canal físico é determinado pelo BD_ADDR do mestre;

O endereço BD_ADDR também é utilizado para que um receptor descubra se um pacote *Baseband* que foi transmitido por outro dispositivo deve ser recebido ou ignorado. Para isso, todo pacote *Baseband* possui um campo inicial chamado de *Access Code*, constituído de 72 bits ou 68 bits. Desses 72 ou 68 bits, 64 bits são uma seqüência pseudoaleatória construída a partir do campo LAP do endereço BD_ADDR de destino. Essa seqüência de 64 bits é construída de forma que endereços LAP diferentes resultem em seqüências de 64 bits com distância de Hamming suficientemente grandes. Isso permite que um receptor possa identificar corretamente se o código de acesso se refere ao seu próprio LAP ou não. No receptor é feita uma comparação bit a bit entre o código de acesso esperado e os últimos 72/68 bits recebidos (janela deslizante de 72/68 bits). Quando um limiar de acertos for ultrapassado, o receptor utiliza essa informação para se sincronizar e começar a receber o resto do pacote *Baseband*. Se o limiar não for ultrapassado, o receptor não tem que processar mais nenhum dado. Num canal físico de uma piconet, o código de acesso é sempre o mesmo, sendo derivado do LAP do BD_ADDR do mestre.

A figura 2.16 apresenta um exemplo do uso do código de acesso. Nesse exemplo temos 3 dispositivos: Disp 1, Disp 2 e Disp 3. Ainda na figura 2.16, f(n) representa uma freqüência das 79 possíveis para recepção/transmissão. O dispositivo Disp 1 está no subestado *Page* e quer se conectar com o Disp 3. Para isso, Disp 1 deriva o código de acesso utilizando o BD_ADDR do Disp 3 e passa transmitir pacotes ID (contendo o código de acesso) e transmite em um mesmo intervalo de 625 μs em duas freqüências diferentes, representadas por f(n) e f(n+1). O Disp 3 está no subestado *Page Scan* e escuta em uma única freqüência f(n+1). O Disp 2 também está no *Page Scan* e escuta numa freqüência f(n) em que o Disp 1 transmite. Pela figura 2.16, tanto o Disp 2 e o Disp 3 vão receber o código de acesso. No entanto o Disp 2 não conseguirá identificar como sendo seu o código que recebeu e irá ignorar o pacote recebido. Já o Disp 3 identifica como sendo este o seu próprio código de acesso e entre no subestado *Slave Response*, para então se conectar com Disp 1.

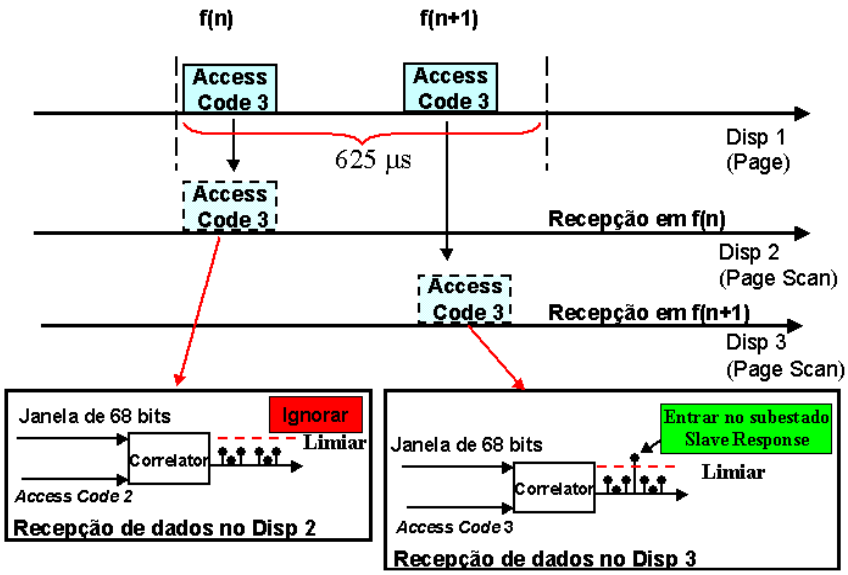


Figura 2.16 – Uso de Códigos de Acesso para a identificação do destinatário de um pacote (para um *Access Code* de 68 bits).

Uma questão a ser levantada é: o que acontece com dispositivos com campos LAP iguais e campos UAP diferentes. Significa que o BD_ADDR é diferente, mas os códigos de acesso são iguais (pois tem o mesmo LAP). Para evitar identificações erradas, além dos códigos de acesso, existe um esquema de verificação de erro que testa não apenas a integridade dos dados recebidos, mas também se o UAP corresponde ao esperado. Na seção 2.2.4.7, os campos de verificação de erro de um pacote *Baseband*, chamados de HEC- *Header Error Check* e CRC – *Cyclic Redundancy Check*, serão apresentados. Um canal físico utiliza os campos LAP e UAP para a seleção dos saltos, portanto não tem o mesmo problema dos códigos de acesso. Um exemplo: piconets diferentes que utilizam o mesmo *Access Code* nunca vão usar o mesmo canal físico. E mesmo que um pacote de uma piconet fosse recebido pela outra, a verificação por erro impediria que o pacote fosse aceito, pois a UAP não seria igual a do mestre da piconet (lembrando ainda que um dispositivo somente pode ser mestre de uma única piconet).

Agora vamos detalhar o procedimento utilizado durante os subestados *Inquiry*, *Inquiry Scan e Inquiry Response*, baseado na ilustração da figura 2.17. Um dispositivo que entra no subestado *Inquiry* ou *Inquiry Scan*, vai utilizar um endereço BD_ADDR reservado LAP = 0x9E8B33 e UAP = 0x00 (especificado em [22]) para obter a seqüência de saltos e o *Access Code*. A seqüência de saltos resulta numa seqüência de apenas 32 canais dos 79 possíveis, com periodicidade de 32 saltos. Seja f(i) é i-ésima freqüência da seqüência de saltos, para i = 1, 2, …, 32. Cada f(i) representa uma freqüência distinta. Saltos em freqüência ocorrem na ordem f(1), f(2),…, f(32), f(1), f(2),…, de forma periódica. O dispositivo 1, no subestado *Inquiry*, transmite pacotes *Baseband* ID com código de acesso derivado do BD_ADDR reservado. A transmissão segue a seguinte ordem: 2 transmissões consecutivas em f(i) e f(i+1) com intervalos entre saltos de 312.5 µs e espera por resposta em f(i) e f(i+1). O dispositivo 2, no subestado *Inquiry Scan*, compara uma janela deslizante os últimos 68 bits recebidos com o *Access Code* esperado e utiliza uma única freqüência em f(j) (na figura 2.17, j = 14). Se o dispositivo 1 transmitir na freqüência j, o dispositivo 2 irá receber o pacote ID e sairá do subestado *Inquiry Scan* por

um tempo aleatório $t_{aleatório}$. Depois de $t_{aleatório}$ segundos, o dispositivo 2 entra de novo no subestado *Inquiry Scan* e se receber novamente um pacote ID do dispositivo 1, deve responder com um pacote FHS (*Inquiry Response*) em 625 µs após a recepção do pacote ID e na freqüência f(j). Os dispositivos 1 e 2 devem liberar o máximo possível de recursos para realizar os procedimentos de pesquisa.

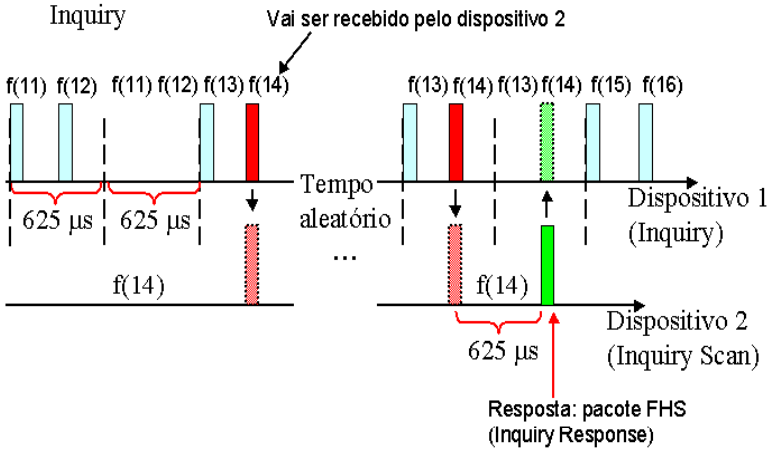


Figura 2.17 – Canal físico *Inquiry*. f(i) denota a i-ésima freqüência da seqüência de 32, para i = 1, 2, …, 32.

No caso dos subestados *Page, Page Scan, Master Response* e *Slave Response*, o procedimento é semelhante ao utilizado no *Inquiry* e *Inquiry Scan*. Uma diferença é que o BD_ADDR utilizado para obter o canal físico e o Access Code é o endereço do dispositivo com o qual se quer conectar. A outra diferença é que apenas um dispositivo deve responder a esse código de acesso, portanto não é necessário esperar por um tempo aleatório para responder a chamada. A figura 2.18 ilustra 2 dispositivos, onde o dispositivo 1 tem o BD_ADDR do dispositivo 2 e tenta se conectar entrando no subestado *Page*. O canal físico utiliza apenas 32 freqüências e tem período de 32. O dispositivo 2 escuta em uma única freqüência f(j) e espera por um pacote ID com o seu código de acesso, sempre verificando os últimos 68 bits recebidos. Se o dispositivo 2 receber um pacote ID, poderá responder 625 µs após a recepção, retransmitindo o pacote ID na mesma freqüência f(j). Nesse caso, pode-se assumir que os dispositivos estão sincronizados e 312.5 µs ambos saltam para a freqüência f(j+1), e o dispositivo 1 transmite um pacote FHS. Finalizando a troca de informações, o dispositivo 2 recebe o pacote FHS e transmite em f(j+1) de novo um pacote ID. O dispositivo 2 sai do canal físico derivado do seu BD_ADDR e passa a utilizar o BD_ADDR do dispositivo 1 para obter o novo canal físico p(n). Ao receber o pacote ID, o dispositivo 1 se torna o mestre dessa piconet e passa a utilizar p(n). O primeiro pacote transmitido na freqüência p(1) indica o início da conexão, com o mestre transmitindo o primeiro pacote para o escravo.

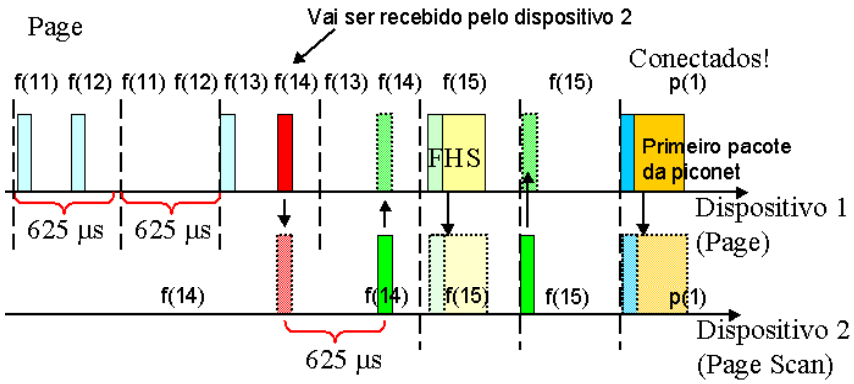


Figura 2.18 - Canal físico *Page*. f(i) pertence ao canal físico *Page*. p(n) pertence ao canal físico da piconet.

2.2.4.6 – Transportes Lógicos: *Synchronous Connection-Oriented* – SCO e *Asynchronous Connection-Less* – ACL.

Durante uma conexão entre dispositivos *Bluetooth*, dois tipos de transportes lógicos podem ser estabelecidos entre um mestre e os escravos de uma piconet. Em cada tipo de transporte lógico são utilizados pacotes *Baseband* diferentes, que devem dar suporte a aplicações diferentes, como por exemplo aplicações de voz em tempo real ou aplicações de dados. No primeiro caso uma taxa constante e latência limitada são necessários para a transmissão de voz, o que pode ser atendido se houver reserva de recursos. No segundo caso, a transferência de dados requer que a integridade dos dados seja assegurada e a vazão deve ser a maior possível.

Um dos transportes lógicos é chamado de *Synchronous Connection-Oriented* – SCO. Uma conexão que utilize SCO deve ser ponto a ponto (entre o mestre e um único escravo). No SCO é feita uma reserva dos intervalos de tempo em que o escravo ou o mestre transmite, de forma que transmissão seja periódica. É negociado entre o mestre e um escravo um período T_{SCO}, que é o período da transmissão, garantindo assim uma taxa constante. Os pacotes *Baseband* que utilizam o SCO nunca são retransmitidos. A reserva de recursos é utilizada tipicamente em redes de comutação de circuito, como nas redes de telefonia fixa. Numa piconet, no máximo 3 conexões com SCO podem existir simultaneamente e a taxa máxima de transmissão é de 64 kb/s (valor típico para a transmissão de voz). O SCO não transporta dados vindos das camadas de protocolos superiores LMP e L2CAP. Retornando a seção 2.2.1, o perfil *Headset* é apresentado novamente na figura 2.19, destacando-se qual transporte lógico cada camada utiliza numa conexão. Note na figura 2.19 que a camada *Headset Control* tem acesso a camada *Baseband*, para a transmissão dos dados de áudio diretamente pelo SCO. Nesse trabalho, enlaces SCO não serão utilizados, mas no capítulo 8, seção 8.4, vamos citar trabalhos futuros que possam envolver o seu uso.

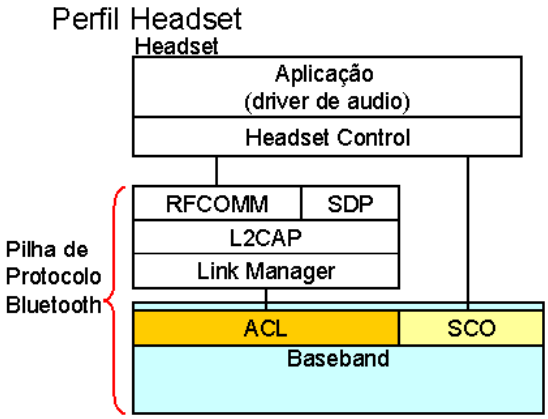


Figura 2.19 – Perfil *Headset* – Transporte de dados de áudio pelo SCO. Transporte de dados de controle e comandos pelo ACL.

O outro transporte lógico é chamado de *Asynchronous Connection-Less* – ACL. No transporte lógico ACL a prioridade é a integridade dos dados e a máxima utilização dos recursos, como por exemplo a transmitir na maior taxa disponível. Por isso não há reserva de recursos como no SCO e os dados sempre são verificados e retransmitidos se necessário. Conexões ACL conectam um mestre a vários escravos (enlace ponto a multiponto), na qual um mestre pode trocar dados com qualquer um dos escravos, utilizando o *Time Division Duplex* e o controle de acesso ao meio com revezamento (*Polling*). Além disso um mestre pode transmitir dados em *broadcast*, ou seja, todos os escravos ativos da piconet vão receber uma transmissão do mestre. Numa conexão utilizando o transporte ACL, uma conexão pode alcançar a taxa máxima (assimétrica) de 723.2 kb/s. As camadas superiores L2CAP e LMP utilizam o ACL para o transporte de suas PDUs (*Protocols Data Units*), que são encapsulados dentro da carga útil de um pacote *Baseband*.

Comparando-se os dois tipos de transporte, 6 considerações importantes são listadas:

- Não pode haver uma conexão SCO sem que haja uma conexão ACL;
- Não é obrigatório o suporte ao SCO em dispositivos Bluetooth;
- O transporte lógico ACL utiliza os recursos que não foram reservados por uma conexão que utilize um transporte lógico SCO. Significa que as taxas de transmissão de conexões ACL podem ser afetadas por uma conexão SCO, mas não o contrário;
- Um pacote que é recebido corrompido no SCO nunca é retransmitido. Deve-se analisar se determinadas aplicações em tempo real suportam a opção de ter dados descartados sem aviso. Exemplo: poderia ser inconveniente se um alarme não fosse entregue, mesmo que tardiamente. Logo a aplicação também teria que garantir a integridade dos dados e fazer retransmissões;
- Durante um procedimento de pesquisa (nos subestados *Inquiry* ou *Inquiry Scan*) ou chamada (nos subestados *Page* ou *Page Scan*), a especificação estabelece que uma conexão SCO não deve ser interrompida, mas recomenda que uma conexão ACL seja colocada no modo *Park* (modo em que não se transmite ou recebe dados, mas se mantêm sincronizado a piconet), para que o máximo de recursos possa ser utilizado para realizar chamadas ou pesquisas. Isso pode comprometer a previsibilidade de tempo de

transmissões em conexões ACL e é incerto qual dispositivo implementa ou não essa recomendação;

- *Softwares* livres como o Bluez implementam protocolos de camadas superiores no sistema operacional Linux. São implementados os protocolos L2CAP (ver seção 2.2.7), Audio SCO, a interface HCI (ver seção 2.2.6), entre outros. O Bluez fornece suporte a programação via socket tanto para conexões ACL quanto SCO. Entretanto, para que se possa utilizar o SCO com o Bluez, 3 condições devem ser verificadas: 1) o módulo *Bluetooth* tem suporte ao SCO; 2) o módulo tem que ser capaz de rotear os pacotes do SCO pelo HCI; 3) a versão atual do HCI do Bluez somente suporta uma conexão SCO por vez e não três como na especificação;

2.2.4.7 – Pacotes *Baseband* e Taxas de Transmissão

De maneira geral, os tipos de pacotes da camada *Baseband* podem ser divididos em 3 grupos: pacotes comuns, pacotes ACL e pacotes SCO.

A figura 2.20, apresenta o formato geral de um pacote do protocolo Baseband. Um pacote possui 3 campos: *Access Code* (código de acesso), *Header* (cabeçalho) *e Payload* (carga útil). O pacote está representado no formato *Little Endian*, ou seja, o bit menos significativo é o mais a esquerda e este também é o primeiro bit ser transmitido no ar. Isso está indicado na figura 2.20. O primeiro campo a ser transmitido é *Access Code* (código de acesso), que está presente em todos os tipos de pacotes e contém um identificador de 64 bits derivado do BD_ADDR. O campo *Header*, ausente apenas num pacote ID, é utilizado nos pacotes para multiplexar entre os escravos, controle de fluxo e confirmação de pacotes (esses dois últimos apenas nos pacotes ACL). O último campo é o de carga útil (*Payload*) que transporta dados provenientes de camadas superiores como o LMP, L2CAP ou Audio, ou informações necessárias à conexão, como no caso do pacote FHS.

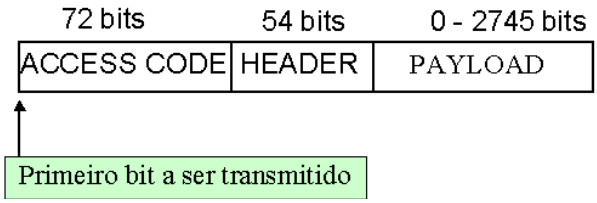


Figura 2.20 – Formato geral de um pacote *Baseband*.

O *Access Code* (código de acesso) está ilustrado na figura 2.21. Os primeiros 4 bits são do preâmbulo, utilizado para compensação DC. O Sync Word é uma seqüência pseudo-aleatória de 64 derivada do LAP do BD_ADDR, que é utilizado para a identificação do canal físico (*Inquiry*, *Page*, *Piconet*). O trailer de 4 bits está presente apenas se o pacote possuir cabeçalho (*Header*). O *Access Code* foi apresentado na seção 2.2.4.5. Um pacote ID contém apenas o *Preamble* e o *Sync Word*.

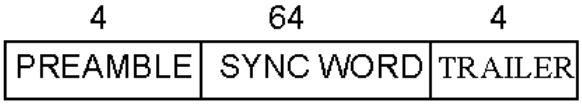


Figura 2.21 – Formato do Código de Acesso (*Access Code*) de 72(68) bits. O Sync Word é derivado de um BD_ADDR. Presente em todos os pacotes.

O campo *Header* (cabeçalho) da figura 2.20 tem tamanho fixo de 54 bits. Os 54 bits são formados a partir da repetição (3 vezes) dos 18 bits ilustrados na figura 2.22. A repetição é uma maneira de adicionar redundância aos dados e permitir a correção de erros pelo receptor, no esquema conhecido como FEC 1/3 (*Forward Error Correction 1/3*).

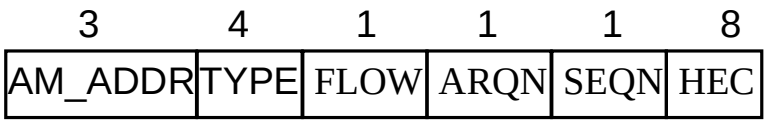


Figura 2.22 – Formato do cabeçalho (*Header*) de 54 bits. Esse campo tem 18 bits que são repetidos 3 vezes pelo esquema FEC 1/3. Não é utilizado em pacotes SCO.

O *Header* (cabeçalho) é dividido em 6 campos: AM_ADDR, TYPE, FLOW, ARQN, SEQN e HEC. O AM_ADDR é um endereço temporário para indexar até 7 escravos em uma piconet. Como numa piconet o *Access Code* é sempre o mesmo, faz-se necessário o uso do AM_ADDR para um escravo descobrir se o pacote é destinado a ele ou não. O AM_ADDR é passado ao escravo pelo mestre através do pacote FHS. O campo TYPE indica o tipo de pacote (existem 12 tipos), o que também vai informar quantos intervalos de 625 μs um pacote ocupa. O bit FLOW é usado no controle de fluxo, indicando com 0 para parar e com 1 para enviar mais pacotes (utilizado em pacotes ACL). O bit ARQN é utilizado para confirmar com 1 o sucesso de uma transmissão e com 0 a falha. SEQN é um bit que sempre muda a cada nova transmissão e permanece no mesmo valor em retransmissões, permitindo assim que pacotes duplicados sejam identificados e descartados. O campo HEC (*Header Error Check*) é um código de 8 bits gerado a partir dos 10 bits anteriores e do campo UAP do BD_ADDR. O HEC é usado para verificar se o *Header* tem algum erro ou se o campo UAP é corresponde ao esperado.

A figura 2.23 apresenta o *Payload*, que é dividido em outros 4 campos: L_CH, FLOW, LENGTH, PAYLOAD Body e CRC. O campo L_CH é constituído de 2 bits que indicam com 01 ou 10 que o pacote pertence a camada superior L2CAP (ver seção 2.2.6), e com 11 ao LMP. O bit FLOW é usado para o controle de fluxo dos dados de camadas superiores. O campo Payload Body é onde se encapsula fragmentos de pacotes L2CAP ou pacotes LMP e o seu tamanho máximo varia de acordo com o tipo de pacote, indicando a vazão efetiva para o transporte de dados de camadas superiores. O campo CRC (*Cyclic Redundancy Check*) é utilizado em pacotes para verificação de erro do *Payload* e para verificar o endereço UAP, da mesma forma que o campo HEC é utilizado no *Header*.

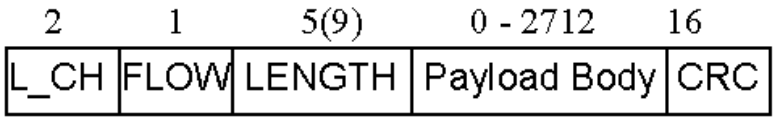


Figura 2.23 – Formato da carga útil (Payload)

O *Payload*, apresentado na figura 2.23, pode ser embaralhado (*whitening*) e protegido com códigos corretores de erro, como nos esquemas FEC 1/3 e FEC 2/3. Mas em troca, o campo PAYLOAD BODY deve diminuir para manter constante o tamanho máximo permitido. Significa que a utilização de FEC 1/3 ou FEC 2/3 reduz a necessidade de retransmissões, pois os erros são corrigidos no próprio receptor, mas isso também reduz a quantidade de carga útil transportada.

Uma rápida descrição dos 3 grupos de pacotes é feita a seguir: pacotes comuns, pacotes ACL e pacotes SCO.

Os pacotes comuns são:

- ID (*Identity*)
- FHS (*Frequency Hop Synchronization*)
- *NULL*
- *POLL*

Os pacotes do tipo ID (*Identity*) e FHS (*Frequency Hop Synchronization*) são utilizados para pesquisa (*Inquiry*) e chamada (*Page*), como descrito na seção 2.2.4.4 e 2.2.4.5. Os pacotes do tipo *NULL* e *POLL* são pacotes sem carga útil (*Payload*), utilizados numa piconet para auxiliar no esquema ARQ (*Automatic Repeat reQuest)* e no esquema de revezamento, respectivamente. Quando um mestre/escravo precisa confirmar um pacote recebido, o mestre/escravo pode transmitir a confirmação junto com os dados (esquema chamado de confirmação de carona ou *Piggybacking)*. Se não houver dados para transmitir, então um pacote NULL é transmitido apenas para informar o bit ARQN do Header. Por outro lado o pacote *POLL* é transmitido apenas pelo mestre, e um escravo que receba este pacote deve responder mesmo que não tenha nada para transmitir.

Os pacotes do tipo ACL são (não será explicado o pacote AUX1):

- DM1
- DM3
- DM5
- DH1
- DH3
- DH5

Pacotes DM indicam *Data Medium Rate* (taxa média de dados). São pacotes mais robustos, com codificação FEC 2/3 e uso de código CRC (*Cyclic Redundancy Check*) para correção e verificação de erros da carga útil (*Payload*), respectivamente. No FEC 2/3, 5 bits de paridade são adicionados a cada 10 bits de dados do *Payload*. Isso diminui a quantidade de dados de camadas superiores transmitida, em favor de uma maior proteção dos dados de um pacote *Baseband*. Os valores 1, 3 e 5 indicam quantos intervalos de 625 μs o pacote ocupa. Quanto maior o número de intervalos, maior é o tamanho máximo de um pacote.

Pacotes DH indicam *Data High Rate* (taxa alta de dados). São pacotes com menor proteção, pois apenas utilizam o código CRC para verificar erros. Os valores 1, 3 e 5 indicam novamente o

número de intervalos de 625 μs que o pacote ocupa. Os tamanhos máximos de *Payload* não variam em relação aos pacotes DM (1,3 e 5), a diferença é que sem o FEC 2/3 há mais espaço no pacote para transportar dados (no *Payload Body*) ao invés de códigos corretores de erros. Numa piconet com apenas um mestre e um escravo, a taxa máxima de 723.2 kb/s pode ser obtida se por exemplo o mestre utilizar apenas pacotes DH5 e o escravo apenas pacotes DH1, alternadamente. Em 1 segundo, seriam feitas em média 266.7 transmissões de pacotes DH5, o que totaliza 723290 bits de *Payload Body* transmitidos do mestre para o escravo (ver tabela 2-3) e 266.5 transmissões de pacotes DH1, o que totaliza 57564 bits do escravo para o mestre (ver tabela 2-3).

A escolha entre pacotes ACL do tipo DH ou DM é feita de acordo com a qualidade do canal. Se muitos erros estiverem ocorrendo, são escolhidos pacotes DM, senão o DH.

Os pacotes SCO são (não será descrito o pacote DV):

- HV1
- HV2
- HV3

Um pacote HV indica *High Quality Voice*. São pacotes de tamanho fixo, que indicam com os números 1, 2 e 3 se o *Payload* carrega 10, 20 ou 30 bytes de dados respectivamente. Os números 1, 2 e 3 também indicam se um pacote usa FEC 1/3, FEC 2/3 ou não usam FEC, respectivamente. Pacotes HV ocupam um único intervalo de 625 μs. Na realidade a transmissão de um pacote HV, DH1 ou DV1 ocupa no máximo 366 μs, pois são 366 bits ao total: 72 bits do Access Code, 54 bits do Header e no máximo 30 bytes de *Payload*. Um pacote HV não contém na carga útil os campo L_CH, FLOW, LENGHT ou CRC e nunca é retransmitido. Numa piconet com um mestre e um único escravo, se pacotes HV3 fossem transmitidos alternadamente, a taxa máxima simétrica seria de 192000 bits/s em ambas as direções.

Por fim, a tabela 2-3 indica os tipos de pacotes *Baseband* existentes para um enlace ACL e SCO, e as taxas de transmissão máxima obtidas numa conexão ponto a ponto. No caso do HV2 e HV3, assume-se que estes são transmitidos a cada 2.5 ms e 3.75 ms, respectivamente, para manter a taxa de 64 kb/s.

Tabela 2-3: Pacotes e taxas associadas, tiradas de [23]

Tipo	Máx. Carga Útil do usuário – *Payload Body* (bits)	FEC	CRC	Taxa simétrica	Taxa Assimétrica Direção	
					Mestre/Escravo	Escravo/Mestre
HV1	80	1/3	Não	64 kb/s	-	-
HV2	160	2/3	Não	64 kb/s	-	-
HV3	240	Não	Não	64 kb/s	-	-
DM1	136	2/3	Sim	108.8 kb/s	108.8 kb/s	108.8 kb/s
DM3	968	2/3	Sim	258.1 kb/s	387.2 kb/s	54.4 kb/s
DM5	1792	2/3	Sim	286.7 kb/s	477.8 kb/s	36.3 kb/s
DH1	216	Não	Sim	172.8 kb/s	172.8 kb/s	172.8 kb/s
DH3	1464	Não	Sim	390.4 kb/s	585.6 kb/s	86.4 kb/s
DH5	2712	Não	Sim	433.9 kb/s	723.2 kb/s	57.6 kb/s

2.2.4.8 – Canais Lógicos

Os canais lógicos são uma conexão lógica entre dois dispositivos, que servem para o transporte de dados entre o *Link Controller* de 2 dispositivos e camadas de protocolo superiores. Na versão 1.1 do *Bluetooth* são definidos 5 canais lógicos:

- *Link Control (LC) Channel*
- *Link Manager (LM) Channel*
- *User Asynchronous/Isochronous (UA/UI) Data Channel*
- *User Synchronous Data (US) Channel*

Os canais lógicos são mapeados no pacote *Baseband*.

O canal *Link Control* (LC) é mapeado no *Header* de um pacote e é utilizado para transportar informações que auxiliam no controle de fluxo (o bit FLOW), retransmissão de pacotes (o bit ARQN), entre outros, todos descritos na figura 2.22, na seção 2.2.4.7. O canal lógico LC está presente em praticamente todos os pacotes *Baseband*, com exceção do pacote ID, que não tem o campo *Header*.

Os canais lógicos UA/UI carregam informações provenientes da camada L2CAP. O canal lógico LM carrega informações da camada LMP. Ambos são mapeados no *Payload*, através do campo L_CH, onde L_CH igual a 10 ou 01 indica os canais UA/UI e L_CH = 11 indica o canal LM. A figura 2.23 ilustra o Payload na seção 2.2.4.7.

Dados do canal lógico US são transportados por uma conexão SCO, na carga útil de pacotes HV.

2.2.5 – *Link Manager Protocol* (LMP)

Na seção 2.2.2 foi apresentado *Link Manager* – LM (ver figura 2.5). O LM é responsável pela administração dos canais lógicos descritos na seção 2.2.4.8. O LM cria, modifica e termina os canais lógicos, assim como atualiza parâmetros relativos a conexão entre dois dispositivos. O *Link Manager Protocol* (LMP) é o protocolo de comunicação utilizado entre o *Link Manager* de dois dispositivos diferentes.

O LMP utiliza o canal lógico LM (ver seção 2.2.4.8), o que significa que as PDUs do LMP são transportadas no campo *Payload* de uma PDU *Baseband* ACL. Uma PDU LMP tem prioridade sobre um PDU L2CAP e deve ser transmitida primeira. A figura 2.24 ilustra o transporte do LMP em uma PDU *Baseband* ACL. O transporte lógico ACL garante a integridade dos dados, mas não garante a latência para a entrega dos mesmos. Isso porque erros sucessivos na recepção de uma PDU *Baseband* implicam em retransmissões sucessivas, o que impede a previsão do tempo para a entrega dos dados.

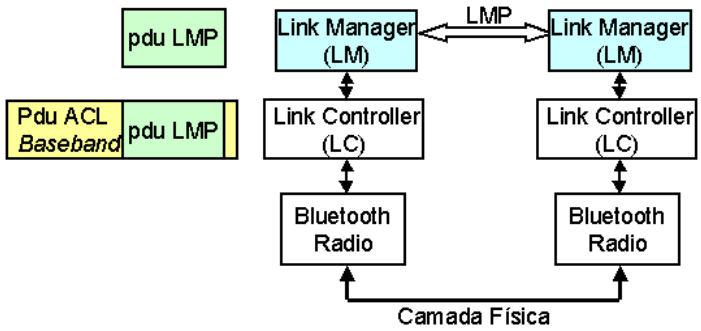


Figura 2.24 – Uma PDU LMP é utilizada na comunicação entre LM de dois dispositivos. A PDU LMP é encapsulada dentro de uma PDU Baseband, e transportada para o outro dispositivo.

O LMP funciona baseado na transação de um conjunto de mensagens encadeadas, que são trocadas entre o *Link Manager (LM)* de dois dispositivos, para alcançar um determinado objetivo. Um exemplo é dado a seguir: se em uma piconet o dispositivo mestre tiver que requisitar que um escravo ativo (que transmite e recebe dados) entre no modo *Park* (não transmite nem recebe dados, mas se mantém sincronizado a piconet), então para essa finalidade o LM do mestre vai enviar uma mensagem chamada de LMP_park_req, que indica uma requisição ao outro LM escravo para que ele entre no modo *Park*. O LM Escravo deve rejeitar ou aceitar a requisição, respondendo com LMP_accepted ou LMP_not_accepted, respectivamente. A figura 2.25 ilustra esse exemplo.

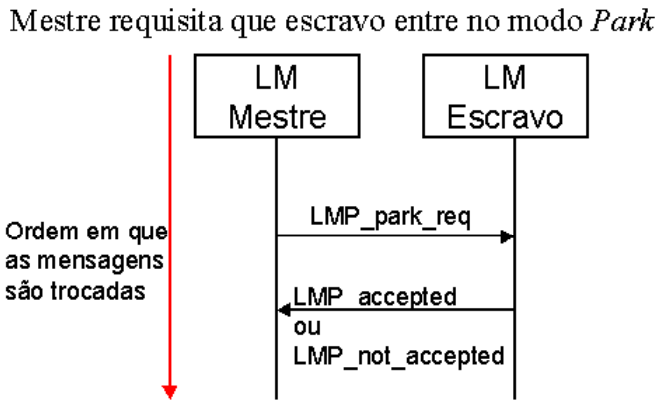


Figura 2.25 – Exemplo de troca de mensagens entre LM de um escravo e um mestre numa piconet.

Na especificação *Bluetooth*, a especificação do LMP descreve cada um dos conjuntos de mensagens a serem trocadas possíveis, organizadas de acordo com da finalidade de cada transação. São descritas também todas as mensagens (mapeadas nas PDUs do LMP) possíveis. A troca de mensagens entre LM deve ser local e não se propaga para as camadas de protocolo superiores.

O LMP também administra a parte de segurança entre dois dispositivos, mas esse assunto não será abordado neste trabalho.

2.2.6 – *Host Controller Interface* (HCI)

Um sistema *Bluetooth* é constituído de um *Bluetooth Host* e um módulo *Bluetooth* conectados. Um *Bluetooth Host* é definido como um sistema computacional, como um computador pessoal, *Laptop*, *Pda* ou telefone celular. No *Bluetooth Host* estão implementados protocolos de camada superior como o L2CAP, RFCOMM e SDP, além da interface HCI. Um módulo *Bluetooth* é composto pelo Rádio *Bluetooth*, *Link Controller* e *Link Manager*, cujos protocolos foram descritos nas seções 2.2.4 e 2.2.5. Na especificação *Bluetooth*, um módulo *Bluetooth* é referido como *Bluetooth Controller*, no entanto vamos utilizar apenas a nomenclatura de módulo *Bluetooth*.

A especificação do *Host Controller Interface* (HCI) descreve uma interface, cuja finalidade é permitir que um *Bluetooth Host* acesse as capacidades de um módulo *Bluetooth*. O HCI possui uma implementação em *software* no *Host*, chamada de HCI Driver, e uma implementação no módulo Bluetooth, chamada de HCI *Firmware*. Uma PDU HCI é projetada para 3 finalidades: transportar comandos, transportar eventos e transportar dados (do tipo SCO ou ACL).

Através do HCI, o *Host* envia PDUs HCI de comando para o módulo *Bluetooth*. A PDU de comando serve para acessar as capacidades do *Link Manager* e *Baseband*. Os comandos HCI são mapeados em códigos de 16 bits e são transportados em uma PDU HCI junto com os parâmetros de um comando. Por exemplo, se o *Host* enviar uma PDU HCI com o comando *Inquiry*, o *Link Manager* deve colocar o *Link Controller* (*Baseband*) no subestado *Inquiry* e iniciar os procedimentos descritos na seção 2.2.4.5. Nesse exemplo, o código do comando *Inquiry* é 0x0001 e é enviado do HCI Driver do *Host* para o HCI do módulo.

O módulo *Bluetooth* envia PDUs HCI para notificar o *Host.* Nesse caso uma PDU é chamada de PDU HCI de eventos, que serve para informar ao *Host* sobre parâmetros e dados. Por exemplo, se o módulo acabou de receber uma resposta de pesquisa *Inquiry*, significa que este recebeu o endereço BD_ADDR (ver seção 2.2.4.5) de um outro dispositivo *Bluetooth*. Para notificar o *Host*, o módulo envia uma PDU HCI de evento "*Inquiry Result Event*", que também transporta dados obtidos, como o BD_ADDR de outro dispositivo.

Por último, uma PDU HCI é utilizada para trocar dados entre um *Host* e um módulo conectados. No *Host*, os dados provenientes do L2CAP ou de uma aplicação de áudio são encapsuladas (e fragmentadas se necessário) dentro de uma PDU de dados HCI e enviadas ao módulo. No módulo, a carga útil de uma PDU *Baseband* é encapsulada numa PDU de dados HCI e enviada ao *Host*.

É importante notar que o que foi descrito até agora é o fluxo de dados lógico, entre o HCI do *Host* e o HCI do módulo *Bluetooth*. O fluxo real de dados deve ser feito através de uma interface de barramento físico, como por exemplo uma interface PC Card ou USB (*Universal Serial Bus*). Como nesse trabalho são utilizados dispositivos *Bluetooth* USB, vamos apresentar a camada de transporte HCI USB, que trata da transmissão de PDUs HCI pela interface USB. A figura 2.26 ilustra o fluxo lógico de dados entre o HCI Driver do Host e HCI *Firmware* do módulo, assim como o fluxo real de dados que são representados pelas setas vermelhas. A figura 2.26 será detalhada a seguir.

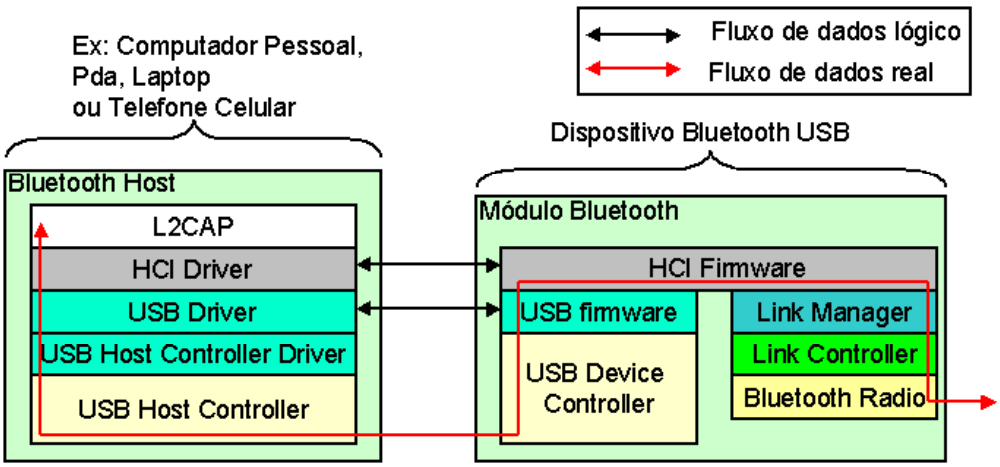


Figura 2.26 – HCI USB, comunicação entre as interfaces HCI do *Host* e do módulo através da interface USB.

O HCI USB especifica como as PDUs da interface HCI são transportados pelo USB. Na figura 2.26, o campo HCI *Driver* representa a interface HCI do *Host*. O USB *Driver* e o USB *Host Cotroller Driver* representam a implementação em *software* necessária para que o sistema operacional dê suporte ao USB, chamado de USB *System*. O USB *Host Controller* é o circuito que lê dados provenientes do USB *System* e os transfere pelo barramento.

No módulo Bluetooth também estão implementados o HCI do módulo (HCI *Firmware*), assim como a implementação em *software* (USB *firmware*) e *hardware* (USB Device Controller), ambos necessários em um dispositivo USB.

A especificação do USB, versão 1.1, apresenta 4 tipos de transferência de dados: *Control, Bulk, Interrupt* e *Isochronous*. A transferência do tipo *Control* não será descrita.

O HCI USB indica um tipo de transferência de dados do USB para cada um dos tipos de dados a serem transferidos: comandos, eventos, dados ACL e dados SCO. Serão listadas a seguir as descrições dos tipos de transferência de dados do USB, associando ao tipo de dados que o HCI USB transporta e mais duas informações sobre o USB:

- A transferência do tipo *Control* é utilizada para a transferência de dados não periódicos em modo de rajada e é sempre iniciada pelo *software* do *Host* em direção ao dispositivo USB. A transferência *Control* é utilizada para transmitir comandos, configurar ou enviar informação de status do Host para o dispositivo USB. Esse tipo de transferência é utilizado para transportar comandos do HCI *Driver* (do *Host*) para o HCI *Firmware* (para o módulo *Bluetooth*).
- A transferência de dados do tipo *Bulk* é utilizada para a transmissão de grandes quantidades de dados, sem compromisso com a latência, mas com garantias da integridade dos dados, utilizando ARQ (retransmissões). Não há reserva de recursos e utiliza o máximo da banda disponível. Esse tipo de transferência é utilizado para dados provenientes de conexões ACL;
- A transferência de dado *Interrupt* garante uma latência limitada e transmissões de poucos dados periodicamente (a cada 1 ms). Esse tipo de transferência é utilizado para o envio de pacotes HCI de eventos;
- A transferência *Isochronous* é utilizada para a transferência de dados de voz e vídeo, pois garante latência limitada através da reserva de banda de transmissão. Os dados nunca são retransmitidos e são em geral transmitidos a cada 1 ms. Esse tipo de transferência é utilizado por dados de conexões SCO;
- O barramento pode ser compartilhado por vários dispositivos. O USB *Host Controller* atua como um mestre e os dispositivos USB como escravos. Um escravo somente pode transmitir se o mestre lhe enviar um pacote USB especial, chamado de Token. Logo se observa que nesse esquema de revezamento, se houver muitos dispositivos por transmitir, a taxa de uma transferência *Bulk* deve ser reduzida e a de uma transferência *Isochronous*, se já estiver estabelecida, não será afetada;
- A taxa de erro esperada é de 10^{-13}, suficiente pequeno para não ser considerada um problema nas transferências *Isochronous* (que nunca retransmitem pacotes corrompidos)*;*

Para mais detalhes sobre o USB, ver [24], [25]. E sobre o HCI USB, ver [26];

2.2.7 – *Logical Link Control and Adoption* (L2CAP)

O L2CAP é um protocolo que provê suporte a multiplexação de protocolos de camadas superiores, realiza a segmentação e a remontagem de PDUs. A multiplexação permite que inúmeros protocolos usem os serviços da camada L2CAP, que transfere as PDUs dessas camadas superiores

para serem transportadas pelo protocolo *Baseband*, através de uma conexão ACL (ver seção 2.2.4.6). O L2CAP também prepara os dados provenientes de camadas superiores para que estejam compatíveis com o tamanho de PDUs de tamanho reduzido da *Baseband* ou do *HCI USB Transport Layer*, realizando a segmentação dos dados. E por fim realiza a montagem das PDUs de tamanho reduzido provenientes das camadas inferiores (*Baseband*) e as entrega ao protocolo superior correto. O L2CAP é um protocolo que pode ser orientado a conexão ou não, possuindo dois tipos diferentes de pacotes.

O L2CAP é projetado com as seguintes hipóteses em relação as camadas inferiores:

- o LMP deve prover apenas enlaces ACL (descritos na seção 2.2.4.6) para o L2CAP, e deve existir apenas um único enlace ACL entre dois módulos distintos;
- o protocolo *Baseband* deve garantir ao L2CAP a impressão de que a comunicação é *Full-Duplex*;
- o protocolo *Baseband* deve entregar pacotes de dados em ordem ao L2CAP;
- o protocolo *Baseband* deve utilizar os recursos de código de verificação de erro (CRC - *Cyclic Redundacy Check*), mecanismos de retransmissão de PDUs (ARQ – *Automatic Repeat reQuest*) e detecção de PDUs duplicados (ver seção 2.2.4.6). Ou seja, o Baseband garante ao L2CAP uma transmissão de dados confiáveis (a exceção à regra ocorre quando temporizadores indicam que a demora de entrega de PDUs ultrapassou um limiar de tempo de espera, logo alguns dados são descartados);

Pelas hipóteses, observa-se que o uso do L2CAP impõe que certos recursos do protocolo *Baseband* sejam utilizados. Por isso as PDUs L2CAP são bastante simples, deixando a cargo do protocolo *Baseband* a verificação de erros e ordenação dos pacotes, o que é razoável já que dados inválidos não são propagados desnecessariamente para as camadas superiores.

A figura 2.27 indica quais protocolos estão abaixo do L2CAP.

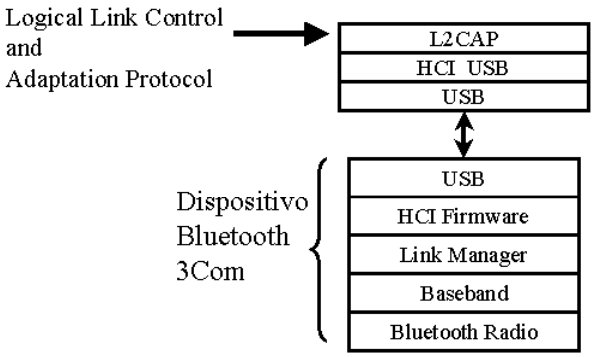


Figura 2.27 –
Neste trabalho o protocolo L2CAP se situa acima do protocolo HCI USB.

O L2CAP fornece serviços de multiplexação a protocolos de camadas superiores, permitindo assim que em cima da camada de L2CAP tenhamos vários protocolos diferentes sendo atendidos. No entanto, se vários protocolos estão sendo atendidos, é necessário que haja uma maneira de identificá-los e entregar ou receber corretamente os dados. Para essa finalidade de identificação, existe um

campo de 16 bits na PDU L2CAP, chamado de *Protocol/Service Multiplexor* (PSM), que é um número de identificação único para cada protocolo. O PSM possui uma faixa de 65536 valores numéricos, sendo que a faixa 0 – 4095 (em decimal) é reservada. Por exemplo, o valor PSM que corresponde ao protocolo RFCOMM é 0x0003 e o SDP é 0x0001. A figura 2.28 apresenta alguns dos protocolos que estão sobre a camada L2CAP e qual a sua PSM, e também indica que se um usuário criasse uma aplicação (identificada como "Minha Aplicação" e PSM = 0x2010), este poderia fazer uso de uma identificação PSM com um valor arbitrário não utilizado e passar a utilizar a comunicação *Bluetooth* a partir do L2CAP. O conceito de multiplexação do L2CAP é análogo ao utilizado pelos protocolos TCP ou UDP, que identifica aplicações diferentes com o conceito de "portas". A figura 2.28 ilustra o exemplo descrito.

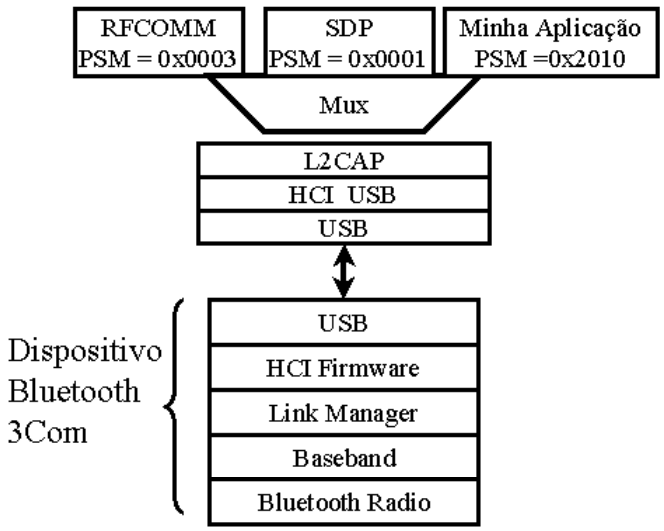


Figura 2.28 – Serviço de Multiplexação de Protocolos e Aplicações fornecido pelo L2CAP.

Além do PSM, o L2CAP também utiliza um campo de 16 bits que se chama *Channel ID*, que associa um canal lógico diferente para cada enlace ACL. Dessa forma, se dois dispositivos remotos *Bluetooth* estiverem tentando acessar o mesmo protocolo de camada superior, temos como distinguir um do outro. Por exemplo, se duas requisições de conexão chegassem ao L2CAP e ambas quisessem trocar dados na PSM = 0x0001, sabemos que ambas querem acessar o protocolo de camada superior SDP, mas cada uma dessas conexões receberá um *Channel ID* diferente para que seja possível responder as requisições para os dispositivos corretos.

O L2CAP é um protocolo que pode ser orientado a conexão ou não. Para isto ele possui dois pacotes de dados diferentes, como mostra a figura 2.29.

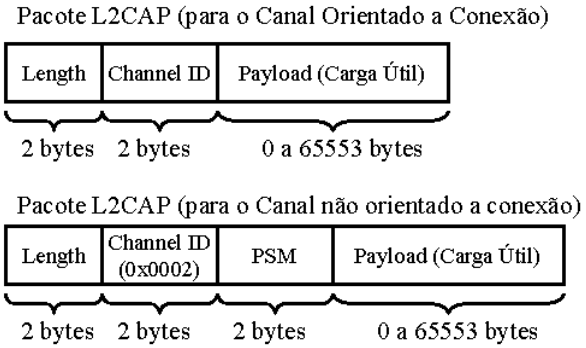


Figura 2.29 – Pacotes de dados L2CAP

Um pacote para um canal orientado a conexão possui no cabeçalho apenas campos que indiquem o comprimento (*Length)* da carga útil (que é variável), o número de identificação do canal (*Channel ID*) que é alocado dinamicamente e a carga útil (Payload). O campo PSM não existe e este valor é utilizado apenas no momento de requisição de conexão, utilizando um pacote de sinalização, que será explicado em seguida e está ilustrado pela figura 2.29.

Um pacote não orientado a conexão possui no cabeçalho um campo que indica o comprimento, um campo que indica um número de identificação fixo e de valor conhecido como 0x0002 (*Channel ID* = 0x0002) e um campo PSM. O valor de identificação de canal é igual a 0x0002 e é reservado, indicando que um pacote de dados L2CAP é não orientado a conexão. No entanto com um número de identificação fixo, não há como associar o número do canal com a conexão ACL proveniente e nem como associar esse número com um dos protocolos de camada superior, logo o PSM precisa ser enviado em cada pacote.

Um pacote de sinalização é utilizado pelo protocolo L2CAP para fazer requisições e enviar respostas, fazendo uso de um conjunto de mensagens fixas e conhecidas por ambas as entidades L2CAP. Na figura 2.30, os campos Comando 1 e Comando 2 sempre são enviados em pacotes com identificação de canal igual a 0x0001, que é o número reservado para pacotes de sinalização. Como exemplo temos comandos como "*Connection Request*", "*Connection Response*", "*Reject*", entre outros, para mais detalhes ver [6]. O comando "*Connection Request"* é enviado para requisitar a conexão com uma outra entidade L2CAP, e utiliza dois bytes para informar qual a PSM e mais dois bytes para informar o *Channel ID* de origem. O comando "*Connection Response*" é enviado para aceitar o pedido de conexão e informa ainda qual o *Channel ID* de destino foi alocado, o que torna desnecessário o uso do valor PSM depois de estabelecida a conexão.

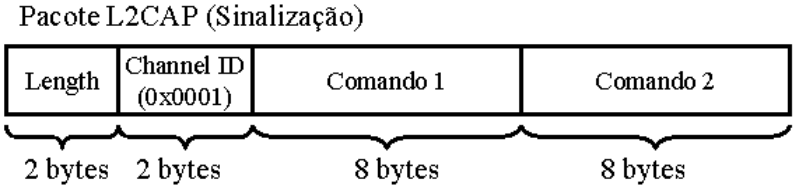


Figura 2.30 –
Pacote L2CAP de sinalização.

Outra importante função do L2CAP é a segmentação de PDUs e remontagem. As PDUs do *Baseband* (como vistos na seção 2.2.4.7, tabela 2-3), tem no máximo o tamanho de 339 bytes de carga útil, como na PDU DH5. Esses tamanhos máximos são projetados para melhorar o desempenho da transmissão aérea, mas podem trazer problemas de eficiência para os protocolos de camadas superiores, como o de transporte e de rede, que teriam um *overhead* muito maior caso o tamanho de uma PDU de camada superior fosse limitado ao tamanho utilizado pela carga útil das PDUs *Baseband*. Dessa forma, o L2CAP permite que as camadas superiores enviem PDUs de tamanho máximo de 65535 bytes, e posteriormente a PDU L2CAP vai ser segmentada em blocos menores para poder ser transmitido pelo *Baseband*. Para remontar a PDU a partir dos PDUs *Baseband*, são colocados dois bits no cabeçalho da carga útil, apresentados na seção 2.2.4.7, chamado de L_CH, que indicam com "10" que este é o primeiro bloco de uma PDU L2CAP e com "01" que o bloco atual é um segmento e faz parte de uma PDU L2CAP que está sendo remontada. A figura 2.31 ilustra o que foi explicado.

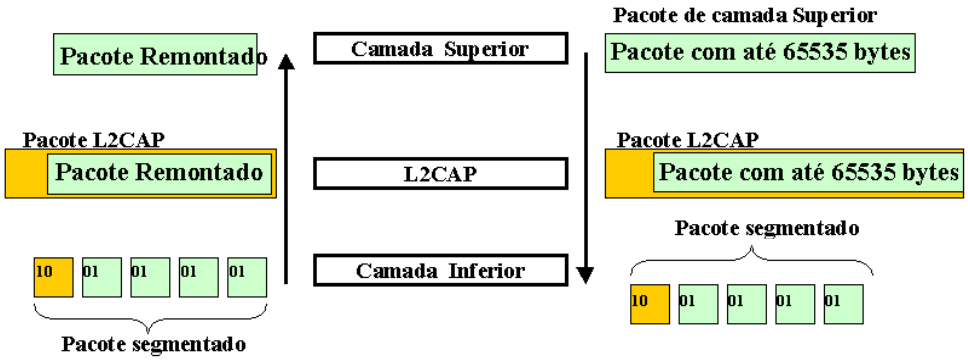


Figura 2.31 – Segmentação e Remontagem

2.3 – Módulos *Bluetooth* adquiridos e utilizados nesse trabalho

Os dois módulos *Bluetooth* USB adquiridos para esse trabalho são fabricados pela 3COM, baseados na especificação *Bluetooth* 1.1 O transceptor *Bluetooth* é de classe 3, tendo alcance de até 10 metros. Um módulo *Bluetooth* USB da 3COM é vendido em conjunto com um CD-ROM contendo o *software* necessário para a sua utilização em um sistema operacional Windows (98SE, Me, 2000 e XP). O *software* inclui um aplicativo chamado *Connection Manager*, com interface gráfica para apresentar e gerenciar pesquisas por novos dispositivos e conexões. A figura 2.32 apresenta uma foto dos módulos *Bluetooth* USB adquiridos.

Figura 2.32 - Módulos Bluetooth da 3COM adquiridos para esse projeto.

Embora os módulos sejam vendidos para operar em ambiente operacional Windows, os módulos da 3COM estão entre os dispositivos *Bluetooth* suportados pelo Bluez, a implementação em software livre que dá suporte ao *Bluetooth* no sistema operacional Linux. Para verificar quais módulos são suportados pelo Bluez, veja [27]. O estudo de protocolos *Host Bluetooth* (os protocolos L2CAP, AUDIO, a interface HCI, entre outros) baseado em *software* livre, permite melhores condições para o aprendizado do sistema *Bluetooth* como um todo, pois não somente existe uma vasta quantidade de informações sobre a implementação e o funcionamento do Bluez, como também sobre a implementação do USB *System Software* para o sistema operacional Linux.

Comercialmente existem vários tipos de kits de desenvolvimento para *Bluetooth* (inclui *hardware* e *software* para desenvolvimento). Também vendem-se módulos *Bluetooth* e implementações de protocolos *Host Bluetooth* separadamente. Na seção 8.4, do capítulo 8, serão indicadas possíveis soluções proprietárias que se adequem melhor a aplicações de *Bluetooth* em sistemas de controle e automação

2.4 – Usando o *Bluetooth* no sistema operacional Linux

2.4.1 – As 4 implementações do *Bluetooth Host* no Linux.

Atualmente, existem 4 implementações de *software* que permitem o uso do *Bluetooth* no sistema operacional Linux:

- Axis OpenBT Stack
- IBM BlueDrekar
- Nokia Affix Bluetooth Stack
- Qualcomm Bluez

Dessas quatro implementações, a implementação da Qualcomm Bluez foi escolhida por Linus Torvalds para se tornar a versão oficial do Linux *Bluetooth Stack* e passou a ser incluída no kernel do Linux a partir da versão 2.4.6, no ano de 2001.

Embora o Bluez não tenha uma documentação apropriada, informações para uso e desenvolvimento podem ser obtidas a partir das listas de discussão. A grande quantidade de emails trocados nas listas de discussão e os sistemas de busca de conteúdo de email possibilitam que se encontrem as informações necessárias para uma implementação, além de sugerir que a comunidade que utiliza e desenvolve o Bluez está bastante ativa.

Os principais desenvolvedores do Bluez, Maksim Krasnyanskiy e Marcel Holtmann, argumentam que uma maneira de se medir o desenvolvimento e qualidade das implementações do *Bluetooth* para o Linux, é observar o volume de emails trocados nas listas de discussão de desenvolvimento. Nesse sentido, as implementações Axis OpenBT Stack IBM BlueDrekar e Nokia Affix Bluetooth Stack parecem ter tido bastante atividade até o ano de 2003, quando a troca de mensagens diminui ou simplesmente cessa.

Por ser a versão oficial do Linux, o Bluez foi escolhido para ser utilizado nesse trabalho. A versão do kernel do Linux é 2.4.20 e a distribuição Linux é o Red Hat 9.0.

O Bluez implementa a interface HCI, o protocolo L2CAP, RFCOMM, AUDIO, entre outros. A figura 2.33 ilustra os arquivos (objeto) que podem ser carregados e ligados dinamicamente ao Kernel, conhecidos como módulos do Kernel, que pertencem ao Bluez. Os arquivos bluez.o/bluetooth.o, l2cap.o, audio.o e rfcomm.o representam respectivamente a implementação da interface HCI e dos protocolos L2CAP, AUDIO e RFCOMM.

A figura 2.33 também apresenta o arquivo hci_usb.o, implementação do HCI USB *Transport Layer* (ver seção 2.2.6), que funciona como o *client driver* do dispositivo *Bluetooth* USB e é um módulo que deve ser carregado automaticamente, tão logo um dispositivo *Bluetooth* USB seja conectado a uma porta USB. O USB Core (USB Driver) e USB *Host Controller* representam a implementação em *software* que em conjunto com o HCI USB, dá suporte ao uso do dispositivo USB *Bluetooth* no Linux.

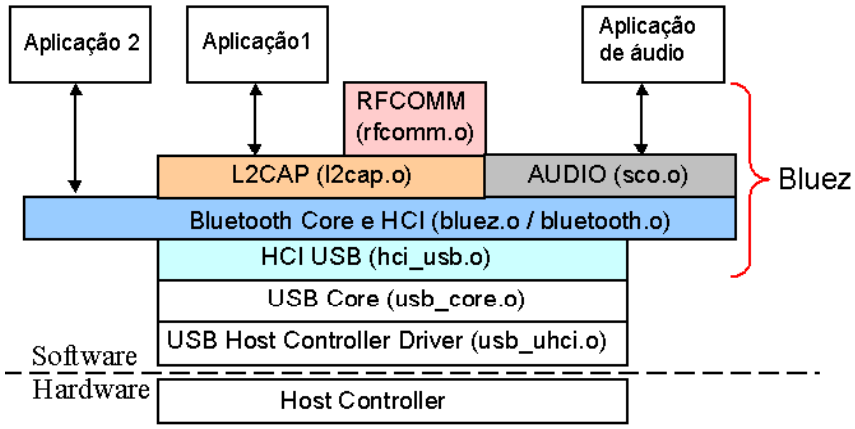


Figura 2.33 – Implementação do Bluez e USB no Linux.

No apêndice B é descrito como adquirir, configurar, instalar, e utilizar o Bluez no sistema operacional Linux, assim como criar programas do tipo cliente e servidor, para realizar a comunicação através do Bluetooth.

CAPÍTULO 3

Real Time Linux

O Real Time Linux (RTLinux) é um sistema operacional em tempo real, ou seja, é um sistema operacional que garante determinismo na execução de tarefas dentro de um prazo. Desde sua criação em 1997, o RTLinux já foi utilizado para aplicações em robótica, instrumentação, controle e simuladores. O RTLinux possui atualmente duas versões: uma versão gratuita chamada de RTLinux/Free e uma versão não gratuita chamada de RTLinux/Pro.

Esse sistema operacional em tempo real será necessário para a implementação de um simulador em tempo real.

A seguir serão descritos os conceitos de aplicações que operam em tempo real.

3.1 –Aplicações em Tempo Real

Aplicações em tempo real são aplicações em que o tempo desempenha um papel fundamental para a sua correta execução. Consideremos por exemplo aplicações de transmissão de dados de voz em redes de telefonia (ex. redes ATM – *Asynchronous Transfer Mode*) e a transmissão de grandes quantidades de dados via redes de computadores (ex. Internet ou rede local), como por exemplo o *download* via FTP (*File Transfer Protocol*) de arquivos. Pode-se observar que estas aplicações têm requerimentos diferentes. No primeiro caso o tempo para a entrega dos dados deve ser limitado e com pouco *jitter* (variação do tempo entre os pacotes), para garantir a cadência necessária em uma conversa entre dois usuários. Os dados de voz muito atrasados nem mesmo precisam ser reproduzidos e poderiam ser descartados. No segundo caso o objetivo é transferir grandes quantidades de dados assegurando a sua integridade, onde a transmissão utiliza a banda disponível na rede, sem que haja preocupação com o atraso de ida de um pacote ou o jitter.

Desses exemplos, a transmissão de voz em sistemas de telefonia fixa pode ser considerada uma aplicação de tempo real, pois requisita que a rede de telefonia honre as restrições de tempo para garantir o desempenho da aplicação, através por exemplo da reserva de banda, do estabelecimento de um circuito virtual, da prioridade sobre outras classes de tráfegos, dentre outras maneiras para se garantir a latência na entrega dos dados. Por outro lado, a transmissão do segundo exemplo não é uma aplicação em tempo real, pois o objetivo é realizar a tarefa da transferência de grandes volumes de dados, não importando por exemplo se a banda disponível varia com o tempo, resultando em um tempo variável de ida de um pacote ou em um maior *jitter* entre pacotes. Nesse caso o tempo não influencia no desempenho dessa aplicação, o que importa é sempre tentar obter da rede a maior banda de transmissão possível para transferir o máximo de dados e garantir uma entrega confiável dos pacotes.

3.1.1 – Distinções entre *Soft Real Time e Hard Real Time*

O tempo real pode ser classificado em duas classes distintas (utilizando-se dos termos em inglês) : *hard real time* e *soft real time*. A distinção entre esses tipos de tempo real é feita em relação aos requerimentos de tempo: se os prazos devem ser atendidos de forma determinística ou se é necessário apenas que o sistema seja rápido o suficiente, por exemplo, garantindo que na média as tarefas são executadas no prazo. Se o sistema pode ser apenas rápido o suficiente, no qual é suportado algumas falhas em se cumprir um prazo, o tempo real é definido como *soft real time*. Se o sistema tem que deterministicamente cumprir os prazos para executar uma determinada tarefa e a falha em se cumprir um prazo não for tolerável, então definimos a isso como *hard real time*.

Retornando ao exemplo da transmissão de voz pela rede de telefonia, o descarte de alguns pacotes de voz, devido atraso excessivo ou descarte em fila de roteadores, pode ser tolerado. Nesse caso, as restrições de tempo são apresentadas mais como uma orientação para um bom desempenho, do que para evitar conseqüências catastróficas. No *soft real time* o caso médio pode ser uma boa maneira para se avaliar o desempenho.

Em *hard real time*, o caso médio para o tempo de resposta a uma determinada tarefa não é de muito interesse e sim o pior caso. Um determinismo “verdadeiro” é necessário, não basta apenas o sistema ser rápido o suficiente. Um exemplo muito citado é a execução de uma seqüência de desligamento de um foguete (ver [30]). Se no pior caso não se garante que um ou mais passos da seqüência de desligamento são executados dentro do prazo, o foguete poderia explodir (ou literalmente ir para o espaço). O *hard real time* pode ainda ser classificado como periódico e aperiódico, sendo que no primeiro caso é importante executar a tarefa precisamente nos instantes agendados para a execução periódica, enquanto que no outro caso é importante executar a tarefa tão logo aconteça um evento.

Analisando as redes do primeiro exemplo, as redes ATM podem não apenas garantir as restrições de tempo da transmissão de voz em tempo real, como também poderiam transmitir os dados da aplicação do segundo exemplo. Já uma rede baseada em arquitetura IP (a Internet) pode ser utilizada para transferir arquivos através do TCP, mas não há garantias de que possa entregar os pacotes de voz em tempo real (não há garantias nem mesmo de que o pacote enviado vá ser entregue), o que se tem é o chamado serviço de entrega de melhor esforço. No entanto podemos supor que a entrega via IP é rápida o suficiente para permitir aplicações de voz em tempo real (do tipo *soft real time*), tolerando-se algumas perdas e atrasos, muito embora isso nem sempre seja uma suposição razoável. Essas duas arquiteturas de redes, Internet e ATM, exemplificam como uma mesma aplicação em *soft real time* pode ser executada em uma rede que garante os prazos de entrega e outra que não fornece garantia alguma. Um problema análogo será abordado nesse trabalho, quando analisarmos os sistema operacional Linux e a comunicação via *Bluetooth*, em relação às aplicações propostas na seção 1.2.4 do capítulo 1.

3.1.2 – Classificação das aplicações propostas

Conforme descrito na seção 1.1.1, um dispositivo inteligente (com capacidades computacionais) utilizado em um sistema de controle, poderia ser equipado com *Bluetooth* para ser acessado por um supervisor através de um PDA ou mesmo de um telefone celular. Nesse caso, possíveis aplicações poderiam ser carregar arquivos com registros de eventos e alterar parâmetros. Nesse caso, a transferência assegurando a integridade dos dados e sem restrições de tempo parece ser a melhor forma para se utilizar o *Bluetooth.*

Na seção 1.2.4 do capítulo 1, são propostas duas aplicações para a utilização da comunicação sem fio com a tecnologia *Bluetooth* em sistemas de controle, onde um dispositivo inteligente está conectado a um processo físico e é equipado com *Bluetooth*, por exemplo. A primeira aplicação é monitoramento baseado nas amostras provenientes de sensores e a segunda aplicação é controle distribuído, com a distribuição das funções (um nó sensor/atuador e outro controlador).

Na primeira aplicação, um computador pessoal recebe dados digitais periodicamente, provenientes de conversores A/D (analógico/digital) dos sinais de sensores (o que será emulado por um programa executado por uma tarefa do sistema operacional RTLinux) e através do *Bluetooth* transmite dados a outro dispositivo que os apresenta em um gráfico em função do tempo ou registra em arquivo. A periodicidade dos dados transmitidos não precisa ser feita na mesma freqüência de amostragem com que os dados são adquiridos, e a atualização do gráfico pode ocorrer tão rápido quanto for possível, não necessitando de nenhum determinismo no tempo de atualização. Podemos caracterizar essa aplicação como *soft real time*. A figura 3.1 ilustra o sistema dessa primeira aplicação.

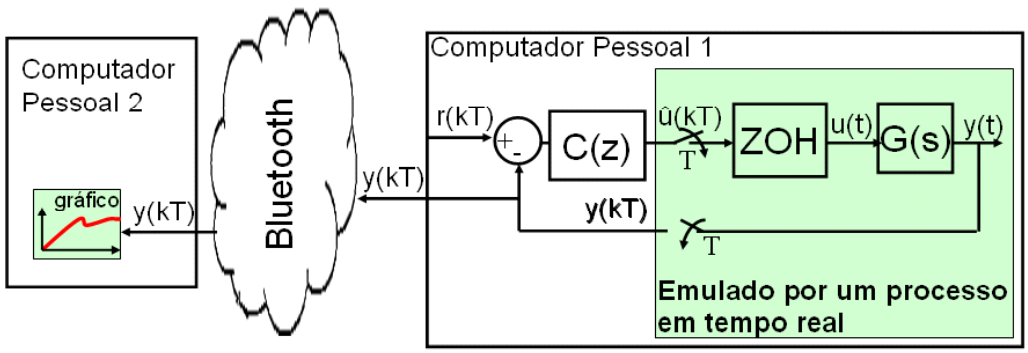


Figura 3.1 – Sistema desenvolvido para uma aplicação de *Bluetooth* em monitoramento.

Na segunda aplicação, um computador pessoal (vamos chamar de PC 1) recebe dados periodicamente dos sensores (o que será emulado por uma tarefa do sistema operacional RTLinux), e transfere os dados a outro computador pessoal (que vamos chamar de PC 2), que utiliza as amostras para calcular o sinal de controle, o que é transmitido de volta para o PC 1 que deve aplicar o sinal de controle em um conversor D/A (digital/analógico) para atuar no processo físico (emulado por uma tarefa do sistema RTLinux). Como foi apresentado na seção 1.2.2 e 1.2.3, a rede de comunicação introduz atraso no sistema de controle distribuído, o que reduz a margem de fase e o desempenho do sistema, podendo até instabilizá-lo. No entanto alguns sistemas de controle são robustos e podem tolerar atrasos de vários períodos de amostragem, o que não os caracteriza como sendo *hard real time*,

enquanto que outras classes de sistemas de controle o são. Exemplos de sistemas de controle ***hard real time*** e ***soft real time*** serão apresentados no capítulo 5. A figura 3.2 ilustra essa segunda aplicação.

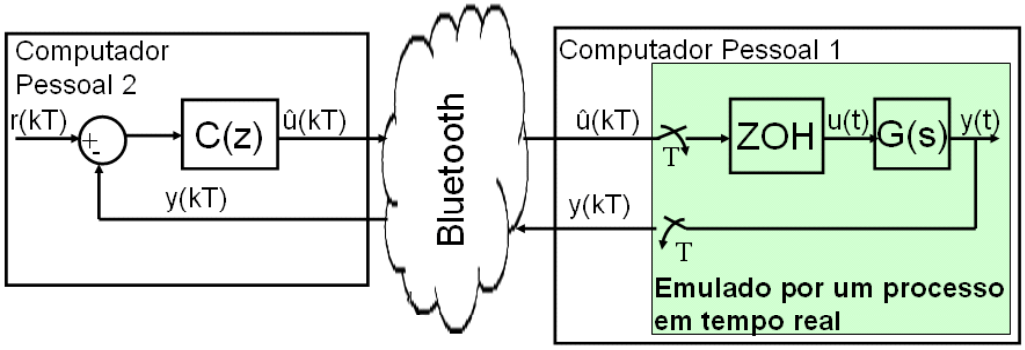


Figura 3.2 – Sistema desenvolvido para uma aplicação de *Bluetooth* em sistemas de controle distribuído.

3.2 –Linux e Real Time Linux/Free (RTLinux)

Nesse projeto, o sistema operacional Linux é utilizado para dar suporte à comunicação, via *Bluetooth,* entre dois computadores pessoais. No entanto, para garantir que amostras sejam geradas periodicamente por um emulador do sistema de aquisição de dados, vamos precisar de um sistema operacional que garanta a periodicidade dessa tarefa que é executada em tempo real. Para esse fim, vamos utilizar o *Real Time Linux*, um sistema operacional que garante a execução de tarefas periódicas, com precisão de algumas dezenas de microssegundos. Nas próximas seções serão apresentados o sistema operacional Linux e o Real Time Linux.

3.2.1 –Linux : Sistema Operacional de Propósito Geral

Nos capítulos anteriores, apenas a comunicação via *Bluetooth* foi apresentada por introduzir latência no sistema de controle distribuído.(ver seção 1.2.3 e 1.2.4). No entanto, neste projeto, o programa que calcula a lei de controle é executada como um processo no espaço de usuário em um sistema operacional Linux, kernel versão 2.4.20. O Linux (2.4.20) é um sistema operacional de propósito geral, com multiprogramação, monolítico, que é projetado para otimizar o desempenho para o caso médio, mas não garante determinismo de tempo para executar uma tarefa.

Os motivos para que o Linux (2.4.20) não seja considerado um sistema operacional de tempo real vão ser listados a seguir e foram apresentados em [30]. Mas primeiro vamos definir, em relação a um sistema operacional, o que são processos, tarefas e threads.

Um processo é um programa em execução, associado a um espaço de endereço e a uma lista dos locais de memória que pode ler e gravar, aos dados do programa e executa um ou mais threads. De acordo com [32], threads são uma abstração que representam os fluxo de execução dentro de um mesmo processo, que compartilham recursos do processo, como memória, dados e o mesmo espaço de endereço. Tarefas são uma maneira de fazer referência a processos a nível do Kernel.

A lista a seguir apresenta algumas características do sistema operacional Linux que têm validade até a sua versão 2.4 do Kernel. Muitas melhorias e alterações foram feitas para o Kernel 2.6 do Linux, a versão mais atual do sistema operacional do Linux, mas que não serão abordadas nesse trabalho (um dos motivos era a falta de suporte do RTLinux ao Linux 2.6). Algumas características do Linux (2.4.20), que impedem a previsibilidade do seu tempo de resposta a uma tarefa em tempo real, são:

- Para evitar que recursos compartilhados sejam acessados simultaneamente, o que causaria uma condição de corrida, um processo que está executando utiliza métodos de sincronização. A sincronização bloqueia o recurso compartilhado, dando exclusividade até que o processo termine de utilizá-lo. Entretanto no Linux, uma das formas de sincronização é obtida desativando-se interrupções no processador local, e um processo de menor prioridade pode monopolizar um recurso que um processo de maior prioridade pode precisar para executar;
- Quando a memória RAM torna-se um recurso escasso, o Linux permite que processos no espaço de usuário façam "swapping", onde se utiliza o disco rígido para criar uma extensão da memória RAM, mas com desempenho muito menor. Nesse caso o tempo de execução de um processo se torna imprevisível e pode aumentar de várias ordens de grandeza em relação a sua execução na memória RAM;
- No Linux, mesmo um processo com menor prioridade vai ganhar uma fatia de tempo para poder executar e concorrer com processos de maior prioridade;
- O agendador do Linux é preemptivo em relação aos processos de usuário (o processo é bloqueado para dar a vez a um processo de maior prioridade). No entanto os processos do Kernel contêm muitos trechos que não podem ser bloqueados, logo é dito que o Kernel não é preemptivo[1] (isso muda a partir da versão 2.6 do Kernel). Um processo de usuário de baixa prioridade que fizer uma chamada de sistema (o que muda o contexto para o espaço do Kernel) não poderá ser bloqueado por um processo de alta prioridade;
- O Linux reordena a ordem de execução das tarefas de entrada/saída para utilizar melhor o *Hardware* (o exemplo citado em [30] é atribuir mais tempo a um processo de menor prioridade que acessa o disco rígido, de forma a minimizar o movimento da cabeça do disco);

preemptivo[1]: Embora não haja uma tradução correta para a palavra inglesa "preemptive", muitos livros técnicos de sistemas operacionais utilizam a palavra preemptivo para a tradução deste termo. Um exemplo é a referência [46]. Nesse contexto, um sistema operacional preemptivo indica que uma tarefa de maior prioridade sempre pode interromper uma de menor prioridade que esteja em execução, e começar a ser executada pelo sistema operacional.

Foram apresentados alguns argumentos que justificam porquê o Linux não pode garantir que uma tarefa de alta prioridade seja executada imediatamente. No entanto, também existem problemas

(além dos citados) para se agendar tarefas periódicas, como a que vamos utilizar nesse trabalho para simular os dados obtidos de sensores.

Periodicamente o agendador do Linux é ativado, utilizando um dispositivo presente nos computadores pessoais modernos, que mede o tempo e pode ser programado para gerar interrupções. Esse dispositivo é chamado de APIC (*Advanced Programable Controller*). Tipicamente a freqüência de 100 Hz é utilizada, o que significa que a cada 10 ms o agendador é ativado. O período das interrupções (ex. 10 ms) é chamado de velocidade de tique. Processos que devem ser executados periodicamente são colocados para "dormir" em uma fila de espera. O Kernel monitora uma lista encadeada, onde cada nó da lista armazena quanto tempo falta para o processo ser acordado. A cada tique, o tempo de espera é atualizado (decrementando de um tique), até que o tempo de espera de um determinado processo expire. Quando isso ocorrer, o processo é posto numa fila de processos prontos para a execução, mas deve esperar até que o agendador o escolha dentre outros processos prontos para execução. Depois de executado, o processo é novamente posto na fila de espera, e o temporizar é reiniciado. Como esse temporizador é implementado em software (monitorado através de uma lista encadeada) e concorre com outros processos para ser executado, não há como garantir uma execução periódica de um processo com boa precisão de tempo, principalmente se houver muitos processos concorrendo para serem executados (a isso chamamos de carga do sistema).

Por fim, o Linux (2.4.20) fornece meios de se alterar a prioridade dos processos e a política do agendador (exemplo: mudar de uma política *Round Robin* para uma baseada apenas em prioridades), o que melhora o tempo de resposta a um evento ou a implementação de tarefas periódicas. No entanto, como descrito em [32], nenhuma dessas opções pode tornar o Linux (2.4.20) em um verdadeiro sistema operacional de tempo real.

Como deve estar claro, além da latência de comunicação inserida pelo *Bluetooth*, referindo-se a retransmissões de pacotes corrompidos e ao serviço de melhor esforço fornecido pelo transporte lógico ACL do protocolo *Baseband* e ao tipo de transferência *Bulk* do USB (ver capítulo 2, seções 2.2.4.6 e 2.2.6), deve-se contabilizar também a latência introduzida pelo sistema operacional Linux, que deve ser mais severa quando temos muitos processos executando (uma carga no sistema alta).

3.2.2 –Real Time Linux/Free (versão 3.2): Sistema Operacional de Tempo Real

Em 1978, foi publicado no *Bell System Technical Journal* a descrição do sistema operacional MERT, que tinha como objetivo de projeto poder executar tanto tarefas de propósito gerais quanto tarefas em tempo real. Para atender a esse objetivo, o MERT decompõe o problema em dois: ao invés de um único sistema operacional tentando dividir o tempo entre vários processos e garantindo o determinismo de tempo em processos de tempo real, são utilizados dois sistemas operacionais que trabalham em conjunto, um para as tarefas em tempo real e outro para as tarefas que não eram em tempo real.

Em 1997, Victor Yodaiken e Michael Barabanov lançam o Real Time Linux, que se baseia no princípio do MERT para resolver o problema de executar tanto as tarefas de tempo real quanto outras tarefas. Veja as referências em [30] e [34]. A idéia de Yodaiken é poder continuar a utilizar todos os

recursos do Linux, como navegadores WWW, editores de texto, interface gráfica, entre outros, ao mesmo tempo em que se poderia executar uma tarefa em tempo real, utilizando um outro sistema operacional (o RTLinux) na mesma máquina.

Para realizar isso, o Real Time Linux coloca uma camada de software entre o Hardware e o Sistema operacional Linux, de forma que todas as interrupções de hardware são atendidas pelo RTLinux e não podem mais ser tratadas diretamente pelo Linux. Além disso, o sistema operacional Linux é tratado como um processo de baixa prioridade do RTLinux, que somente pode ser executado se não houver mais nenhum processo de tempo real do RTLinux executando. Se, por exemplo, uma interrupção de *hardware* deve ser atendida por um *driver* de dispositivo do Linux, o RTLinux envia uma interrupção por software, que emula a interrupção de *hardware*, para o Linux assim que for possível, ou seja, quando não houver tarefas em tempo real executando. Todos os comandos do Linux referentes a desabilitar interrupções no processador local são trocados por comandos que não podem efetivamente desabilitar as interrupções de *hardware*. O Linux torna-se completamente preemptivo e pode ser bloqueado por um processo de tempo real do RTLinux, não importa o que o Linux esteja executando. A figura 3.3 ilustra o sistema operacional Linux como um processo do RTLinux e como o RTLinux se interpõe entre o Linux e o *hardware*.

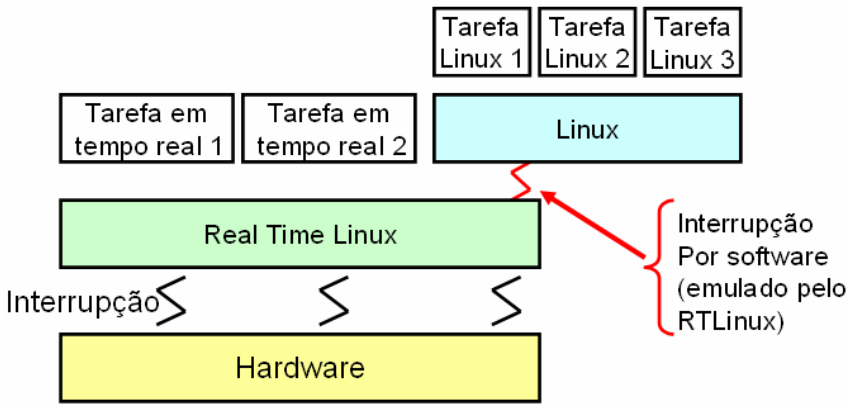


Figura 3.3 – Os Sistemas Operacionais: Real Time Linux e Linux.

Vamos chamar as tarefas em tempo real de tarefas RT. A seguir são listadas algumas razões para o RTLinux ser considerado um sistema operacional em tempo real.:

- O agendador do RTLinux utiliza uma política de prioridades, onde uma tarefa de maior prioridade é sempre executada primeira e pode bloquear uma tarefa de menor prioridade. Para cada tarefa é designada uma única prioridade (o Linux recebe a menor de todas), o que é representada por um número entre 0 e 10000;
- As tarefas em tempo real (tarefas RT) do RTLinux não podem alocar memória dinamicamente (durante a execução) e não fazem uso de memória virtual (não usa paginação). A memória alocada estaticamente fica reservada exclusivamente para uma tarefa RT (a tarefa RT se "tranca" na memória), a menos que sejam reservados recursos explicitamente para a comunicação entre processos;
- Para ganhar maior precisão em relação ao tempo, o RTLinux utiliza interrupções programadas em função do tempo, chamadas de modo *One Shot*, onde uma tarefa RT

(e isso inclui o agendador do RTLinux) pode ser programada para acordar em um tempo futuro qualquer com precisão de microssegundos, utilizando por exemplo o APIC (*Advanced Programable Controller*). Isso se contrapõe a estratégia do Linux de utilizar interrupções periódicas de pouca precisão (tiques), como por exemplo de 10 ms, e temporizadores baseados em software (monitorados em lista encadeadas), para programar eventos futuros;

- Não é permitido a uma tarefa RT esperar para que um processo do Linux libere algum recurso e isso implica em limitações na comunicação entre processos do Linux e do RTLinux. Por exemplo, não seria prudente que uma tarefa RT bloqueasse esperando por um processo do Linux terminar de processar algum dado. Como o Linux é completamente preemptivo, essa espera se prolongaria de forma imprevisível, num problema chamado de inversão de prioridade;
- O RTLinux não é completamente preemptivo, pois algumas tarefas podem utilizar métodos de sincronização, como desabilitar a interrupção, o que vai impedir um bloqueio. Entretanto o RTLinux é um Kernel pequeno, com poucas funcionalidades e as tarefas RT devem ser projetadas para serem pequenas o suficiente para não interferir muito no determinismo de tempo. Além disso, todas as tarefas que não precisam de tempo real ou são muito complexas, são deixadas a cargo do Linux.;

Como o RTLinux tem maior precisão para agendar e executar tarefas futuras, é possível criar uma tarefa RT periódica, com erros muito pequenos. O artigo em [30], do ano de 2001, afirma que uma tarefa periódica foi executada e que o pior caso em relação ao desvio do tempo correto de execução foi de 35 microssegundos, em um x86. No nosso projeto, o pior caso obtido foi de 19 microssegundos, o que será apresentado na seção 3.2.3, comparando-se com uma tarefa periódica no sistema operacional Linux (sem ter o RTLinux instalado).

Programas de usuários são carregados como tarefas em tempo real no espaço do Kernel, por isso um usuário que queira programar para o RTLinux tem que ter privilégio de superusuário. O RTLinux carrega esses programas como módulos (arquivos .o), através de comandos como modprobe ou insmod.

A comunicação entre os processos do Linux e do RTLinux é feita a partir das chamadas RT-Fifos, áreas de memória compartilhadas em que um processo pode ler e escrever. As RT-Fifos são tratadas pelo Linux como arquivos que podem ser abertos, fechados, escritos e lidos, e estão mapeadas no /dev/rtfX (no sistema de arquivos virtual), onde X pode assumir valores entre 0 e 150 (são 151 RT-Fifos diferentes). No entanto, apenas processos no espaço do Kernel (ou com privilégios de superusuário) podem abrir uma RT-Fifo. Um processo do RTLinux é responsável por criar e reservar memória para uma RT-Fifo, e se a tarefa RT for terminada, esta deve destruir a RT-Fifo. Uma tarefa RT nunca vai bloquear, esperando que algo seja escrito ou lido numa RT-Fifo. A figura 3.4 ilustra como duas RT-Fifos, rtf25 e rtf26, podem ser utilizadas para criar uma comunicação entre um processo RT e um processo do Linux com permissão de superusuário – root.

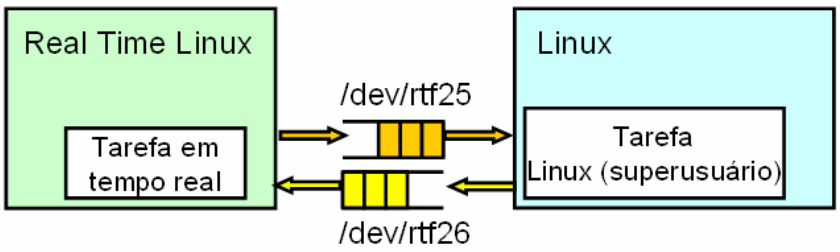


Figura 3.4 – A tarefa RT escreve em /dev/rtf25 e a tarefa do Linux lê dados de tamanho fixo (ex. uma estrutura de dado, com campos de cabeçalho e carga útil de tamanho fixo). O fluxo de dados no sentido inverso passa por /dev/rtf26.

Nesse trabalho, é através das RT-Fifos que a tarefa em tempo real, que emula os dados de sensores, vai passar os dados para um processo do Linux que deve ser responsável por encaminhar esses dados via o *Bluetooth* para um outro computador. Duas questões precisam ser abordadas nesse ponto: se não seria possível portar a comunicação *Bluetooth* para o RTLinux e se este não for o caso, se as tarefas do RTLinux não vão interferir no desempenho da comunicação *Bluetooth*.

Atualmente o RTLinux/Free é a versão gratuita e de código fonte aberto fornecida em [34], pela empresa FSMLABS. Essa empresa foi fundada por Victor Yodaiken, o criador do RTLinux.

3.2.3 – Comunicação sem fio *Bluetooth* e RTLinux

Nos artigo de Yodaiken em [30] e [33], o criador do RTLinux indica várias aplicações para o RTLinux, entre elas aplicações em sistemas de controle de robôs, simulação de avião e roteadores para telecomunicação. Nesse último caso, o suporte a protocolos de camada de rede (ex. IP) começava a ser implementado para o RTLinux da empresa FSMLABS (não para versão gratuita), para aplicar no processamento de pacotes em tempo real. Atualmente o RTLinux/Free possui suporte ao protocolo UDP em tempo real, mas o *driver* da placa de rede e a camada IP ainda são tratados pelo Linux e não pelo RTLinux (que implementa apenas a camada de Transporte de UDP).

Em [33], um dos principais quesitos para a implementação de um sistema operacional em tempo real é a simplicidade, que somente pode ser conseguido com um sistema operacional pequeno (mínimo), deixando a cargo do sistema operacional monolítico, como o Linux, os serviços mais complexos. Implementar vários serviços complexos em um sistema operacional em tempo real traz um custo/esforço muito grande para continuar a garantir a previsibilidade e a rapidez de resposta do sistema.

Uma implementação do *Bluetooth Host* no RTLinux, para o caso específico de uso de dispositivos USB, exigiria que se portassem todos os módulos referentes ao USB System (*driver* do controlador de *Host* e *driver* USB), camadas superiores do *Bluetooth* (l2cap.o e sco.o), interface HCI (bluez.o), driver do dispositivo USB (hci_usb.o). Mesmo se baseando no código dos módulos do Linux, a programação de *drivers* de dispositivos é mais complexa, pois diferente do espaço de usuários, o Kernel do Linux não tem acesso à biblioteca C padrão e usa implementações próprias, como por exemplo o printk() ao invés do conhecido printf(). Além disso, é necessário garantir que

nenhuma função utilizada faça chamadas de sistema (para o Linux) e sejam seguras de se utilizar no RTLinux/Free. Por exemplo, muitas das funções da biblioteca matemática (cujo cabeçalho é <math.h>) são seguras para serem chamadas pelo RTLinux. Por outro lado, não se pode utilizar nenhuma função em C que utilize o comando malloc(), como por exemplo sprintf().

A investigação de uma implementação do *Bluetooth Host* em um sistema operacional de tempo real está fora do escopo deste projeto, dado a complexidade dessa implementação.

Portanto, o que se pode fazer para minimizar o problema com a interferência de outros processos na comunicação *Bluetooth* é garantir que a carga do sistema não seja intensa suficiente para comprometer o funcionamento da comunicação *Bluetooth*. Durante transmissões via *Bluetooth*, com o Linux dividindo o tempo entre vários processos concorrentes, como *browsers* (ex. Mozilla), ferramentas de cálculo numérico (ex. Octave), entre outros, não houve mudanças significativas no atraso de comunicação *Bluetooth*. No entanto, quando utilizamos o Octave (programa que funciona como um Matlab, rodando scripts .m) para fazer operações com matrizes grandes o suficiente para que o swap seja utilizado (usa-se o disco rígido para "estender" a memória RAM), então o atraso de comunicação aumenta em algumas dezenas de ms (ex. 110 ms).

Por último, temos que verificar se a tarefa do RTLinux não irá interferir na execução do sistema operacional Linux. O RTLinux deve cumprir apenas o papel de fornecer periodicamente os dados, sem comprometer a comunicação. Isso pode ser obtido se o código da tarefa RT for pequeno e simples, e as chamadas periódicas tenham períodos suficientemente grandes para não interferir no funcionamento do Linux. Exemplo: foram testados os períodos de 0.5 ms e 1 ms, para executar tarefas como cálculos com números de ponto flutuante com precisão dupla.

3.2.4 – Alguns testes de desempenho

Nesta seção serão apresentados alguns resultados que exemplificam como os processos no espaço de usuário do Linux, que executam periodicamente, são afetados pela carga do sistema (outros processos concorrendo). Mostra-se também como minimizar esse problema no próprio sistema operacional Linux, trocando-se a política do agendador do Linux. Também será apresentado os resultados de uma tarefa do RTLinux que executa periodicamente.

No Linux (2.4.20), um teste para verificar se um processo consegue executar periodicamente pode ser feito com o seguinte trecho de código, que utiliza as funções nanosleep() e gettimeofday(). O código completo está no programa teste_periodo_other.c, que se encontra no Apêndice A.

```
/*trecho do código teste_periodo_other.c……*/
      while(i < 4000)
      {
            gettimeofday(&start,NULL); /*armazena o instante antes de
            dormir*/
            nanosleep (&ts, NULL); /* o processo é posto para dormir */
            gettimeofday(&stop,NULL);/*armazena o instante em que o
            processo acordou */
            timersub(&stop,&start,&diff); /*t_diff = t_stop - t_start */
```

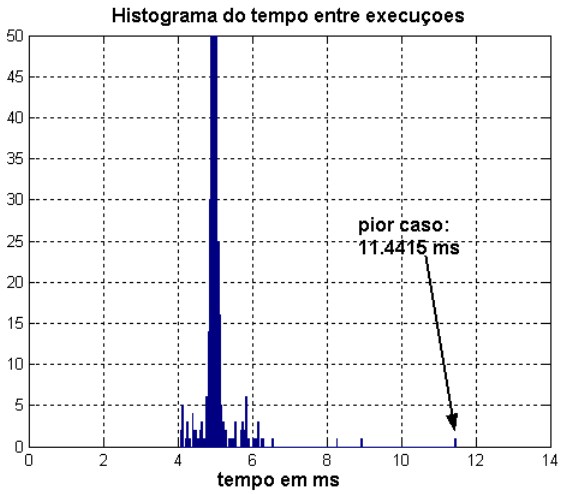
```
        funcao_armazena(diff);/*armazene o valor de diff, que indica o
        tempo entre execuções consecutivas */

        i++;
}
```

No Linux, 3 tipos de política do agendador podem ser usadas e são identificadas pelos termos SCHED_RR, SCHED_FIFO e SCHED_OTHER. A função sched_getscheduler() pode ser chamada por um processo para verificar qual das 3 políticas está sendo aplicada, e a função sched_setscheduler(, ,) pode ser utilizada para mudar a política do agendador. sched_getscheduler() e sched_setscheduler(, ,) são utilizadas no código de teste_periodo_other.c e teste_periodo_fifo.c, ambos disponíveis no apêndice A.

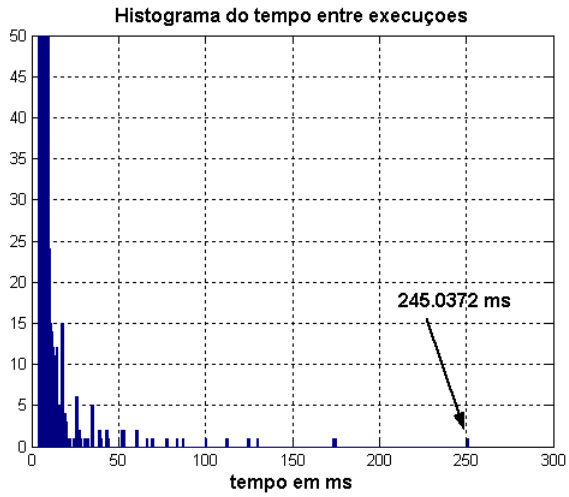
O termo SCHED_OTHER indica que uma política de compartilhamento de tempo (*Round Robin*) entre os processos está sendo aplicada, o que garante que um processo de baixa prioridade vai executar, mesmo que isso atrase um processo de alta prioridade. SCHED_FIFO e SCHED_RR representam políticas de prioridade, onde um processo de maior prioridade deve ser chamado imediatamente e é a opção que deve ser usada por processos que queiram operar em tempo real no Linux.

A função nanosleep() deve servir para um processo atrasar a sua execução, com a precisão de microssegundos. A primeira política a ser testada é a representada por SCHED_OTHER, e serão testados os períodos de 5 ms e 50 ms.

A figura 3.5 apresenta dois histogramas que indicam o tempo entre execuções consecutivas, para uma execução com período esperado de 5 ms. Na figura 3.5 (a), o sistema apresenta poucos processos concorrendo e o pior caso observado foi de 11.44 ms. Na figura 3.5 (b), vários processos são executados em paralelo, concorrendo e atrasando a execução da tarefa. A maneira de se obter essa maior carga, foi executar vários cálculos com matrizes muito grandes no programa Octave (programa que similar ao Matlab). A figura 3.5(b) tem amostras com atrasos de até 240 ms além do esperado (5 ms).



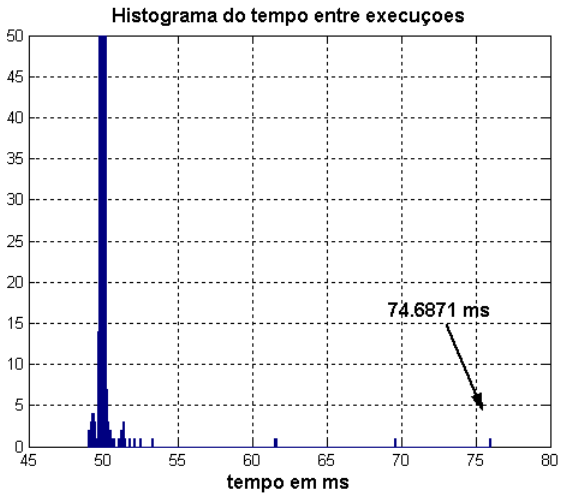
(a) pouca carga de sistema



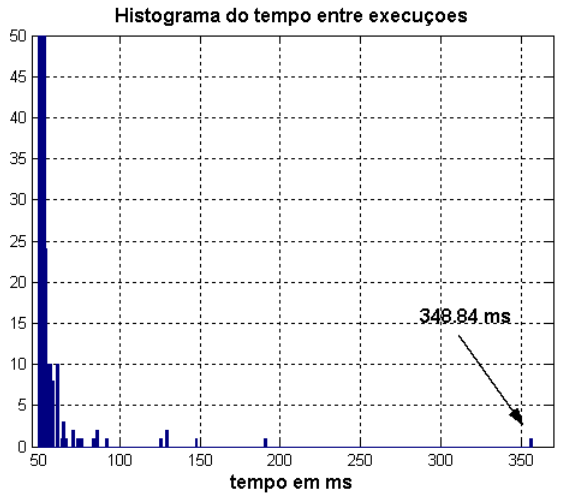
(b) muita carga de sistema

Figura 3.5 – Histogramas do tempo entre execuções periódicas no Linux, com a política de agendador *Round Robin* (SCHED_OTHER) e período de 5 ms.

O mesmo procedimento pode ser visto na figura 3.6, para a política Round Robin, mas com período de 50 ms.



(a) pouca carga de sistema



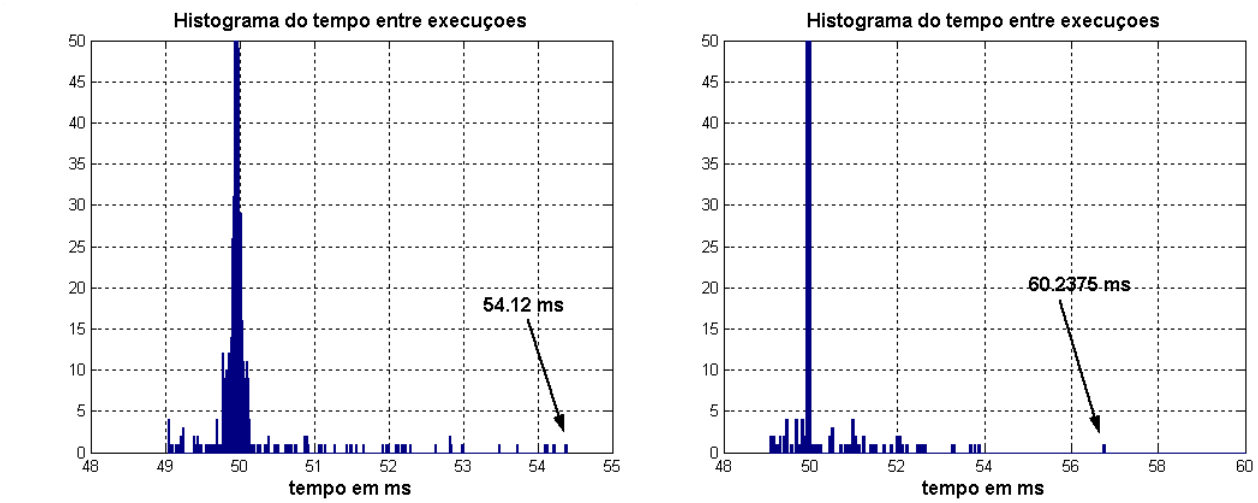
(b) muita carga de sistema

Figura 3.6 – Histogramas do tempo entre execuções periódicas no Linux, com a política de agendador *Round Robin* (SCHED_OTHER) e período de 50 ms.

Vamos fazer os mesmos procedimentos no Linux, mas trocando a política do agendador para SCHED_FIFO (política de prioridades). Processos com esse tipo de prioridade devem ser utilizados com cautela, pois podem executar indefinidamente, inclusive causando o congelamento do sistema operacional (que é monopolizado pelo processo). Serão testados 3 períodos: 50 ms, 5 ms e 1 ms. Para períodos de 50 ms e 5 ms, o nanosleep() faz com que o processo seja posto para dormir e posteriormente o agendador vai acordá-lo, o que implica em atrasos imprevisíveis. Para períodos abaixo de 2 ms (ex. 1 ms), o nanosleep()

permite que um processo com SCHED_FIFO faça uma espera ocupada, o que pode levar a uma maior precisão de tempo, mas o processo vai monopolizar o sistema.

A figura 3.7 apresenta os histogramas de tempos entre execuções consecutivas, com o período de 50 ms e política SCHED_FIFO. A amostra com maior desvio do valor esperado foi de 60.23 ms entre duas execuções consecutivas.

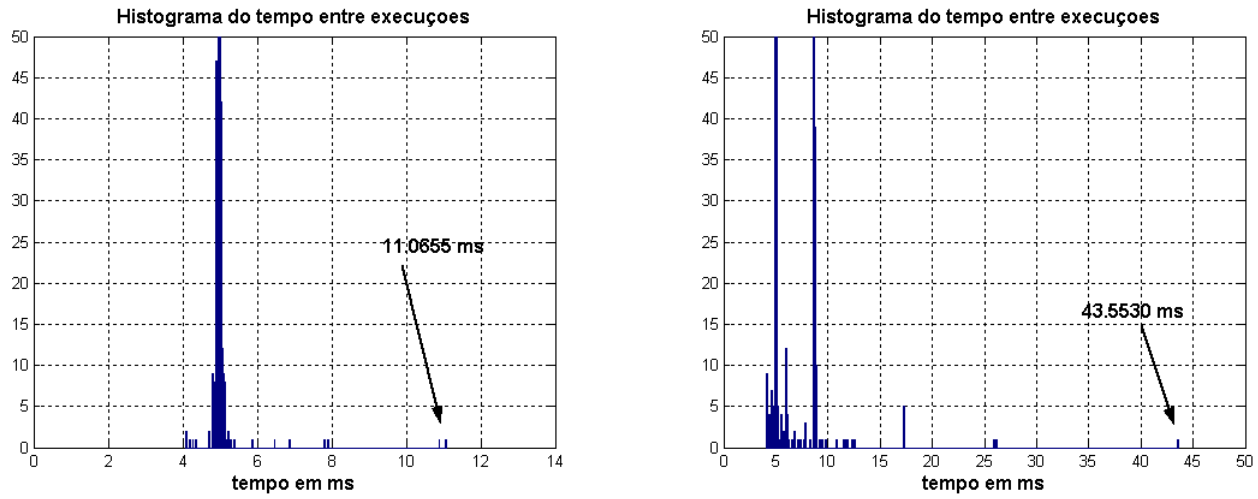


(a) pouca carga de sistema

(b) muita carga de sistema

Figura 3.7 – Histogramas do tempo entre execuções periódicas no Linux, com a política de agendador baseada em prioridades (SCHED_FIFO) e período de 50 ms

A figura 3.8 apresenta o mesmo caso da figura 3.7, mas com período de 5 ms.

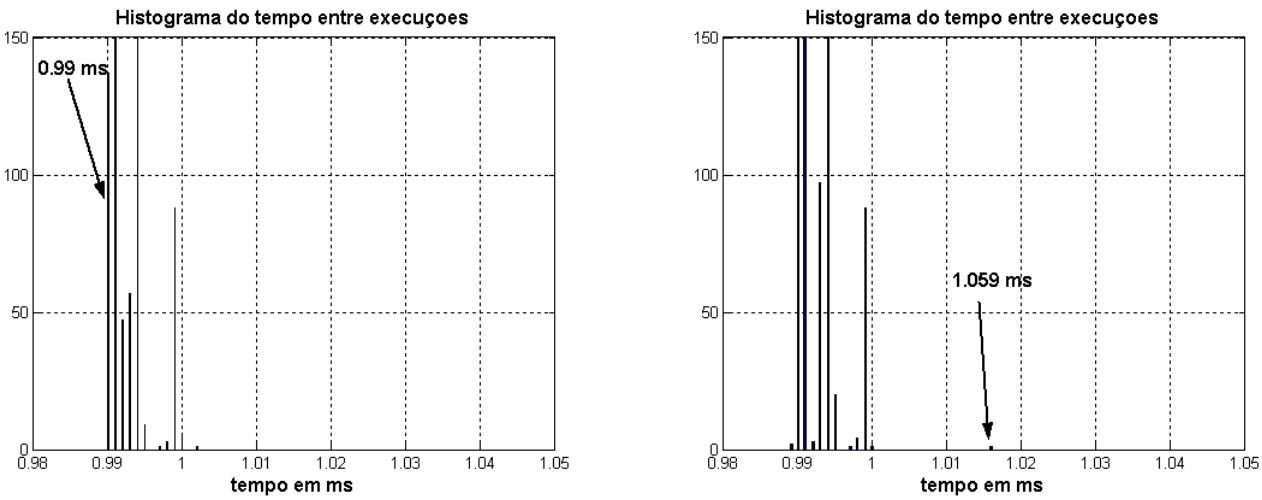


(a) pouca carga de sistema

(b) muita carga de sistema

Figura 3.8 – Histogramas do tempo entre execuções periódicas no Linux, com a política de agendador baseada em prioridades (SCHED_FIFO) e período de 5 ms

A figura 3.9 apresenta o mesmo caso da figura 3.8, mas com período de 1 ms. No entanto, nesse caso, é feita o nanosleep() faz uma espera ocupada, a precisão dos tempos entre execuções consecutivas é melhorada, ao custo de se ter um sistema operacional dedicado a uma única tarefa.

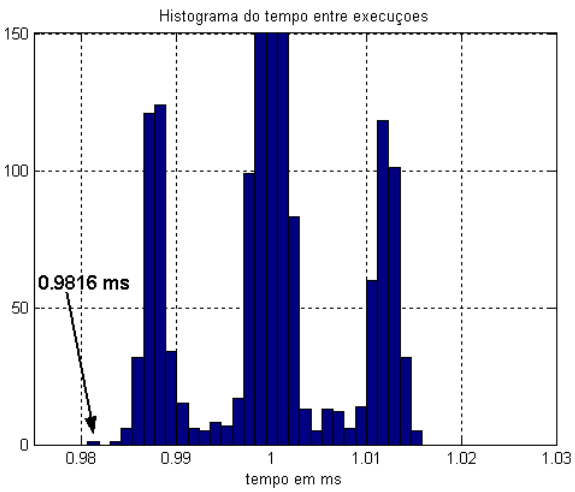


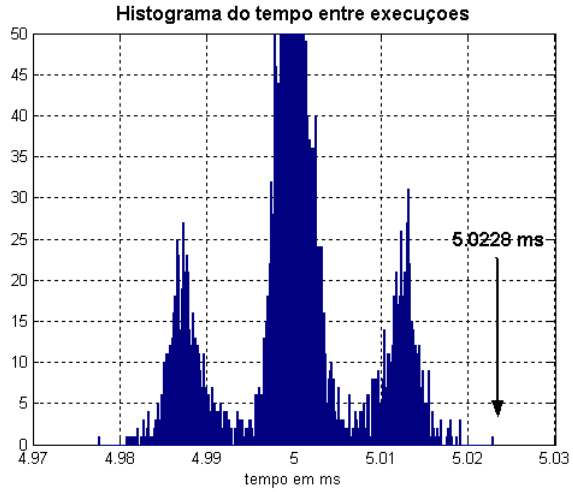
(a) pouca carga de sistema (b) muita carga de sistema

Figura 3.9– Histogramas do tempo entre execuções periódicas no Linux, com a política de agendador baseada em prioridades (SCHED_FIFO) e período de 1 ms

A seguir serão apresentados os histogramas para os tempos entre execução de tarefas periódicas no RTLinux, para períodos de 1 ms e 5 ms, com os sistema Linux realizando muitas tarefas, como por exemplo realizando cálculos com matrizes muitos grandes no Octave.

A figura 3.10 (a) apresenta o histograma para a execução de uma tarefa com período de 1 ms, comparando-se com a figura 3.9 (a) e (b), percebe-se que a tarefa do RTLinux não é afetada pela execução de processos do Linux, como também é muito mais preciso em executar a tarefas a cada 1 ms. Outra vantagem é que a tarefa do RTLinux não monopoliza o processamento e recursos do computador pessoal, como no caso do Linux com política SCHED_FIFO, permitindo que o Linux continue a executar suas tarefas. A figura 3.10 (b) apresenta o histograma correspondente a execução periódica de 5 ms. A figura 3.11 apresenta um histograma para o tempo entre execuções consecutivas para o período de 0.5 ms.





(a) Histograma para período de 1 ms (b) Histograma para período de 5 ms

Figura 3.10– Histogramas do tempo entre execuções periódicas no RTLinux, para períodos de 1 ms e 5 ms e com o Linux com muita carga no sistema.

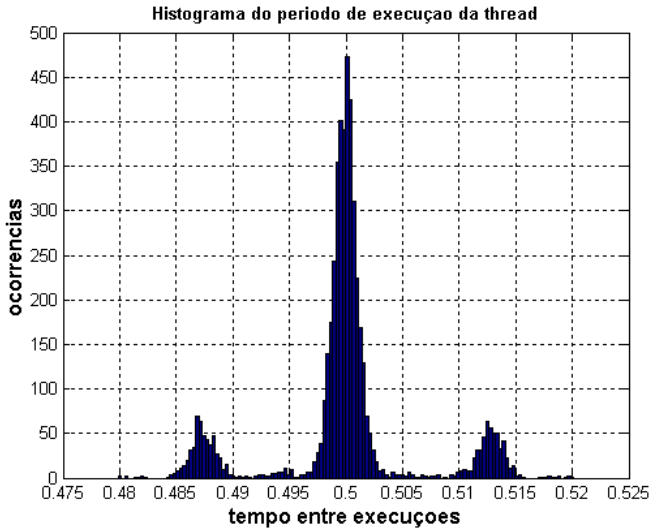


Figura 3.11 - Histograma de tempo entre duas execuções consecutivas de uma tarefa do RTLinux, com período de 500 μs.

Embora os histogramas das figura 3.5 a 3.11 ilustrem como a execução de uma tarefa em tempo real pode ser mais ou menos afetada pela carga do sistema operacional Linux, ou não ser alterada como no caso da tarefa do RTLinux, essa análise não é suficiente para garantir que um computador vá executar suas tarefas em tempo real e com tempos correspondentes ao processo que queremos emular, como o sistema físico e o sistema de aquisição de dados apresentados nas figuras 3.1 e 3.2.

Para verificar essa correspondência entre um simulador em tempo real, que será apresentado no capítulo 4, e o tempo correto em que os eventos devem ocorrer, vamos na figura 3.12 e 3.13 apresentar o resultado da seguinte fórmula:

Seja:

T o período de execução

N o número de amostras dos tempos entre execução de uma tarefa.

d(k) a diferença de tempo entre duas execuções consecutivas, para k = 0 ,1 ,2 , …, N

e(k) é a diferença entre o tempo esperado e o tempo acumulado para a k-ésima execução.

$$e(k) = \sum_{i=0}^{k} d(i) - T \times k \text{ , para } k = 0 ,1 ,2 , \ldots, N$$

e(k) representa o erro acumulado em relação ao tempo correto de execução. Se e(k) estiver negativo, significa que a execução de um processo está atrasada em relação ao tempo correto. Se e(k) se mantiver próximo de zero, a execução está próxima do tempo correto. Se e(k) estiver positivo, a execução está atrasada em relação ao tempo correto.

Na figura 3.12, são apresentados três curvas de e(k), para T = 5 ms, utilizando-se as amostras obtidas com os processos do Linux, com as políticas do agendador SCHED_FIFO e SCHED_OTHER, e com uma tarefa periódica do RTLinux. Na figura 3.12, o processo do Linux que ajusta o agendador para SCHED_FIFO (política de prioridade) está representado em verde. A curva verde da figura 3.12 demonstra que o processo sempre executa adiantado (antes do 5 ms) e que sofre alguns atrasos (como o que ocorre na amostra 800 e 849). A curva preta indica o erro do processo do Linux que executa sob a política de *Round Robin*, tendendo a ser executado com atraso, embora até a amostra 485 tenha executado adiantado. Isso demonstra que além do atraso, há pouca precisão nessa implementação da tarefa periódica para o Linux. A curva em vermelho, que representa o erro e(k) para a tarefa do RTLinux, tem o erro muito próximo de zero e não diverge conforme o tempo passa, como ocorreu nos casos anteriores (representados nas curvas verde e preta). O erro máximo registrado nesse caso foi de 16.9 μs. Pela figura 3.12, apenas a tarefa do RTLinux tem uma correspondência real entre o tempo de execução e o tempo desejado.

Na figura 3.13, é apresentado o mesmo tipo de curva da figura 3.12, mas apenas para a verificação do erro no tempo de execução de uma tarefa com período de 500 μs.

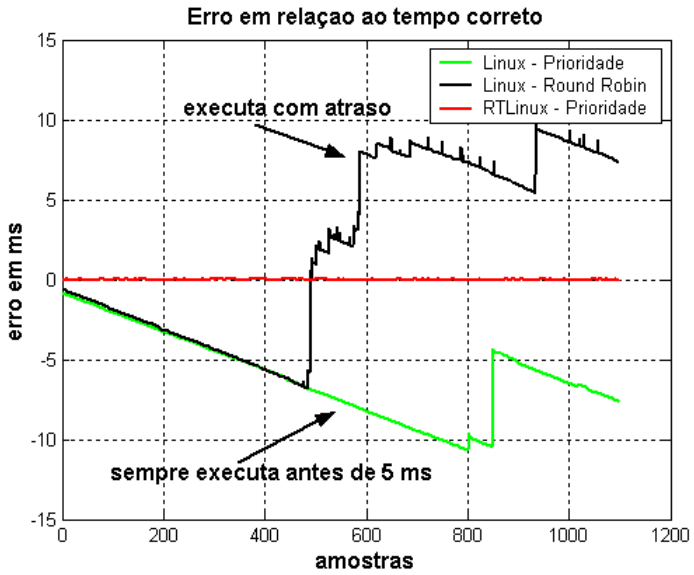


Figura 3.12 – Comparação do erro acumulado em relação ao tempo correto de execução, entre os processos do Linux e a tarefa periódica do RTLinux

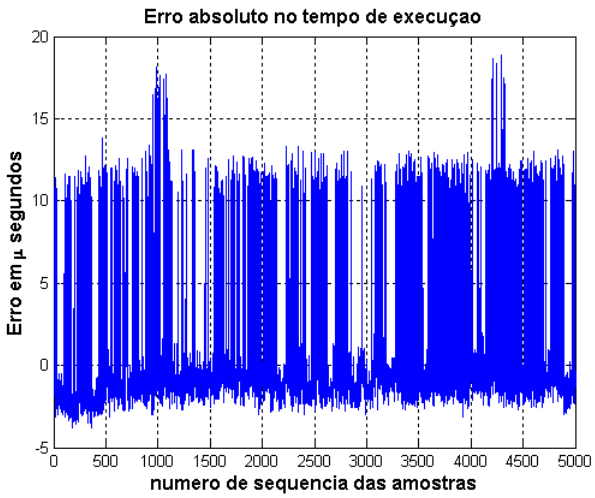


Figura 3.13 – Erro absoluto na execução periódica de uma tarefa do RTLinux, com período de 500 μs. Um valor negativo significa que a execução se adiantou, um valor positivo indica atraso na execução.

No apêndice C é descrito como adquirir, instalar, configurar e utilizar o RTLinux/Free. Apresentam-se também 3 exemplos de tarefas executadas em tempo real: um exemplo de como executar periodicamente; um exemplo que ensina a habilitar o acesso a números em ponto flutuante; e um exemplo que demonstre a utilização das filas RT-Fifo, para comunicação entre processos.

CAPÍTULO 4

Simulador em tempo real

Um simulador em tempo real é definido como uma simulação que tenta reproduzir o mesmo comportamento no tempo que o sistema real considerado. Esse tipo de simulação é uma etapa intermediária entre uma simulação puramente em *software* e a realização de testes em sistemas reais. Uma vantagem desse tipo de simulação, em relação a uma simulação puramente em *software*, é a possibilidade de avaliar o desempenho e restrições de equipamentos ou pessoal que interagem com o sistema virtual em tempo real. Uma vantagem da simulação em tempo real, em relação aos testes em sistemas reais, pode ser um custo reduzido e a possibilidade de se alterar configurações do sistema rapidamente e repetir experimentos sempre nas mesmas condições de operação.

Nesse trabalho será utilizado um simulador em tempo real para avaliar o desempenho de sistema de controle que utilizem da comunicação Bluetooth. Em um dos computadores pessoais são gerados periodicamente dados para serem transmitidos através do *Bluetooth*. Esses dados representam amostras adquiridas por sensores conectados a um sistema físico. No entanto, esse sistema físico e o sistema de aquisição de dados são todos virtuais, emulados por um simulador em tempo real, que vai ser executado como uma tarefa periódica no RTLinux/Free. De fato, esse sistema físico virtual é representado por um modelo matemático, o qual é descrito por equações diferenciais ordinárias.

A seção 4.1 descreve algumas motivações para a utilização de simulações em tempo real. A seção 4.2 descreve os métodos utilizados para a resolução numérica das equações diferenciais ordinárias dos modelos matemáticos. Alguns desses métodos numéricos serão utilizados para a implementação do simulador.

4.1 – Por que Simular em Tempo Real?

Como descrito no capítulo 1, esse tipo de simulação pode ser utilizado no meio acadêmico para a realização de experimentos em laboratórios virtuais, onde são simulados sistemas dinâmicos a partir de seus modelos matemáticos, auxiliando no ensino de teorias de controle como um complemento à realização de experimentos reais. Na indústria, a simulação em tempo real é utilizada para testar equipamentos e/ou o *software* nele presente, para verificar por falhas e limitações que não poderiam ser detectadas em simulações puramente em *software* ou que demandariam por testes muito demorados e caros em um sistema real.

Um exemplo de aplicação de simulação em tempo real é o chamado *Real Time Hardware-in-the-Loop* (HITL), ver introdução de [36] e [37]. Em [36] e [37], um protótipo de controlador digital é implementando em *hardware*, por exemplo numa DSP (*Digital Signal Processing*), e vai interagir com um sistema físico virtual, que é simulado em tempo real. A implementação do simulador em tempo real é feita em um computador pessoal (PC) com o sistema operacional RTAI Linux, um outro sistema operacional Linux em tempo real, que utiliza uma abordagem diferente do RTLinux/Free. Esse

tipo de simulação mais realística tem várias vantagens, pois a implementação em *hardware* do controlador vai expor suas limitações e condições de operação, que muitas vezes são ignoradas numa simulação puramente em *software*.

Assim, simulações em tempo real podem ser utilizadas para auxiliar no ensino acadêmico de teorias de controle e para auxiliar no testes e validação de protótipos de um controlador digital antes de aplicá-los em sistemas reais. Algumas características em comum desse tipo de abordagem são, por exemplo:

- o sistema simulado pode facilmente ser alterado, variando-se parâmetros, incluindo-se dinâmicas não modeladas ou simplificando o sistema simulado;
- os experimentos podem ser repetidos exatamente na mesma condição. Por exemplo, nesse trabalho a maior parte dos experimentos envolvendo o sistema de controle do pêndulo invertido são realizados com o pêndulo já numa posição próxima do ponto de equilíbrio instável, partindo do repouso. Em uma simulação, basta ajustar as condições iniciais das variáveis de estado via uma interface para o usuário. Em um sistema físico real, teríamos que de alguma maneira segurar o pêndulo nesta posição;
- em sistemas de controle, a simulação pode envolver ao mesmo tempo o projeto e o teste de um controlador, pois podemos variar parâmetros e verificar resultados rapidamente;
- uma simulação em tempo real traz maior realismo e permite avaliar resultados e custos em função de distintas escolhas para o período de amostragem;

4.2 – Métodos Numéricos de resolução de Equações Diferenciais Ordinárias (EDO)

No capítulo 5, serão apresentados modelos matemáticos que descrevem os sistemas dinâmicos, como por exemplo um modelo simplificado de um motor DC. No entanto, é necessário descrever quais métodos numéricos serão utilizados para resolver as equações diferenciais ordinárias desses modelos matemáticos, e porquê esses métodos numéricos são mais adequados para a simulação em tempo real.

Os modelos matemáticos de sistemas dinâmicos são descritos por equações diferenciais. Equações diferenciais ordinárias (EDO) são aquelas onde apenas derivadas em relação a apenas uma única variável independente estão envolvidas. A ordem de uma equação diferencial é fornecida pela maior ordem de uma derivada.

Para a resolução numérica, estamos interessados em aplicar métodos de integração a equações EDO de primeira ordem, ou a um conjunto destas. Por exemplo, a equação diferencial ordinária abaixo é de segunda ordem:

$$a\frac{d^2x(t)}{dt^2} + b\frac{dx(t)}{dt} + c\,x(t) = u(t) \quad \text{eq. de } 2^a \text{ ordem} \qquad (1)$$

Sempre podemos reduzir um problema de ordem superior, em um conjunto de EDO de primeira ordem. Exemplo, a equação (1) pode ser reduzida a conjunto de equações descritos em (2):

$$2\ \text{eq. de}\ 1^{a}\ \text{ordem}\quad \begin{cases} \dfrac{dx(t)}{dt} = x_2(t) \\ \dfrac{d\,x_2(t)}{dt} = \dfrac{1}{a}\left(u(t) - b x_2(t) - c\,x(t)\right) \end{cases} \qquad (2)$$

De maneira geral, vamos aplicar os métodos de resolução numérica a equações diferenciais de primeira ordem, dado uma condição inicial, para obter a resolução de um problema de valor inicial:

$$\frac{dx}{dt} = \dot{x} = f(t, x)\ ,\ x(t_0) = x_0 \qquad (3)$$

4.2.1 – Métodos Numéricos

Os métodos de resolução numérica se baseiam em obter uma aproximação para o problema do valor inicial, calculando a solução da EDO em tempos discretos t_1, t_2, t_3, …, t_n, partindo da condição inicial $x(t_0)$. Vamos chamar de h a diferença entre dois tempos discretos consecutivos:

$$h = t_{k+1} - t_k\ ,\ \text{para}\ k = 0,1,2,\ldots$$

O valor de h pode ser fixo ou variável. Dependendo do método utilizado, o passo variável permite que se obtenha resultados mais precisos reduzindo-se o tamanho de h, ou diminuindo gasto computacional quando o valor de h aumenta.

A partir da equação (4-4), abaixo descrita, os métodos de resolução numérica tentam obter o próximo valor x(t+h), utilizando o valor atual de x(t) e uma aproximação para a integral do lado esquerdo da equação:

$$x(t+h) = x(t) + \int_t^{t+h} f(t\ , x(\tau)\ )\,d\tau \qquad (4.4)$$

Vamos considerar que x(t) está correto na equação (4-4). Uma das maneiras mais simples de se aproximar o resultado da integral do lado esquerda da equação (4-4) à resposta exata, é assumir que f(t, x(t)) é constante durante o intervalo [t, t+h], o que assume que todas as derivadas de ordem superior f'(t, x(t)), f''(t, x(t)), f '''(t, x(t)), …, são todas iguais a zero. Isso pode não considerado como uma hipótese razoável e levar a erros significativos em relação a resposta exata. A esse erro relativo a resposta exata, cometido em um único passo, chamamos de erro de truncamento local, e ocorre quando fazemos uma aproximação a integral no intervalo [t, t+h] da função f(t, x(t)).

Vamos agora listar alguns métodos numéricos.

Um dos métodos mais simples é o método de Euler ou Runge Kutta de 1ª ordem, que calcula o próximo passo conforme a fórmula abaixo, baseado numa da aproximação da equação $x(t+h) = x(t) + \int_t^{t+h} f(\,t\,, x(\,t\,)\,)\,dt$. Aproximação pelo método de Euler assume que f(t, x(t)) se mantém constante durante o intervalo [t, t+h], logo obtemos a equação abaixo:

Método de Euler

$$x(t+h) = x(t) + h\ f(t,x) \qquad (4\text{-}5)$$

Um método mais refinado é o método de Heun, ou Runge Kutta de 2ª ordem, que utiliza a seguinte fórmula:

Método de Heun

$$x(t+h) = x(t) + \frac{h}{2}(f_1 + f_2)$$
$$f_1 = f(t, x(t)) \qquad (4\text{-}5)$$
$$f_2 = f(t+h,\ x(t) + h\, f_1)$$

O método clássico de Runge Kutta, ou Runge Kutta de 4ª ordem, é o método mais utilizado na prática, e é descrito pela fórmula abaixo:

Método Clássico de Runge Kutta (4ª ordem)

$$x(t+h) = x(t) + \frac{h}{6}(f_1 + 2f_2 + 2f_3 + f_4)$$
$$f_1 = f(t, x)$$
$$f_2 = f(t + \frac{h}{2}, x + \frac{h}{2} f_1) \qquad (4\text{-}6)$$
$$f_3 = f(t + \frac{h}{2}, x + \frac{h}{2} f_2)$$
$$f_4 = f(t+h, x + h\, f_3)$$

Esses 3 métodos são considerados métodos de passo simples, pois utilizam apenas um único valor, x(t), para calcular um próximo valor em x(t+h). A figura 4.1 apresenta quantas vezes é preciso calcular os valores de f(,) em função do método de passo simples escolhido.

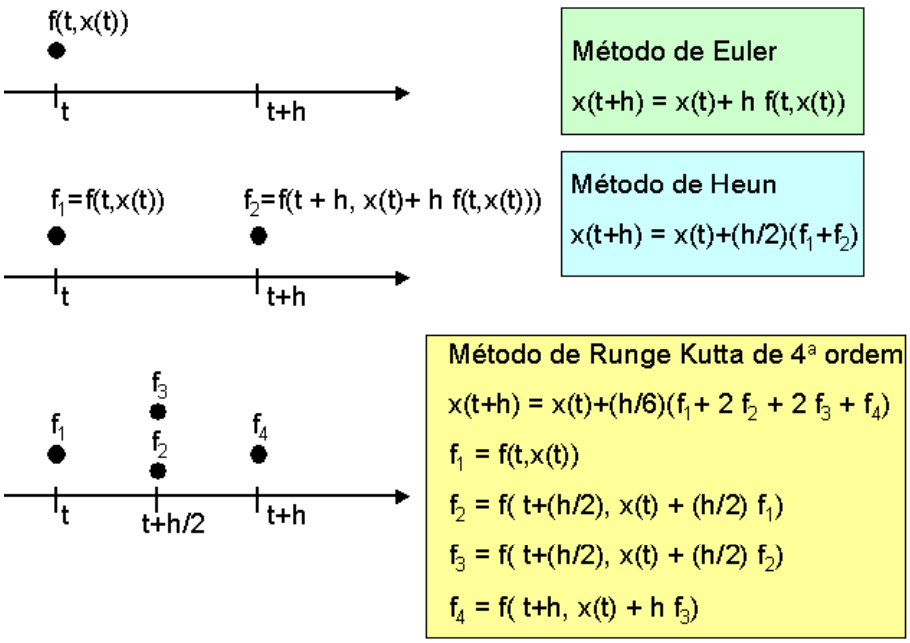


Figura 4.1- Nessa figura o pontos "•" indicam quantas vezes precisamos calcular f(,) em cada método de passo simples.

Os métodos de Euler, Heun e o método Clássico Runge Kutta são todos pertencentes a classe de métodos de Runge Kutta e são descritos como Runge Kutta de 1ª, 2ª e 4ª ordem, respectivamente. A ordem desses métodos implica em aproximações com diferentes erros de truncamento local, que é uma medida de quanto se erra do valor exato em um passo, em função de h, o passo de tempo, e uma constante que depende das derivadas superiores de f(t,x(t)). O erro de truncamento de Euler é da ordem de h^2, de Heun é da ordem de h^3 e o método Clássico de Runge Kutta é da ordem de h^5.

Além desses métodos de passo simples, também existem os métodos de passos múltiplos, que são caracterizados por utilizar a informação de mais de um ponto de f(t, x(t)), para aproximar a integral do lado esquerdo da eq. (4-4) por um polinômio. Vamos apresentar o método de Adams-Bashforth de quarta ordem, que é um método de passo múltiplos:

Método de Adams-Bashforth de 4ª ordem

$$
\begin{aligned}
&x(t+h) = x(t) + \frac{h}{24}\left(55\ f_t - 59\, f_{t-h} + 37\, f_{t-2h} - 9\, f_{t-3h}\right) \\
&f_t = f(t, x(t)), \\
&f_{t-h} = f(t-h, x(t-h)), \\
&f_{t-2h} = f(t-2h, x(t-2h)), \\
&f_{t-3h} = f(t-3h, x(t-3h))
\end{aligned}
\qquad (4\text{-}7)
$$

Observa-se que o método de Adams-Bashforth de 4ª ordem utiliza os valores de $f(t_i,x(t_i))$ de 3 passos anteriores. Durante a execução do algoritmo, basta armazenar os últimos 3 valores de f(,) e atualizar o valor atual para . Mas no instante inicial, apenas $x(t_0)$ e $f(t_0,x_0)$ vão estar disponíveis e o método precisará de uma etapa para ser inicializado. A solução para o método de Adams-Bashforth de 4ª ordem é utilizar um dos métodos de Runge Kutta para inicializar o algoritmo, calculando os 3 primeiros passos de tempo, e então trocar para o algoritmo de passo múltiplo. Por esse motivo, os métodos de simples também são chamados de métodos de partida. O erro de truncamento local do método de Adams-Bashforth é proporcional a h^5.

Todos esses métodos descritos utilizam os valores de x(t) já computados. Esses métodos são chamados de métodos explícitos. Métodos implícitos são aqueles em que o lado direito da equação para calcular x(t+h), também apresenta o termo x(t+h). Um exemplo é o método de Adams-Multon de segunda ordem:

Método de Adams-Multon de 2ª ordem

$$
\begin{aligned}
&x(t+h) = x(t) + \frac{h}{2}\left(f_t + 9\, f_{t+h}\right) \\
&f_t = f(t, x(t)), \\
&f_{t+h} = f(t+h, x(t+h))
\end{aligned}
\qquad (4\text{-}8)
$$

Um método implícito, como o método de Adams-Multon, pode ter que ser resolvidos de forma iterativa se a equação for não linear em x(t), o que torna imprevisível o tempo computacional necessário para obtermos o valor de x(t+h). Um método implícito conhecido é o método de Adams-

Basfoth-Multon, que executa um passo utilizando o método de Adams-Basforth e é chamado de preditor, e outro passo que executa o método de Adams-Multon e é conhecido como corretor.

Por último, o erro de truncamento local, que é ocasionado pela aproximação numérica de cada método, é da ordem de h^{k+1}, sendo k (em geral) equivalente a ordem do método escolhido. Isso significa que quanto menor h e maior a ordem do método, menor deve ser o erro de truncamento local. O erro de truncamento acumulado é, em geral, proporcional a h^k, para k novamente igual a ordem do método. Entretanto, erros também são introduzidos na simulação por conta da precisão finita dos sistemas computacionais, os quais são chamados de erros de arredondamento. Quanto menor for o passo de tempo h e maior a ordem do método, maiores serão os erros de arredondamento acumulados para se calcular um mesmo intervalo, de forma que esse erro pode ultrapassar o erro de truncamento. Isso resulta numa troca entre os erros de truncamente e arredondamento, em que temos que procurar um passo de tempo h que seja satisfatório para evitar instabilidade numérica, no qual os erros acumulados fazem a solução numérica divergir da solução exata.

Resumindo, os métodos numéricos podem ser caracterizados da seguinte forma:

- Métodos com passo de tempo h variável ou fixo;
- Métodos explícitos ou implícitos;
- Ordem k do método;
- Métodos de passo simples ou passos múltiplos;
- Erro de truncamento local ($O(h^{k+1})$) e global ($O(h^k)$);

A tabela 4.1 apresenta apenas os métodos numérico explícitos descritos, indicando a ordem de cada método e o erro de truncamento local associado.

Tabela 4.1 – Métodos numéricos explícitos descritos.

Método Numérico	Ordem	Ordem do erro de truncamento local
Euler	1	$O(h^2)$
Heun	2	$O(h^3)$
Runge Kutta Clássico	4	$O(h^5)$
Adams-Bashforth de 4ª ordem	4	$O(h^5)$

Na seção 4.2.2 essas características serão analisadas, e vamos apresentar alguns argumentos sobre quais características são consideradas desejáveis aplicação em simulações de tempo real e quais não são. Mais análises sobre os métodos numéricos podem ser encontrados em [38] e [39].

4.2.2 – Qual método numérico utilizar?

Vamos analisar que características podem comprometer a operação em tempo real de um simulador:

- Métodos implícitos aplicados na resolução de ODE, com termos não lineares em x(t), como por exemplo $\dot{x} = e^{x}$, são mais complexos de serem resolvidos e requerem técnicas iterativas para o cálculo de f(t+h, x(t+h)), o que além de aumentar o custo computacional, torna imprevisível o tempo de execução;
- Métodos com passo de tempo h variável possibilitam uma resolução numérica de melhor precisão e também reduzem o custo computacional médio se comparados a métodos de passo fixo. Um exemplo é o método de Runge Kutta Fehlberg 4 e 5, onde dois algoritmos de Runge Kutta de ordem 4 e 5 são utilizados para estimar o erro de truncamento e ajustar o tamanho do passo h em função disso. Esse tipo de método é iterativo em relação ao ajuste do tamanho de h e consome mais recursos computacionais para poder executar dois métodos diferentes. Embora, os métodos de passos de tempo variáveis tragam vantagens em simulações que não são em tempo real, não são utilizadas para simulações em tempo real, por tornarem imprevisíveis o tempo computacional gasto em cada passo e por problemas do *hardware* garantir o tempo real quando uma função não é periódica ;
- A ordem do método indica um melhor desempenho em relação aos erros de truncamento, mas é necessário observar 3 itens. (1) - métodos como de Runge Kutta de 4ª ordem e de Adams Bashforth de 4ª ordem tem a mesma ordem de erro de truncamento local $O(h^5)$, mas o método de Adams Bashforth utiliza variáveis que já foram calculadas em passos anteriores, enquanto no método de Runge Kutta é necessário calcular, a cada passo, vários valores de f(,) em passos futuros, o que é uma desvantagem em relação ao custo computacional do método de Adams Bashforth. (2) – Quanto maior a ordem, menor deve ser o erro de truncamento, mas pode-se aumentar o erro de arredondamento e (3), maior também deve ser a necessidade de velocidade de processamento ou espaço em memória;
- Como descrito no item anterior, métodos de passo simples, como os de Runge Kutta de ordem superior, podem ter que calcular vários valores de f(,) que não estão disponíveis, aumentando o tempo de execução da rotina. Métodos de passos múltiplos e explícitos, conseguem um erro de truncamento equivalente, utilizando dados já disponíveis;
- Deve-se minimizar o erro de truncamento e o erro de arredondamento, além de verificar um limite uso dos recursos computacionais. Para isso, deve-se escolher métodos de ordem superior (ex. 4ª ordem), um passo de tempo h suficientemente pequeno (e não o menor possível) e se possível utilizar ponto flutuante com precisão dupla (ex. do tipo *double*);
- Um dos principais desafios na resolução numérica de EDO são os chamados problemas mal condicionadas ou inflexíveis (do termo em inglês *stiff*), onde o sistema apresenta constantes de tempo que diferem de algumas ordens de grandeza umas das outras. Nesses casos, a seleção do passo de tempo h é crítica, e deve acomodar tanto as componentes rápidas quanto as lentas e os métodos numéricos implícitos são os mais usados nesses problemas. No entanto, como foi descrito, métodos implícitos não são usualmente utilizados em simulações em tempo real quando há termos não lineares;

No RTLinux/Free, o suporte a tarefas periódicas é satisfatório, pois como apresentado na seção 3.2.4, a execução periódica não diverge do tempo correto de execução. No entanto, tarefas aperiódicas não tem suporte adequado, o que dificulta a implementação de simulações em tempo real que utilizem passo no tempo variável. Por exemplo, se fôssemos utilizar um método de passo de tempo variável, teríamos que calcular o novo passo h_{novo}, e então agendar a rotina para o tempo atual mais $h_{novo} - T_{execução} - T_{atraso}$, onde $T_{execução}$ é o tempo em que a tarefa executou e T_{atraso} é o atraso relativo ao tempo correto para o início da execução. No RTLinux/Free isso poderia ser feito com o comando clock_nanosleep(), que sempre se atrasa de algumas dezenas de microssegundos, mas teria que ser monitorado pela rotina de usuário para não divergir do tempo em que realmente deve executar.

Os módulos do RTLinux/Free, como apresentados nos exemplos da seção 3.4.1, podem utilizar números em ponto flutuante, inclusive com precisão dupla. Esse recurso será utilizado nas simulações desse trabalho, para reduzir os erros de arredondamento.

Em relação aos métodos implícitos, o primeiro exemplo a ser apresentado no capítulo 5, um modelo simplificado de um motor DC, é linear e poderia utilizar métodos numéricos implícitos. No entanto os outros modelos tem muitos termos não-lineares em relação as variáveis dependentes, o que não os torna atrativos para essa implementação de um simulador.

O método numérico para resolução de EDO utilizado nesse trabalho é o método de Clássico de Runge Kutta, com passo fixo de tempo.

No simulador implementado no RTLinux/Free foram utilizadas os seguintes recursos:

- números em ponto flutuante de precisão dupla;
- suporte a periodicidade de 1 ms ou 0.5 ms entre execuções;
- comunicação com o sistema operacional Linux através de RT-Fifos (memória compartilhada)

Capítulo 5

Sistemas de Controle : Estudos de Caso

Como descrito no capítulo 1, vamos utilizar a comunicação *Bluetooth*, versão 1.1, para duas aplicações distintas: monitoramento e controle distribuído. Os dados provenientes de sensores conectados a um sistema físico são transportados pela comunicação *Bluetooth*, para um outro computador pessoal. No entanto, o sistema físico e o sistema de aquisição de dados são todos virtuais, sendo descritos por modelos matemáticos. As equações diferenciais que descrevem um determinado sistema dinâmico são resolvidas através de métodos numéricos descritos no capítulo 4. A simulação em tempo real é executada através de uma tarefa periódica, no sistema operacional RTLinux/Free (versão 3.2), um sistema operacional descrito no capítulo 3.

Nesse capítulo, vamos apresentar 2 estudos de caso sobre sistemas de controle que utilizam um sistema de comunicação *Bluetooth*. No entanto, como não serão realizadas comparações com sistema reais para validar os modelos matemáticos que serão propostos, os dados obtidos nesses estudos de caso não representam com fidelidade qualquer sistema real, e servem apenas para auxiliar no entendimento de alguns aspectos teóricos, de forma qualitativa.

Em cada caso, vamos descrever os modelos matemáticos de sistemas físicos que vão ser simulados e apresentar os resultados de simulações numéricas. Vamos mostrar os resultados de simulação utilizando os métodos numéricos de passo de tempo fixo, e compará-los com métodos numéricos de passo variável e de ordem superior, a fim de verificar a estabilidade numérica dos métodos utilizados em determinadas simulações.

Os parâmetros de tais modelos são ajustados de acordo com as descrições de sistemas em [40] e [41]. Em [40] temos um manual que fornece os detalhes técnicos para realizar o controle de um pêndulo invertido. Em [41], temos um exemplo didático sobre a aplicação de teorias de controle em sistema lineares invariantes no tempo, a partir de um modelo de um sistema de 2ª ordem.

O primeiro estudo de caso, na seção 5.1, apresenta um sistema de controle linear, em que estamos interessados em rastrear um sinal de referência do tipo degrau e rejeitar perturbações externas que interfiram nesse controle. Esse primeiro estudo tem os seguintes propósitos:

- Exemplificar uma simulação em tempo real de um modelo linear no RTLinux/Free;
- Exemplificar abordagens de controle digital linear;
- Apresentar um sistema de controle distribuído que é robusto em relação a um atraso de comunicação desconhecido, que é assumido como um atraso de valor limitado e estático;

O segundo estudo de caso, nas seção 5.2, apresenta um modelo não-linear de um pêndulo invertido rotatório, em que o objetivo de controle é estabilizar o pêndulo no ponto de equilíbrio instável. Os objetivos desse exemplo são:

- Exemplificar uma simulação em tempo real de um modelo não-linear no RTLinux/Free;

- Apresentar um sistema de controle distribuído em que o atraso de comunicação na malha de controle pode dificultar a prática de alguns algoritmos de controle não lineares;

Na seção 5.3 vamos fazer uma pequena introdução ao controle digital de sistemas dinâmicos, e mostrar como os algoritmos de controle foram discretizados nesse trabalho.

Na seção 5.4 vamos apresentar uma abordagem descrita em [5] e [15] que de maneira simples reduz a variabilidade do atraso de comunicação. Se com essa abordagem for razoável assumir o sistema como invariante no tempo, é possível analisar e projetar o sistema de controle com auxílio de teorias e resultados válidos apenas para os sistemas invariantes no tempo.

5.1 – 1º Estudo de Caso: Sistema de 2ª ordem

Um modelo de um sistema de 2ª ordem é apresentado em [41]. Esse modelo é utilizado em [41] para exemplificar como sistemas lineares invariantes no tempo podem ser modelados com a ajuda de ferramentas do Matlab (Matlab é uma linguagem de programação de alto nível e um ambiente de desenvolvimento interativo). Este modelo linear é bastante simples e serve como um ponto de partida para testar e ajustar o sistema de comunicação que queremos desenvolver. O modelo linear invariante no tempo pode ter seus parâmetros ajustados de forma que a resolução numérica de suas equações diferenciais ordinárias não exija métodos muitos sofisticados. Vamos utilizar o método de Runge Kutta Clássico, com passo de tempo de 0.001. A figura 5.1 ilustra o sistema que vamos descrever.

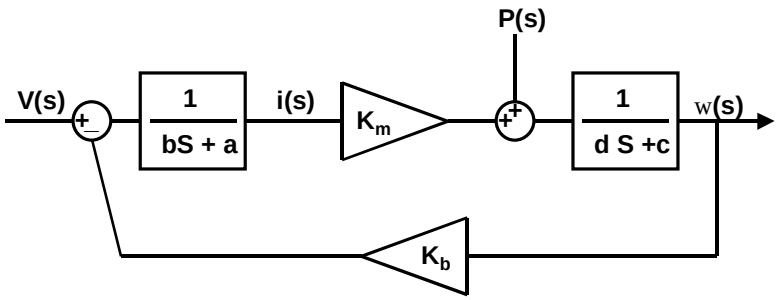


Figura 5.1 – Diagrama de blocos de um sistema linear invariante no tempo de segunda ordem.

5.1.1 – Descrição do Modelo e objetivos de controle

No modelo da figura 5.1, vamos utilizar os parâmetros descritos na tabela 5.1

.Tabela 5.1 –Parâmetros do sistema de 2ª ordem

Símbolo	Valor
a	2.0
b	0.5
c	0.2
d	0.02
K_a	0.1
K_b	0.1

Tabela 5.1 –Parâmetros do sistema de 2ª ordem (continuação)

Símbolo	Descrição
V(t)	Sinal de entrada
i(t)	Variável de estado
ω(t)	Variável de estado
P(t)	Sinal de Perturbação

Informações adicionais:

Ts = Período de Amostragem = 100 ms

h = Passo de tempo fixo para resolução numérica = 1 ms

Método numérico: Runge Kutta de 4ª ordem

Como foi descrito, esses parâmetros não se baseiam em nenhum sistema real e servem para realizar um estudo de caso bastante simplificado, que deve servir como um passo básico para o desenvolvimento de outros estudos de caso mais complexos.

Uma representação no espaço de estados deste modelo de 2ª ordem é dada abaixo:

$$\frac{d}{dt}\begin{bmatrix} i(t) \\ \omega(t) \end{bmatrix} = \begin{bmatrix} -\frac{a}{b} & -\frac{K_b}{b} \\ \frac{K_a}{d} & \frac{c}{d} \end{bmatrix}\begin{bmatrix} i(t) \\ \omega(t) \end{bmatrix} + \begin{bmatrix} \frac{1}{b} \\ 0 \end{bmatrix} V(t) + \begin{bmatrix} 0 \\ \frac{1}{d} \end{bmatrix} P(t) \qquad (5\text{-}1)$$

Substituindo os valores da tabela 5.1, e obtendo a função de transferência de ω(s)/V(s), para P(s) = 0, vamos obter a seguinte função de transferência G(s):

$$G(s) = \frac{\omega(s)}{V_{entrada}(s)} = \frac{10}{s^2 + 14s + 41} = \frac{10}{(s + 9.8284)(s + 4.1716)} \qquad (5\text{-}2)$$

Serão descritos os objetivos de controle.

Um primeiro objetivo desse sistema de controle é: ω(t) deve rastrear um sinal de baixa freqüência de referência r(t). Vamos definir r(t) como um sinal aperiódico do tipo degrau, como definido abaixo:

$$r(t) = \begin{cases} 0, & t < 0 \\ R_0, & t > 0 \end{cases} \qquad (5\text{-}3)$$

Um segundo objetivo é garantir que se um sinal de perturbação P(t), um sinal do tipo degrau, for aplicado ao sistema, os efeitos dessa perturbação possam ser compensados e ω(t) continue a rastrear o valor da referência r(t) sem erro em regime. Vamos definir P(t) como uma função também do tipo degrau, mas atrasada no tempo em 5 segundos, como abaixo:

$$P(t) = \begin{cases} 0, & t < 5 \\ P_0, & t > 5 \end{cases} \qquad (5\text{-}4)$$

Nesse caso, ω(s) pode ser descrita pela equação abaixo:

$$\omega(s) = V_{entrada}(s)\frac{10}{s^2 + 14s + 41} + P(s)\frac{(50s + 200)}{s^2 + 14s + 41}, \text{ para } \begin{cases} V_{entrada}(s) = \dfrac{R_0}{s} \\ P(s) = \dfrac{P_0}{s}e^{-5s} \end{cases} \qquad (5\text{-}5)$$

Um terceiro objetivo é ter um projeto robusto em relação a atrasos de comunicação, que vamos assumir que são estáticos e desconhecidos, mas limitados por um valor não muito maior que 200 ms. Embora um atraso estático de comunicação desconhecido não possa ser explicitamente compensado pela lei de controle, por exemplo usando um filtro *lead* ou uma abordagem de Preditor de *Smith*, um atraso estático desconhecido pode ser considerado no projeto a partir da análise no domínio da freqüência do sistema, através da margem de fase e sua respectiva freqüência de cruzamento. Com isso é possível calcular a margem ao atraso do sistema, ou seja, quanto de atraso estático na malha o sistema pode suportar sem que este se torne instável. Através da margem de atraso, não é necessário conhecer exatamente o tempo de atraso estático na malha, mas apenas garantir que este valor não ultrapasse um certo limiar. Assume-se nessas análise que o atraso é estático. A figura 5.2 ilustra como se obtém a margem de fase e a freqüência de cruzamento a partir da transferência de malha L(s), utilizando-se do diagrama de Bode ou do diagrama de Nyquist. De acordo com Ogata, em [42], um sistema pode operar satisfatoriamente se tiver uma margem de fase entre 30° e 60°, e um margem de fase positiva indica estabilidade, e uma margem de fase negativa a instabilidade do sistema.

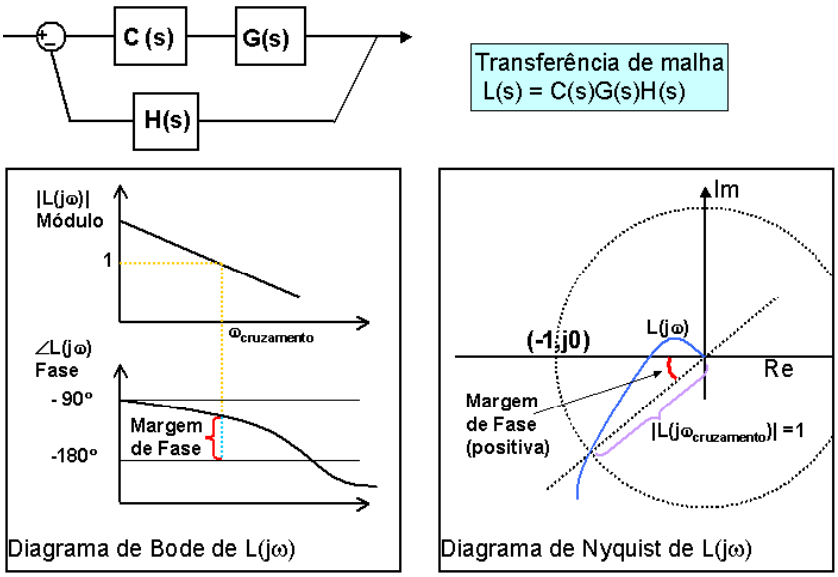


Figura 5.2 – Esboço de como se obter a margem de fase e a freqüência de cruzamento $\omega_{cruzamento}$, a partir do transferência de malha L(s) = C(s)G(s)H(s), e dos diagramas de Bode ou de Nyquist.

A margem ao atraso pode ser calculada com a seguinte fórmula, apresentada em [45]:

$$\text{Margem ao atraso} = \frac{\text{Margem de Fase (em rad)}}{\text{Frequência de Cruzamento (rad/s)}} \quad (5\text{-}6)$$

Vamos exemplificar:

Exemplo 1:

Seja C(s) = $1+\frac{4.5}{s}$, G(s) = $\frac{10}{s^2+14s+41}$ e H(s) = 1, na figura 5.4. Então a análise do Diagrama de Bode de L(s) = $\left(1+\frac{4.5}{s}\right)\left(\frac{10}{s^2+14s+41}\right)$ nos fornece que a fase de margem de 82.7° e freqüência de cruzamento de 1.09 rad/s. Como a margem de fase é positiva, o sistema é estável. A margem de atraso desse sistema é de:

$$\text{Margem ao atraso: } \frac{\left(82.7/180\right)\pi}{1.09} = \frac{1.443\,\text{rad}}{1.09\,\text{rad/s}} = 1.324\,\text{segundos}$$

A figura 5.3 ilustra dois atrasos estáticos na malha, τ_1 e τ_2.

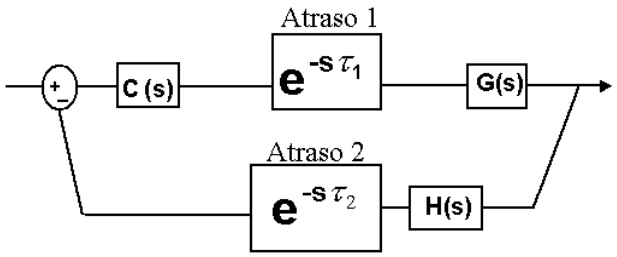


Figura 5.3 – Atrasos estáticos de ida e volta τ_1 e τ_2.

Pelo exemplo 1, para o sistema ser estável, a soma dos atrasos estáticos τ_1 e τ_2 deve ser menor que a margem de atraso de L(s) = $\left(1+\frac{4.5}{s}\right)\left(\frac{10}{s^2+14s+41}\right)$, que é de 1.324 segundos. Assim:

$\tau_1 + \tau_2 <$ Margem ao atraso = 1.324 $\Rightarrow$ sistema estável

5.1.2 –Controle Contínuo no tempo

Nesta seção, serão descritas algumas leis de controle que se baseiam em projetos de sistemas de controle contínuo no tempo. No entanto, esse trabalho apresenta sistemas de controle digitais, ou seja, dados analógicos são convertidos em sinais digitais e as leis de controle são discretizadas e processadas por um sistema computacional digital. Na seção 5.3, será apresentada uma introdução ao controle digital de sistemas dinâmicos e uma forma de se obter leis de controle discretas equivalentes as suas versões contínuas.

Uma primeira abordagem para fazer com que a velocidade $\omega(t)$ rastreie um valor de referência r(t) é aplicar um sinal de entrada V(t) igual a r(t) multiplicada pelo inverso do ganho DC da planta. O ganho DC pode ser calculado através de (5-2), fazendo s tender a zero:

$$\text{Ganho DC} = \lim_{s\to 0} \frac{10}{s^2 + 14s + 41} = \frac{10}{41}$$

Assim, a figura 5.4 ilustra a primeira abordagem de controle, que é feita à malha aberta, partindo do conhecimento do ganho DC da planta.

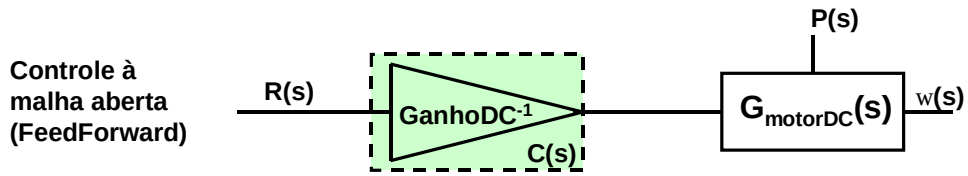


Figura 5.4 – Controle à malha aberta (*FeedForward*), compensa-se a diferença entre y(t) e r(t) ajustando-se um ganho.

Entretanto essa abordagem de controle assume que ao longo do tempo nenhum sinal externo de perturbação P(t) será aplicado ao sistema, o que não é razoável nesse exemplo. Para verificar o erro obtido, basta utilizar a equação (5-5) e o teorema do valor final para obter o valor de $\omega(t)$ em regime:

$$V_{\text{entrada}}(s) = \frac{R_0}{s}\left(\frac{1}{\text{ganho DC}}\right) = \frac{1}{s}\left(\frac{41}{10}\right)$$

$$\omega(t\to\infty) = \lim_{s\to 0} s\left(\frac{R_0}{s}\left(\frac{41}{10}\right)\frac{10}{s^2+14s+41} + \frac{P_0}{s}\frac{(50s+200)}{s^2+14s+41}\right) \qquad (5.8)$$

$$\omega(t\to\infty) = \left(\left(\frac{41}{10}\right)\frac{10}{0+0+41} + P_0\frac{(0+200)}{0+0+41}\right) = \left(R_0 + P_0\frac{200}{41}\right)$$

Se P_0 = -0.1 e R_0 = 1, o valor de $\omega(t)$ para um tempo muito grande será de $\omega(t\to\infty) = 1 - 0.1\times(200/41) = 0.5122$ rad/s e não 1 rad/s como o desejado.

Por outro lado, um atraso de comunicação estático vai apenas atrasar a resposta do sistema no tempo como um todo, mas não vai causar instabilidade.

O resultado do controle a malha aberta, está apresentado pela curva em cor azul na figura 5.12, ao final desta seção. Com essa abordagem conseguimos abordar apenas a dois objetivos dos 3 que gostaríamos de obter, pois esta abordagem não rejeita perturbações P(t) do tipo descrito pela definição em (5-4).

O próximo passo é utilizarmos controle a malha fechada. Nesse caso um atraso estático de comunicação pode reduzir a margem de fase do sistema e instabilizá-lo, por isso, quando aplicável, o sistema será projetado para ter uma margem ao atraso da ordem de 1 segundo ou superior, para termos um projeto robusto a um atraso de comunicação estático limitado que, por hipótese, é um valor muito menor que 1 segundo. A figura 5.5 ilustra a primeira abordagem de controle a malha fechada, utilizando um controle proporcional.

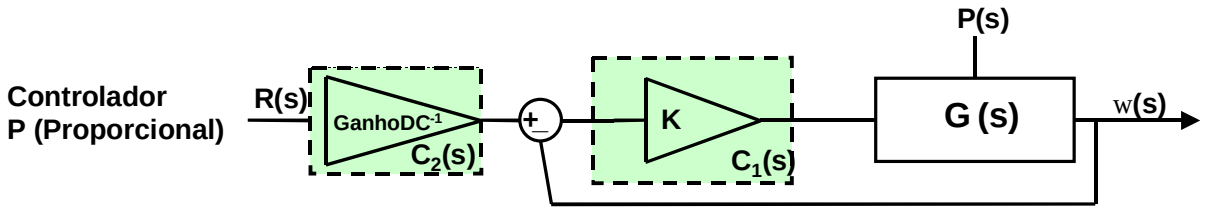


Figura 5.5 – Controle a malha fechada, e compensador proporcional $C_1(s) = K$. $C_2(s)$ compensa pelo ganho DC do sistema a malha fechada.

Utilizando a abordagem da figura 5.5, com um compensador $C_1(s)$, temos que $\omega(s)$ pode ser expresso por:

$$\omega(s) = R(s)G_1(s) + P(s)G_2(s),$$

$$\text{para} \begin{cases} R(s) = \dfrac{R_0^{'}}{s}, & G_1 = \dfrac{\mathbf{C(s)}K_a}{s^2(db) + s(da + cb) + (ac + K_b K_a) + \mathbf{C(s)}K_a} \\ P(s) = \dfrac{P_0}{s} e^{-5s}, & G_2 = \dfrac{(sb + a)}{s^2(db) + s(da + cb) + (ac + K_b K_a) + \mathbf{C(s)}K_a} \end{cases} \quad (5\text{-}9)$$

Substituindo e m(5-9) C(s) por K, e aplicando o teorema do valor final, temos que:

$$\omega(t \to \infty) = \lim_{s \to 0} (sR(s)G_1(s) + sP(s)G_2(s)) = R_0^{'} \frac{K_a}{(ac + K_b K_a) + K K_a} + P_0 \frac{a}{(ac + K_b K_a) + K K_a} \quad .$$

$$\text{Se } R_0^{'} = R_0 \frac{(ac + K_b K_a) + K K_a}{K_a}$$

$$\omega(t \to \infty) = \lim_{s \to 0} (sR(s)G_1(s) + sP(s)G_2(s)) = R_0 + P_0 \frac{a}{(ac + K_b K_a) + K K_a} \quad .$$

Pelo resultado acima, sempre vai haver um erro em regime. No entanto, quanto maior o valor do ganho K, menor será esse erro em regime. Infelizmente, o valor do ganho K não pode ser arbitrariamente escolhido para reduzir o erro em regime, pois deve-se considerar a relação entre o valor do ganho K e a margem de atraso.

Para averiguar a relação entre o valor de K e a margem ao atraso, foi utilizada uma ferramenta do Matlab chamada sisotool, que permite o ajuste do ganho K e a visualização imediata das medidas de margem de fase do sistema e sua freqüência de cruzamento, baseadas no Diagrama de Bode da função de transferência de malha $L(s) = C(s)G(s)$. A figura 5.6 ilustra o diagrama de Bode para o caso de $C(s) = K = 4.5$.

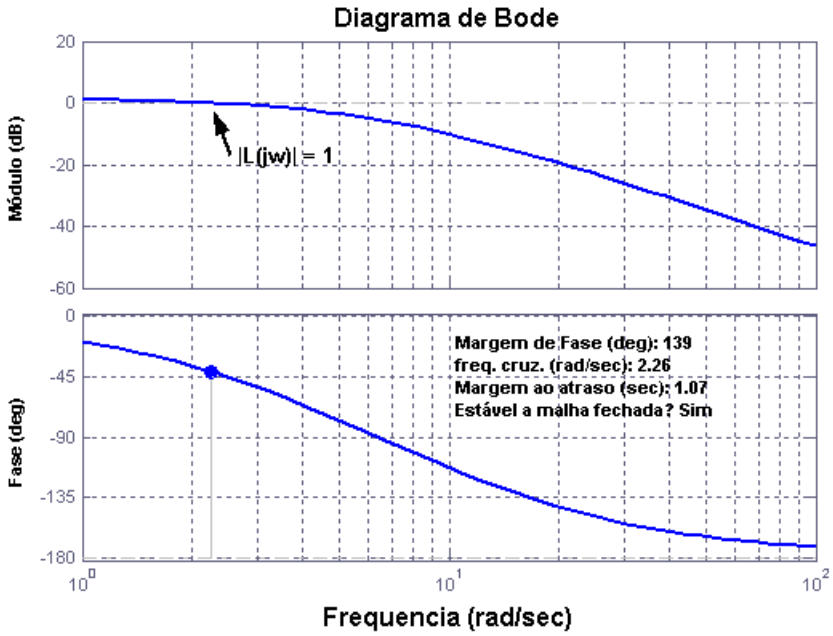


Figura 5.6 – Diagrama de Bode da transferência de malha L(s) = C(s)G(s), C(s) = 4.5

Vamos apresentar alguns casos de como a margem de atraso varia conforme aumentamos o ganho K. A figura 5.7 ilustra o diagrama de Nyquist em relação a L(s)= KG(s), para valores de K = 4, 4.5,10 e 75.

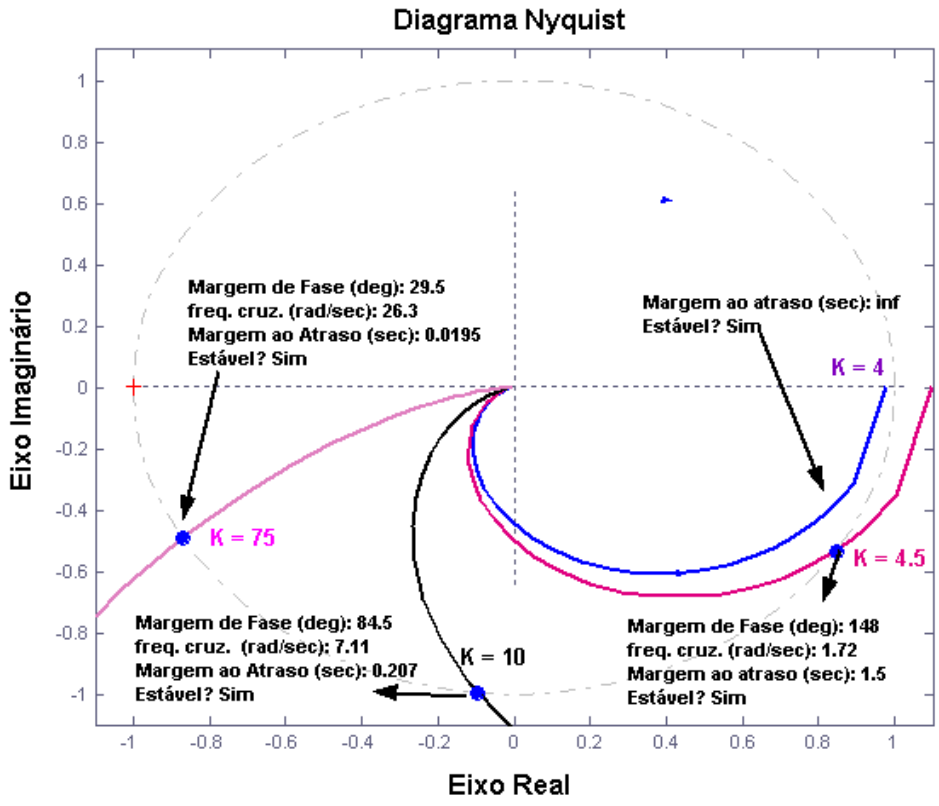


Figura 5.7 – Seleção dos valores de K e margem ao atraso associada. Conforme aumentamos o valor de K a margem ao atraso é reduzida. Para K ≤ 4.1 a margem ao atraso é infinita.

Simulações para o caso de K = 4 estão apresentadas na figura 5.10.

A próxima abordagem de controle é utilizar um compensador com ação proporcional e integral para rejeitar a perturbação P(t), como definida em (5-4).. A figura 5.8 ilustra essa nova abordagem de controle.

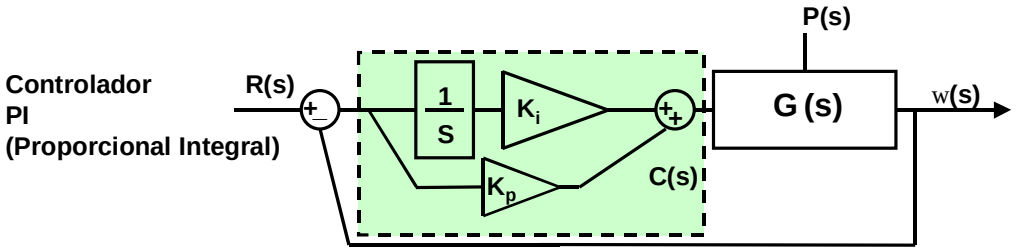


Figura 5.8 – Compensador C(s) do tipo PI (Proporcional Integral)

Na figura 5.8, $C(s) = \frac{K_p(s + K_i/K_p)}{s}$, o que faz com que apareça um termo de s no numerador de $G_2(s)$, definida em (5-9). Substuindo C(s) por $\frac{K_p(s + K_i/K_p)}{s}$ em (5-9) e aplicando o teorema do valor final, verifica-se que $\omega(t)$ tende para o valor da referência R_0 para $t \to \infty$:

$$\omega(t \to \infty) = \lim_{s \to 0} (sR(s)G_1(s) + sP(s)G_2(s)) = R_0$$

Para projetar os valores de K_p e K_i foi utilizada a ferramenta sisotool, que possibilita a verificação da resposta ao degrau e do diagrama de Bode ao mesmo tempo em que varia-se a posição do zero em K_i/K_p e o ganho K_p. Nesse tipo de projeto, ter uma margem ao atraso acima de 1 segundo e um tempo de subida pequeno são objetivos contraditórios, já que para aumentarmos a margem ao atraso do sistema, temos que na prática tornar este sistema mais lento, utilizando pequenos valores de K_p. Após algumas tentativas e verificando o desempenho do sistema para diferentes atrasos estáticos (como será visto na figura 5.10), os valores de $K_i = 4.5$ e $K_p = 1$ parecem ser razoáveis, pois fornecem uma margem ao atraso de 1.33 segundos, embora o sistema tenha uma resposta mais lenta que nos dois casos anteriores, como será apresentado na figura 5.10.

As três primeiras abordagens utilizaram métodos de transformada para alcançar aos objetivos de controle, assumindo que o sistema é linear invariante no tempo. Uma última abordagem é descrita pela figura 5.9, que utiliza métodos no espaço de estados, utilizando um integrador. Com isso temos uma terceira variável de estado que vamos chamar de q(t). Para projetar os valores dos ganhos K_1, vamos resolver o problema do Regulador Linear Quadrático para um tempo infinito, e especificar alguns dos quesitos de projeto através dos índices de uma função de custo quadrática.

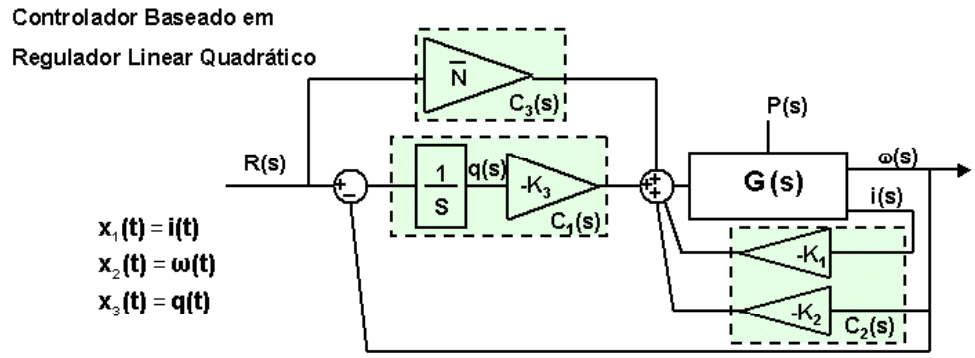


Figura 5.9 – Nessa abordagem, vamos calcular um vetor **K** = [K_1 K_2 K_3], do sinal de controle ótimo u(t) = - **Kx(t)**

Vamos representar o sistema da figura 5.9 no espaço de estados, na forma $\dot{\mathbf{x}} = \mathbf{Ax} + \mathbf{Bu}$. A variável de estado q(t) representa a integral do erro entre a variável de estado ω(t) e a referência r(t), ou seja, $\frac{dq(t)}{dt} = r(t) - \omega(t)$. O termo $\overline{N}$ representa o inverso do ganho DC entre a entrada V(s) e a saída ω(s) do sistema a malha fechada em relação aos ganhos K_1 e K_2, não incluindo a malha com o integrador nos cálculos. A forma de calcular $\overline{N}$

$$\frac{d}{dt}\begin{bmatrix} i(t) \\ \omega(t) \\ q(t) \end{bmatrix} = \begin{bmatrix} -\frac{a}{b} & -\frac{K_b}{b} & 0 \\ \frac{K_a}{d} & \frac{c}{d} & 0 \\ 0 & -1 & 0 \end{bmatrix}\begin{bmatrix} i(t) \\ \omega(t) \\ q(t) \end{bmatrix} + \begin{bmatrix} \frac{1}{b} \\ 0 \\ 0 \end{bmatrix} V_{entrada}(t) + \begin{bmatrix} 0 \\ \frac{1}{d} \\ 0 \end{bmatrix} P(t) + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} r(t) \tag{5-11}$$

$$V_{entrada}(t) = -K_1 i(t) - K_2 \omega(t) - K_3 q(t) + \overline{N}\, r(t) = -\begin{bmatrix} K_1 & K_2 & K_3 \end{bmatrix}\begin{bmatrix} i(t) \\ \omega(t) \\ q(t) \end{bmatrix} + r(t)\overline{N}$$

$$\overline{N} = N_V + \begin{bmatrix} K_1 & K_2 \end{bmatrix}\begin{bmatrix} N_{x1} \\ N_{x2} \end{bmatrix} \tag{5-12}$$

$$\begin{bmatrix} N_{x1} \\ N_{x2} \\ N_u \end{bmatrix} = \begin{bmatrix} A & B \\ C & D \end{bmatrix}^{-1}\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} -\frac{a}{b} & -\frac{K_b}{b} & 0 \\ \frac{K_a}{d} & \frac{c}{d} & 0 \\ 0 & 1 & 0 \end{bmatrix}^{-1}\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{5-13}$$

Assume-se que as medidas das variáveis de estado i(t), ω(t) e q(t) são todas disponíveis para serem utilizadas para a realimentação de estados. O problema do Regulador Quadrático Linear para tempo infinito pode ser expresso como o problema de encontrar o sinal de controle u(t) = - **Kx**(t) , que minimiza a função de custo:

$$J = \int_{t=0}^{\infty} (x^T(t)Qx(t) + u^T(t)Ru(t))dt \qquad (5\text{-}14)$$

A matriz Q é uma matriz simétrica real definidas positivas (ou semi definida positiva) e R é uma matriz simétrica definida positiva, ou seja, $x^T Q\, x \geq 0 \, e \, u^T R\, u > 0$. As matrizes Q e R que determinam a relevância das variáveis de estado e do sinal de controle nessa função de custo J, respectivamente. A solução a esse problema de minimização é dada por:

$$\mathbf{K} = \mathbf{R}^{-1}\mathbf{B}^{\mathbf{T}}\mathbf{P} \qquad (5\text{-}15)$$

Onde P é obtido resolvendo-se a equação matricial de Riccati reduzida:

$$\mathbf{A}^{\mathbf{T}}\mathbf{P} + \mathbf{PA} - \mathbf{PBR}^{-1}\mathbf{B}^{\mathbf{T}}\mathbf{P} + \mathbf{Q} = 0 \qquad (5\text{-}16)$$

As matrizes Q e R foram definidas como:

$$\mathbf{Q} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 30 \end{bmatrix}, \mathbf{R} = 1 \qquad (5\text{-}17)$$

Isso resulta em:

$$J = \int_{t=0}^{\infty} (1)\ i^2(t) + (1)\omega^2(t) + (30)q^2(t) + (1)\ V(t)\ \ dt \qquad (5\text{-}18)$$

$$\mathbf{K} = \begin{bmatrix} 0.7894 & 0.5562 & -5.4772 \end{bmatrix} \text{ e } \overline{N} = 6.2350 \qquad (5\text{-}19)$$

A função de custo J penaliza a variável de estado q(t) e para minimizar J, q(t) deve ser o menor possível ao longo do tempo. Esses resultados foram obtidos com a ajuda de um ferramenta do Matlab chamada de lqr( ), que recebe como argumentos A, B, Q e Ṛ e retorna o valor de K, depois de resolver o valor da matriz P da equação (5-14).

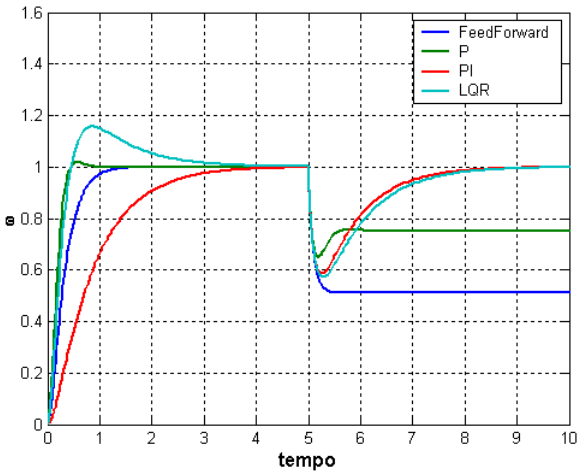
A tabela 5-2 resume os tipos de compensadores, os valores de seus parâmetros, e a margem ao atraso associada a cada caso.

Tabela 5-2: C(s), Ganhos e Margem ao atraso

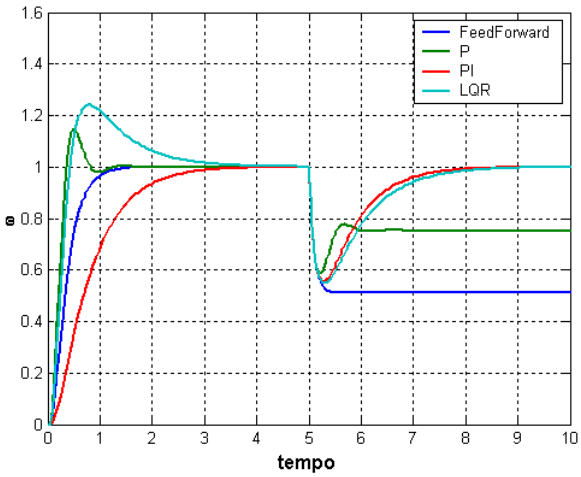
Compensador	Margem ao Atraso
FeedForward $C(s) = \frac{1}{\text{Ganho DC}} = 4.1$	-
P: $C(s) = K = 4$	∞
PI: $C(s) = \frac{K_p s + K_i}{s} = \frac{s + 4.5}{s}$	1.330 segundos
LQR(*):	1.1935 segundos

(*) valor obtido em simulação.

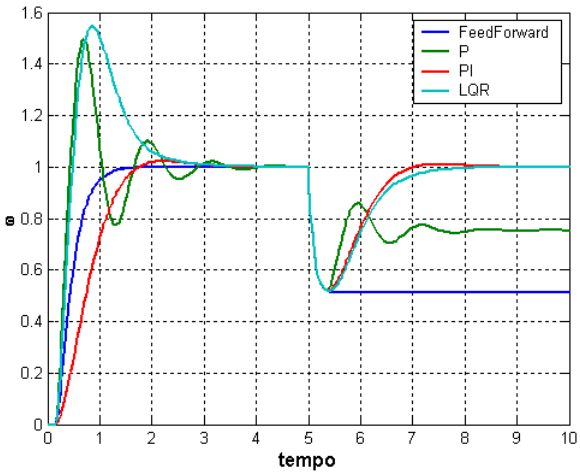
A figura 5.12, apresenta simulações do sistema, para um atraso de ida τ_1 e de volta τ_2, de forma que ambos tenham o mesmo valor: $\tau_1 = \tau_2$.



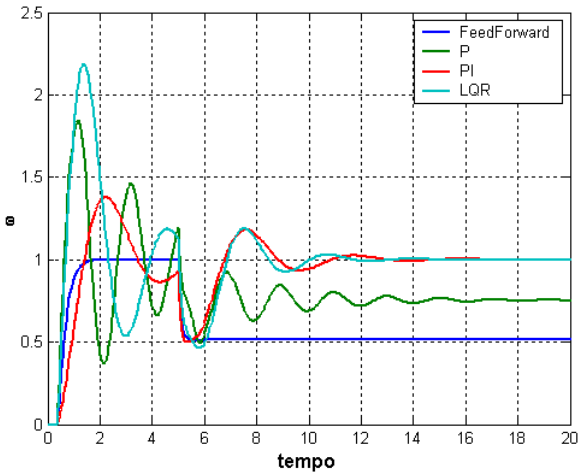
(a) $\tau_1 + \tau_2 = 0$ segundos



(c) $\tau_1 + \tau_2 = 100$ ms.



(b) $\tau_1 + \tau_2 = 300$ ms



(d) $\tau_1 + \tau_2 = 750$ ms

Figuras 5.10 – Curvas de ω(t), para as várias abordagens de controle e diferentes atrasos estáticos na malha.

5.1.3 – Simulação numérica

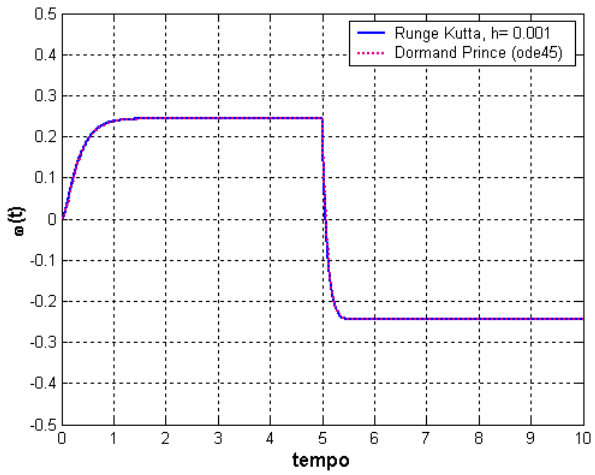
Vamos apresentar um esboço de uma rotina que executa no RTLinux/Free e simula a planta de um sistema de 2ª ordem, descrita na seção 5.1.1. O método de resolução numérica de EDO utilizado é o método de Runge Kutta Clássico (Runge Kutta de 4ª ordem). O programa completo pode ser obtido no apêndice A, no programa motor.c

```
/* Esboço de Simulação no RTLinux/Free do primeiro estudo de caso */
```

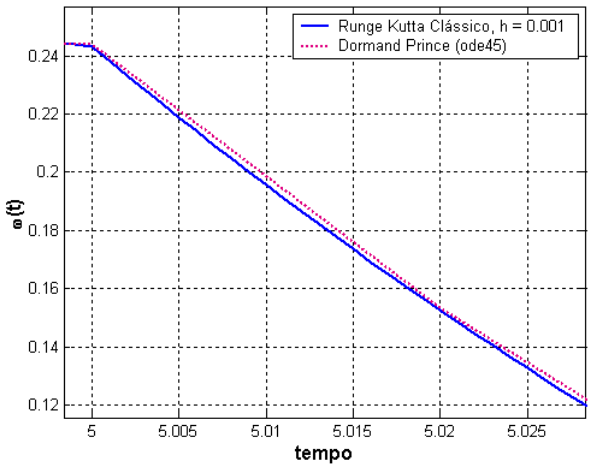
```
void * thread_motor(void)
{
	pthread_self()->uses_fp = 0;
	pthread_setfp_np (pthread_self(), 1);
	/*habilita o uso de números em ponto flutuante */
	struct sched_param p;
	/*struct necessário para alterar a prioridade do thread */
	p . sched_priority = 99;/* prioridade */
	pthread_setschedparam (pthread_self(), SCHED_FIFO, &p);
	double t,u,h,R,L,J,Kb,Kf,Km,a,b,c,d,e,f;
	t=0;		/*tempo de simulação */
	h=0.005;	/*passo de integração */
	u=0;		 /*sinal de controle */
	R  = 2.0;    /*Ohm   Resistência da Armadura */
	L  = 0.5;	  /*Henry Indutancia  da Armadura */
	Km = Kb = 0.1;
	Kf = 0.2;
	J  = 0.02;
	a = -R/L;
	b = -Kb/L;	  /* [di]  = [a b][i]  + [c]u  + [0]Td */
	c =  1/L;	  /* [dw]    [d e][w]    [0]     [f] */
	d =  Km/J;
	e = -Kf/J;
	f =  1/J;
	double i,w,di,dw;
	i=w=0;
	di=dw=0;
	double x[2],dx[4][2];
	int it =0;
	double h_tmp =h/2;
	pthread_make_periodic_np (pthread_self(), gethrtime(), 1000000);
	/*executa a cada 1 ms */

	while(1)
	{
		pthread_wait_np (); /*a thread vai dormir e acorda em 1 ms */
		/*Utilizar o comando rtf_put() para enviar amostras para um processo do
		Linux. Ex:
		- Enviando dados para /dev/rtf20
			rtf_put(20,&i,sizeof(i));
			rtf_put(20,&w,sizeof(w));
		Utilizar o comando rtf_get() para verificar se algum dado chegou. Ex:
		- Verificando se dados de controle chegaram em /dev/rtf21
			rtf_get(23,&u,sizeof(u));
		Observação : deve-se reservar memória para as filas Fifos antes
		de usá-las, e isso é feito através de comandos como rtf_create() durante
		no init_module(). */
		/*Runge Kutta de 4ª ordem*/
		x[0]	= i;
		x[1]	= w;
```

```
            it=0;
            for(;it<4;it++)
            {
                  if(it < 2)/*nao influencia no calculo de k1*/
                  {
                        h_tmp=h/2; /*para o calculo de k2 e k3*/
                  }
                  else
                  {
                        h_tmp=h;/*para o calculo de k4*/
                  }

                  dx[it][0] = a*x[0] + b*x[1] + c*u;
                  dx[it][1] = d*x[0] + e*x[1] + f*Td;
                  if(it < 3)/*so e necessario para calcular k2 e k3*/
                  {
                        x[0]  = i  + h_tmp * dx[it][0];
                        x[1]  = w  + h_tmp * dx[it][1];
                  }
            }
                        /*atualizando os estados*/
            i= i + (h/6) * (dx[0][0] + 2*dx[1][0] + 2*dx[2][0] + dx[3][0]);
            w= w + (h/6) * (dx[0][1] + 2*dx[1][1] + 2*dx[2][1] + dx[3][1]);
            t = t+h;
      }
      return 0;
}/*Fim do Esboço de Simulação no RTLinux/Free */
```

Esse esboço mostra como é simples implementar esse primeiro estudo de caso. No entanto, a estabilidade numérica do método de Runge Kutta de 4ª ordem para esse exemplo deve ser avaliada, variando-se o passo de tempo h e através de comparações com um outro método numérico de ordem superior, com passo de tempo variável. O método escolhido para comparação é o método de Dormand-Prince 5(4), disponível no Matlab como uma função chamada de ode45(), que utiliza um par de métodos de Runge Kutta de ordem 4 e 5 para monitorar e controlar os erros, adaptando o tamanho do passo de tempo h. A figura 5.12 apresenta a resposta do sistema a entrada V(t) = r(t) e P(t), definidos em (5-3) e (5-4), para $R_0 = 1$ e $P_0 = -0.1$. A ferramenta ode45 controla o erro a cada passo, utilizando como referência os limiares de erro relativo e absoluto fornecidos pelo usuário. A tolerância ao erro relativo utilizada é de 10^{-6}. Não será calculada a resposta exata, assumindo-se que o método do ode45 é apurado o suficiente para este problema.



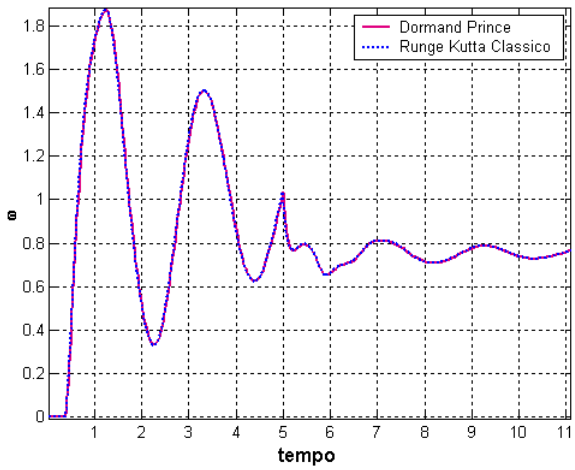
(a) ω(t) para t entre 0 e 10 s



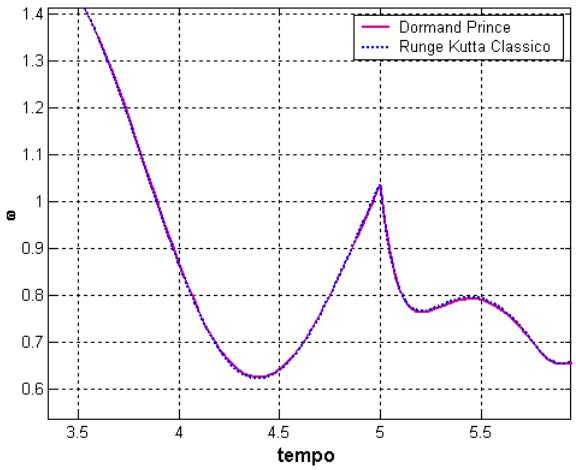
(b) Ampliando ω(t) para t entre 0 e 1.5 s

Figura 5.11 – Comparação entre o método numérico de Runge Kutta Clássico(4ª ordem), com passo de tempo h = 0.001 s, e Runge Kutta (5)4, de passo variável, com tolerância a erro relativo de 10^{-6}.

A figura 5.12 mostra a resposta do sistema de controle com C(s) = K (proporcional), para o caso em que temos um atraso estático de 750 ms.



(a) ω(t)



(b) ω(t) entre 3.5 e 6 segundos

Figura 5.12- Comparação entre: método de Runge Kutta Clássico, com passo de tempo 0.001 segundos, e o método de Dormand Prince, de passo variável, com tolerância a erro relativo de 10^{-6}.

Nas figuras 5.11 e 5.12, o método numérico de Runge Kutta Clássico, para h =0.001, sempre exibe erros em relação a curva gerada pelo método de Dormand Prince, no entanto esses erros não são significativos para esse estudo de caso, em que o método parece ser estável numericamente para os tipos de entradas que vamos aplicar ao sistema. A figura 5.14 mostra a mesma simulação da figura 5.11, utilizando diferentes valores de h, o passo de tempo fixo. Para h ≥ 300 ms, os erros de simulação se acumulam de forma não limitada, e a resolução numérica é dita instável.

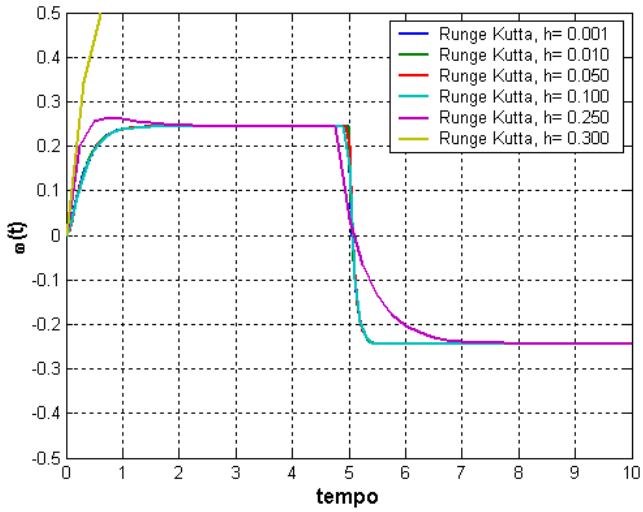


Figura 5.13 – Simulações com o método de Euler para passos de tempo h = 0.001, 0.010, 0.050, 0.100, 0.250 e 0.300.

Observa-se que nesse estudo de caso estamos limitando o sinal de Referência r(t) e o sinal de perturbação P(t) a sinais de baixa freqüência, o que reduz a necessidade de um passo de tempo pequeno para evitar a instabilidade numérica, e facilita, entre outras coisas, na depuração de programas e na realização de testes das aplicações implementadas para o Linux e RTLinux/Free.

5.2 – 2º Estudo de Caso: Modelo de um pêndulo rotacional

Nesse 2º estudo de caso, vamos estudar o problema do pêndulo invertido rotacional, que é esboçado na figura 5.15, e é baseado na referência [40]. O sistema do pêndulo em [40] é constituído de um motor DC, engrenagens, uma trave (ou braço) do pêndulo de comprimento L_T, e um pêndulo de comprimento L_P. O modelo que descreve o pêndulo é não linear, em termos de suas 4 variáveis dependentes $\theta_P(t)$, $\theta_T(t)$, $d\theta_P(t)/dt$ e $d\theta_T(t)/dt$, que serão descritas na seção seguinte. Para simular esse modelo não linear é utilizado o método de Runge Kutta de 4ª ordem, para um passo fixo de tempo de 0.0005 segundo.

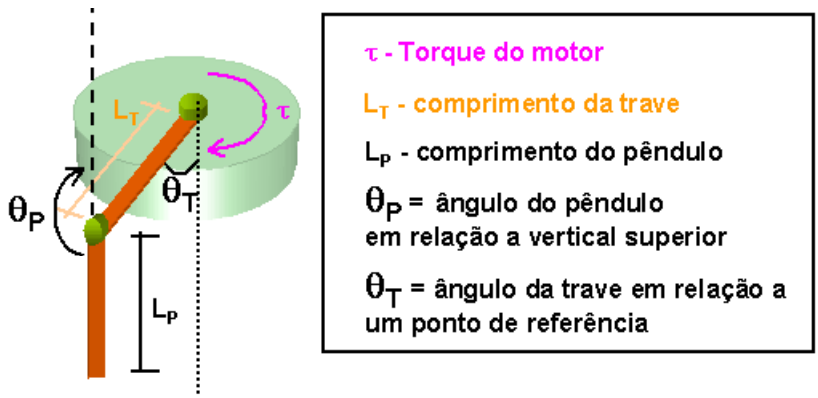


Figura 5.15 – Esboço de um Pêndulo Rotacional.

A figura 5.16 ilustra os dois pontos de equilíbrio do sistema do pêndulo. O nosso interesse é manter o pêndulo na posição de equilíbrio instável, aplicando um sinal de controle de tensão no motor

DC, que em conseqüência aplica um torque τ(t), como ilustrado na figura 5.15. O nome de pêndulo rotacional invertido deve-se ao movimento de rotação da trave (braço) do pêndulo para equilibrá-lo na posição invertida.

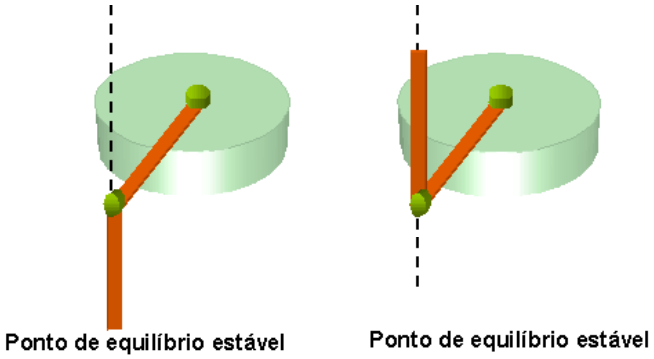


Figura 5.16 – Esboço dos Pontos de equilíbrio do sistema do pêndulo invertido rotacional: ponto de equilíbrio estável e instável.

Em [40], para equilibrar o pêndulo na posição vertical superior, são utilizados 3 blocos de diferentes funções, que estão ilustradas na figura 5.17.

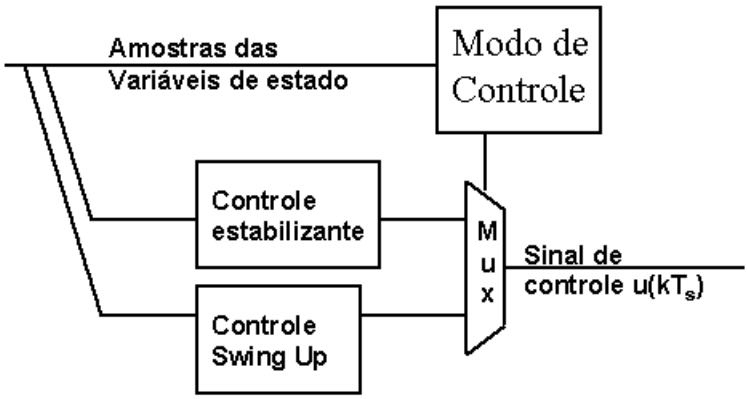


Figura 5.17 – Modos de controle

Cada um desses blocos será explicado a seguir:

- Controle de “Swing Up”: é uma lei de controle que leva o pêndulo do seu ponto de equilíbrio estável para próximo do ponto de equilíbrio instável. Isso é feito oscilando a trave do pêndulo, até que o pêndulo tenha energia suficiente para passar perto do ponto de equilíbrio instável;
- a lei de controle estabilizante, que é uma lei de controle linear aplicada apenas quando o pêndulo está próximo o suficiente do ponto de equilíbrio e as suas velocidades angulares não excedem um certo limiar;
- por último temos o “modo de controle”, que decide quando se deve comutar entre a lei de controle de Swing Up ou o estabilizante, em função de algoritmos que permitam avaliar quando é possível aplicar controle linear ou não;

Nesse estudo, vamos descrever métodos no espaço de estados que permitam que se estabilize o pêndulo no seu ponto de equilíbrio instável, considerando que as condições iniciais do sistema são as

necessárias para que controle linear possa ser utilizado nessa tarefa. O bloco "modo de controle" e o "controle de Swing Up" serão apresentados na seção 5.2.2 e simulados na seção 5.2.3, mas não serão implementados nas aplicações cliente-servidor que utilizam o *Bluetooth*, já que o algoritmo de "Swing Up" é não linear e apresentou um comportamento imprevisível diante de um atraso de comunicação, além de instabilidade numérica , o que será explicado na seção a 5.2.3.

5.2.1 – Descrição do Modelo e Objetivos do Controle

Este sistema tem 4 variáveis de estado: $\theta_P(t)$, $\theta_T(t)$, $\frac{d\theta_P(t)}{dt} = \dot{\theta}_P$ e $\frac{d\theta_T(t)}{dt} = \dot{\theta}_T$. Estas variáveis de estado representam respectivamente: o ângulo do pêndulo com a vertical superior, o ângulo da trave que está conectada ao pêndulo em relação a uma dada posição, a velocidade angular do pêndulo e a velocidade angular da trave. Os ângulos $\theta_P(t)$ e $\theta_T(t)$ podem ser observados na figura 5.15. De [40], as equações diferenciais que descrevem o sistema são apresentadas abaixo:

$$\begin{aligned} &(m_p L_T^2 + J_b)\ddot{\theta}_T + m_p r_p L_T \ddot{\theta}_P \cos(\theta_P) - m_P r_p L_T \dot{\theta}_P^2 \sin(\theta_P) = t \\ &m_p r_p L_T \cos(\theta_p)\ddot{\theta}_T - m_p r_p L_T \sin(\theta_P)\dot{\theta}_P \dot{\theta}_T + m_p r_p^2 \ddot{\theta}_P - m_p r_p \sin(\theta_P) g = 0 \end{aligned} \quad (5\text{-}18)$$

No entanto, uma observação importante a ser feita é que o controle estabilizante, o controle de Swing Up e o "modo de controle" utilizam 3 diferentes medidas da mesma variável de estado do ângulo do pêndulo. As medidas utilizadas nesse trabalho vão ser listadas a seguir:

- $\theta_{Pfull}(t)$: é o ângulo que começa com 0° na posição do pêndulo para baixo e se acumula sem restrições, podendo ultrapassar o valor de 360° várias vezes. É a medida da qual as outras duas são obtidas, e é utilizada pelo "modo de controle";
- $\theta_{Psup}(t)$: é uma medida do ângulo do pêndulo com a vertical superior. $\theta_{Psup}(t)$ é dado pela seguinte fórmula: $\theta_{Psup}(t) = -\sin^{-1}(\sin(\theta_{Pfull}(t)))$. Além disso, é restrito a valores entre $-\pi/2 \leq \theta_{Psup}(t) \leq \pi/2$. É a medida utilizada pelo controle estabilizante;
- $\theta_{Pinf}(t)$: é uma medida do ângulo do pêndulo com a vertical inferior. $\theta_{Pinf}(t)$ é dado por: $\theta_{Psup}(t) = \tan 2^{-1}(\sin(\theta_{Pfull}(t)), \cos(\theta_{Pfull}(t)))$. $\theta_{Pinf}(t)$ é restrito a valores entre $-\pi \leq \theta_{Pinf}(t) \leq \pi$. Essa medida é utilizada pelo controle de Swing Up, para oscilar o pêndulo;
- $\theta_P(t)$: é a medida do ângulo do pêndulo com a vertical superior, mas diferente de $\theta_{Psup}(t)$, este não está restrito a valores entre $-\pi/2$ e $\pi/2$, e pode acumular livremente. É uma mesma medida defasada de 180° que $\theta_{Pfull}(t)$, e pode ser expressa por $\theta_{Pfull}(t) = (\theta_P(t) + \pi)$. Essa medida $\theta_P(t)$ é a medida que será utilizada nas equações (5-18).

Obviamente utilizar uma dessas medidas do ângulo do pêndulo no algoritmo de controle errado pode levar a falhas nesse algoritmo. No entanto, para a implementação do simulador, não é necessário implementar as 4 medidas de ângulos, basta a medida $\theta_P(t)$. Na implementação dos algoritmos de controle de Swing Up, estabilizante e "modo de controle", aí sim é necessário fazer a conversão de ângulo de $\theta_P(t)$

para $\theta_{Pfull}(t)$ e a partir disso calcular $\theta_{Psup}(t)$ e $\theta_{Pinf}(t)$. Como vamos apenas utilizar controle linear, não é inválido assumir que $\theta_{Psup}(t) = \theta_P(t)$ para a faixa de operação em que vamos analisar o sistema, mas isso só é válido se $-\pi/2 \leq \theta_P(t) \leq \pi/2$, para todo t. A tabela 5-3 apresenta um resumo do que foi descrito:

Tabela 5.3: Medidas do ângulo do pêndulo

Símbolo	Descrição	Restrições	Fórmula de conversão
$\theta_{Pfull}(t)$	Ângulo do pêndulo com a vertical inferior	-	$\theta_P(t) + \pi$
$\theta_{Psup}(t)$	Ângulo do pêndulo com a vertical superior	$-\pi/2 \leq \theta_{Psup}(t) \leq \pi/2$	$\theta_{Psup}(t) = -\sin^{-1}(\sin(\theta_{Pfull}(t)))$
$\theta_{Pinf}(t)$	Ângulo do pêndulo com a vertical inferior	$-\pi \leq \theta_{Pinf}(t) \leq \pi$	$\tan2^{-1}(\sin(\theta_{Pfull}(t)), \cos(\theta_{Pfull}(t)))$
$\theta_P(t)$	Ângulo do pêndulo com a vertical superior	-	-

A figura 5.18 abaixo ilustra as medidas dos ângulos $\theta_P(t)$, $\theta_{full}(t)$, $\theta_{Psup}(t)$ e $\theta_{Pinf}(t)$.

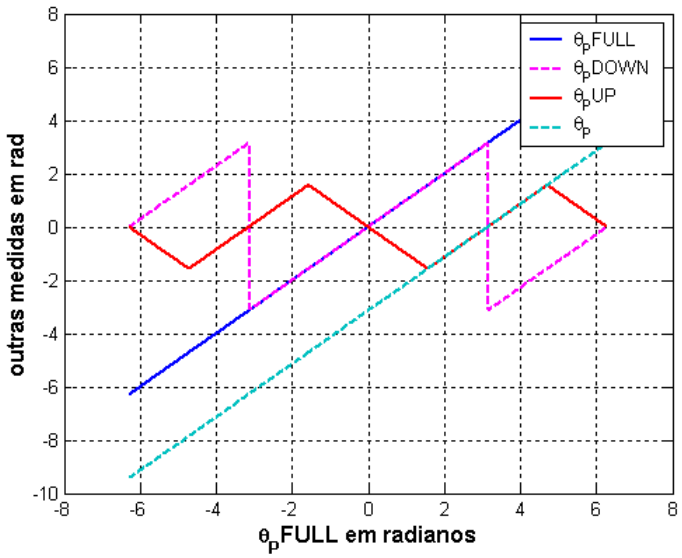


Figura 5.18 – Ângulos $\theta_P(t)$, $\theta_{Pfull}(t)$, $\theta_{Psup}(t)$ e $\theta_{Pinf}(t)$ em função de $\theta_{Pfull}(t)$.

Os termos das equações (5-18) são descritos pela tabela 5.4, cujos valores são os mesmos apresentados na especificação do sistema em [40]. Esse modelo não linear é o modelo que vamos utilizar para a simulação, no entanto para um projeto de controle linear, vamos trabalhar com modelo linearizado em torno do ponto de equilíbrio instável.

Tabela 5.4: Descrição dos parâmetros do sistema do pêndulo e suas variáveis de estado,

Símbolo	Descrição	Valor	Unidade
m_p	Massa do pêndulo	0.14	Kg
L_T	Comprimento da Trave	0.21	m
L_p	Comprimento do pêndulo	0.43	m
r_p	Metade do comprimento do pêndulo	$0.5 \times L_p$	m
J_b	Inércia da trave e engrenagens	0.0044	Kg/m^2
K_m	Constante de Torque do Motor	0.00767	Nm/A
K_b	Constante da força contra-eletromotriz	0.00767	V/(rad/s)
K_g	Razão entre as engrenagens	60.5	-
R	Resistência da Armadura	2.6	Ω
g	Aceleração da Gravidade	9.8	M/s^2
u(t)	Tensão de Entrada no motor DC	Entre –5 e +5	V
$\tau(t)$	Torque de entrada	-	N m
$\theta_T(t)$	Ângulo da trave em relação a posição zero	-	Rad
$\theta_P(t)$	Ângulo do pêndulo em relação a posição vertical superior	-	Rad
$\frac{d\theta_T(t)}{dt}$	Velocidade angular da trave	-	Rad/s
$\frac{d\theta_P(t)}{dt}$	Velocidade angular do pêndulo	-	Rad

De acordo com [40], o ângulo da trave do pêndulo é 0° em uma posição (única), em que um transdutor (potenciômetro) marca zero Volts e a tensão varia continuamente se a trave for deslocada um pouco dessa posição. Outra posição da trave também marca zero Volts, mas a tensão varia abruptamente de 0 Volts para + 5V ou –5V. Essa descontinuidade limita a faixa de ângulos da trave que podem ser medidos corretamente pelo potenciômetro.

As velocidades angulares do pêndulo e da trave são obtidas através das amostras de θ_P e θ_T, que são aplicadas em um filtro digital passa alta, que tem a função de estimar as derivadas de θ_P e θ_T.

Para continuar o estudo, iremos inicialmente manipular as equações, de forma a isolar os termos $\ddot{\theta}_T$ e $\ddot{\theta}_P$ como uma função de θ_T, θ_P, $\dot{\theta}_T$ e $\dot{\theta}_P$. Substituindo os seguintes termos, podemos ter uma forma menos complicada de se escrever as equações (e com isso evitar alguns erros):

$$\begin{aligned} a &= m_p L_T^2 + J_b \\ b(t) &= m_p r_p L_T \cos(\theta_P) \\ c(t) &= m_p r_p \sin(\theta_p) \\ d &= m_p r_p^2 \end{aligned} \quad (5\text{-}19)$$

As equações de (5-18) se tornam:

$$a\ddot{\theta}_T + b(t)\ddot{\theta}_P - c(t)L_T\dot{\theta}_P^2 = \tau \qquad (5\text{-}20)$$

$$b(t)\ddot{\theta}_T - c(t)L_T\dot{\theta}_P\dot{\theta}_T + d\ddot{\theta}_P - c(t)g = 0 \qquad (5\text{-}21)$$

Reordenando (5-21) de duas formas diferentes, temos que:

$$\ddot{\theta}_T = \frac{c(t)}{b(t)}(L_T\dot{\theta}_P\dot{\theta}_T + g) - \frac{d}{b(t)}\ddot{\theta}_P \qquad (5\text{-}22)$$

$$\ddot{\theta}_P = \frac{c(t)}{d}(L_T\dot{\theta}_P\dot{\theta}_T + g) - \frac{b(t)}{d}\ddot{\theta}_T \qquad (5\text{-}23)$$

Substituindo (5-23) em (5-20) e (5-22) em (5-21) e reordenando os termos, teremos finalmente uma forma de $\ddot{\theta}_T = f(\theta_T, \theta_P, \dot{\theta}_T, \dot{\theta}_P, t)$ e $\ddot{\theta}_P = g(\theta_T, \theta_P, \dot{\theta}_T, \dot{\theta}_P, t)$ que é apropriada para a resolução numérica:

$$\ddot{\theta}_T = \frac{1}{(ad - b(t)^2)}\left(d\tau - b(t)c(t)(L_T\dot{\theta}_P\dot{\theta}_T + g) + dc(t)L_T\dot{\theta}_P^2\right) \qquad (5\text{-}23)$$

$$\ddot{\theta}_P = \frac{1}{(ad - b(t)^2)}\left(-b(t)\tau + ac(t)(L_T\dot{\theta}_P\dot{\theta}_T + g) - b(t)c(t)L_T\dot{\theta}_P^2\right) \qquad (5\text{-}24)$$

Deve-se lembrar que os termos b(t) e c(t) em (5-19) não são constantes e sim funções sin() ou cos() de $\theta_P(t)$.

No simulador utilizamos as equações (5-23) e (5-24) para simular o sistema.

A partir da linearização das equações (5-23) e (5-24) em torno do ponto de equilíbrio instável, o sistema linear pode ser descrito por uma representação no espaço de estados, na forma $\dot{\mathbf{x}} = \mathbf{Ax} + \mathbf{Bu}$ descrita abaixo.

$$\begin{bmatrix} \dot{\theta}_T \\ \dot{\theta}_P \\ \ddot{\theta}_T \\ \ddot{\theta}_P \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & -\dfrac{m_p L_T g}{J_b} & 0 & 0 \\ 0 & \dfrac{g(m_p L_T^2 + J_b)}{r_p J_b} & 0 & 0 \end{bmatrix} \begin{bmatrix} \theta_T \\ \theta_P \\ \dot{\theta}_T \\ \dot{\theta}_P \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \dfrac{1}{J_b} \\ -\dfrac{L_T}{J_b r_p} \end{bmatrix} \tau \qquad (5\text{-}25)$$

As equações de (5-25) precisam de uma última mudança para poderem ser utilizadas para o projeto de controle: a variável de entrada do sistema será a tensão u(t). A equação abaixo descreve a relação entre o torque $\tau(t)$ e o sinal de entrada u(t) é:

$$\tau(t) = u(t)\frac{K_m K_g}{R} - \dot{\theta}_T(t)\frac{K_m^2 K_g^2}{R} \qquad (5\text{-}26)$$

Substituindo $\tau(t)$ pelo lado direito da eq. (5-26), a equação no espaço de estados torna-se:

$$\begin{bmatrix} \dot{\theta}_T \\ \dot{\theta}_P \\ \ddot{\theta}_T \\ \ddot{\theta}_P \end{bmatrix} = \underbrace{\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & -\frac{m_p L_T g}{J_b} & -\frac{K_m^2 K_g^2}{J_b R} & 0 \\ 0 & \frac{g(m_p L_T^2 + J_b)}{r_p J_b} & \frac{K_m^2 K_g^2 L_T}{J_b r_p R} & 0 \end{bmatrix}}_{\textbf{Matriz A}} \begin{bmatrix} \theta_T \\ \theta_P \\ \dot{\theta}_T \\ \dot{\theta}_P \end{bmatrix} + \underbrace{\begin{bmatrix} 0 \\ 0 \\ \frac{K_m K_g}{J_b R} \\ -\frac{L_T K_m K_g}{J_b r_p R} \end{bmatrix}}_{\textbf{Matriz B}} u \quad (5\text{-}27)$$

O objetivo de controle nesse estudo de caso é estabilizar o sistema descrito por (5-27), de forma que o pêndulo se mantenha na posição vertical superior. Vamos apresentar, a seguir, algumas dificuldades nesse estudo de caso que deverão ser avaliadas no projeto de controle.

A matriz **A** da equação (5-27) possui autovalores [0, -27.6568, 7.7901,-5.3309], o que significa que um termo do tipo $e^{7.7901t}$ vai aparecer na solução das equações de estado e para qualquer condição inicial próxima ao ponto de equilíbrio instável, o sistema vai divergir e deixar a zona linear em que a aproximação linear é válida.

Em relação ao atraso de comunicação, assume-se que este é variante no tempo e que temos algumas informações sobre suas estatísticas em relação a um cenário bastante específico, descrito na seção 5.3.

Assume-se também que todas as variáveis de estado estão disponíveis para medição e livres de ruído.

5.2.2 –Controle Contínuo no Tempo

Nessa seção vamos apresentar o controle estabilizante, o controle de Swing Up e o modo de controle.

- Controle estabilizante

O controle estabilizante utiliza a medida $\theta_{Psup}(t)$, ver tabela 5.3, para calcular o sinal de controle u(t).

Esse controle deve estabilizar o pêndulo na sua posição invertida, se o modelo linear for considerado válido pelo modo de controle. A abordagem utilizada em [40] é resolver o problema Regulador Linear Quadrático e obter o vetor de ganho **K**, que fornece o sinal de controle ótimo u = **-Kx**, da mesma forma como foi descrito na seção 5.1.2, pelas equações (5-12) a (5-14). As matrizes Q e R foram definidas em [40] como:

$$\mathbf{Q} = \begin{bmatrix} 0.25 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \mathbf{R} = 0.05 \qquad (5\text{-}26)$$

Isso resulta em uma função de custo J, apresentada abaixo, em que a velocidade angular da trave $\dot{\theta}_T$ não tem custo algum, e penaliza-se a variável $\theta_{Psup}(t)$.

$$J = \int_{t=0}^{\infty} (0.25)\ \theta_T{}^2(t) + (4)\theta_{Psup}{}^2(t) + (1)\dot{\theta}_T{}^2(t) + (0.05)\,T(t)\ \ dt \quad \textbf{(5-27)}$$

O vetor de ganho **K** obtido com a solução desse problema de minimização da função de custo quadrática é:

$$\mathbf{K} = \begin{bmatrix} -2.2361 & -31.2532 & -1.7227 & -6.2673 \end{bmatrix} \qquad (5\text{-}28)$$

Com o valor **K** e as medidas de $(\theta_T, \theta_P, \dot{\theta}_T, \dot{\theta}_{Psup})$, basta calcular o sinal de controle ótimo para a função de custo quadrática (5-27):

$$u(t) = K \begin{bmatrix} \theta_T \\ \theta_P \\ \dot{\theta}_T \\ \dot{\theta}_P \end{bmatrix}$$

Deve-se observar ainda que o controle estabilizante deve rejeitar pequenas perturbações que afastem o pêndulo do ponto de equilíbrio instável, no entanto o "modo de controle" é projetado para comutar do controle de estabilizante para o de Swing Up se o ângulo $\theta_{Psup}(t)$ exceder o valor de 1 ou 2 graus, entre outros.

- Controle de Swing Up

O controle de Swing Up utiliza a medida de $\theta_{Pinf}(t)$, como descrito na tabela 5.3.

O controle de Swing Up deve oscilar o pêndulo que está na posição para baixo, e trazê-lo para a posição invertida para que assim o controle estabilizante possa ser aplicado. Em [40], a maneira de se obter esse objetivo é baseado no controle linear da posição da trave (braço) do pêndulo. Utilizando o modelo linear e estático do motor DC que está conectado ao braço do pêndulo, propõe-se que a variável de estado $\theta_T(t)$ deve rastrear um sinal de referência $\theta_D(t)$, que é dado pela equação abaixo:

$$\theta_D(t) = P\ \theta_{Pinf}(t), \qquad \text{e} \qquad -8° \le \theta_D(t) \le 8°$$

O modelo linear e estático do motor DC, como descrito em [40], é representado na figura 5.19.

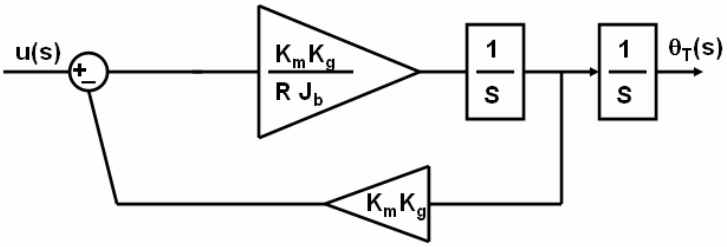


Figura 5.19 – Modelo linear e estático do motor DC.

O sistema de controle linear proposto é ilustrado na figura 5.20:

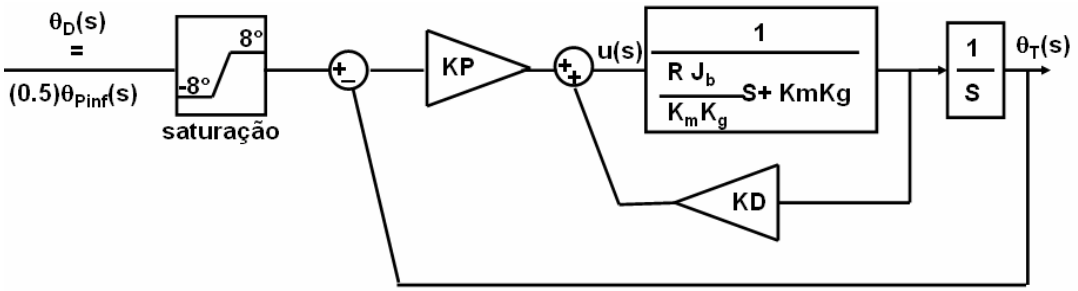


Figura 5.20 – Diagrama de blocos do controle linear para que $\theta_T(t)$ rastreie o sinal de referência $\theta_D(t)$

O projeto desse algoritmo de SwingUp consiste em escolher os valores de KP e KD da figura 5.19 que permitam ao sistema rastrear o sinal de referência $\theta_D(t)$, que é um sinal proporcional ao ângulo do pêndulo. A função de transferência a malha fechada do sistema da figura 5.19, já com os valores da tabela 5.4, é:

$$\frac{47KP}{s^2 + (25 - 47KD)s + 47KP} = \frac{47KP}{s^2 + (2\varsigma w_o)s + \omega_o^2} \qquad (5\text{-}29)$$

ω_o e ζ são respectivamente a freqüência natural e o coeficiente de amortecimento do sistema a malha fechada. Para calcular KP e KD, é utilizado como parâmetro o valor da freqüência natural do pêndulo para pequenas oscilações:

$$\omega_p = \sqrt{\frac{g}{r_p}} \qquad (5\text{-}30)$$

Vamos calcular KP e KD para que o coeficiente de amortecimento e a freqüência natural do sistema a malha fechada sejam iguais a $\zeta = 0.707$ e $\omega_o = 6*\omega_p$. Na verdade o valor da freqüência natural ω_o pode variar entre $5*\omega_p$ e $7*\omega_p$.

A observação a ser feita é que este projeto é todo baseado no modelo linear, para pequenas oscilações, no entanto o que se deseja é criar grandes oscilações para instabilizar o pêndulo, logo o modelo linear perde a sua validade na maior parte da missão de controle de SwingUp. Os autores de [40] se desculpam pela falta de rigor nessa análise apresentada, mas garantem que o algoritmo funciona.

- Modo de Controle

O "modo de controle" utiliza os valores de θ_{Pfull}, descrito na tabela 5.3.

O modo de controle decide quando o controle de SwingUp ou o controle estabilizante são aplicados, a partir de critérios intuitivos como verificar se o pêndulo está perto do ponto de equilíbrio, se a velocidade $\dot{\theta}_P$ e a posição θ_T são pequenos o suficiente e se o pêndulo está para cima. Abaixo vamos apresentar as condições para o modo de controle mudar de SwingUp para estabilizante:

Condições para Comutar de controle de SwingUp para Estabilizante:

$$|\theta_{Psup}| < 15 \text{ graus}$$

$$|\theta_T| < 25 \text{ graus}$$

$$|\dot{\theta}_P| < 200 \text{ graus/s}$$

$$\cos(\theta_{Pfull}) < 0 \qquad (5\text{-}31)$$

Condições para permanecer com o controle Estabilizante:

$$|\theta_{Psup}| < 2 \text{ graus}$$

$$|\theta_T| < 25 \text{ graus}$$

$$|\dot{\theta}_P| < 200 \text{ graus/s}$$

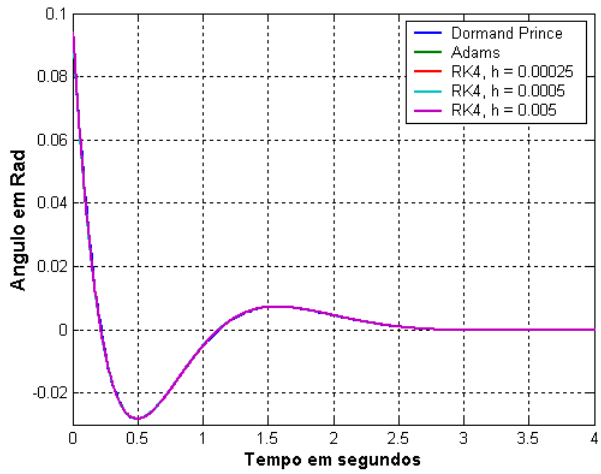
$$\cos(\theta_{Pfull}) < 0 \qquad (5\text{-}32)$$

Quando o modo de controle comuta de Swing Up para o controle Estabilizante, a mudança de limiar de $|\theta_{Psup}|$ de 15 graus para 2 graus deve ocorrer com um certo atraso para dar tempo do sistema ser estabilizado. Para isso é utilizado um bloco de atraso, descrito pela função de transferência (0.2/(s+0.2)), que é um dos únicos elementos do "modo de controle" que devem ser discretizado para implementação no controlador digital.

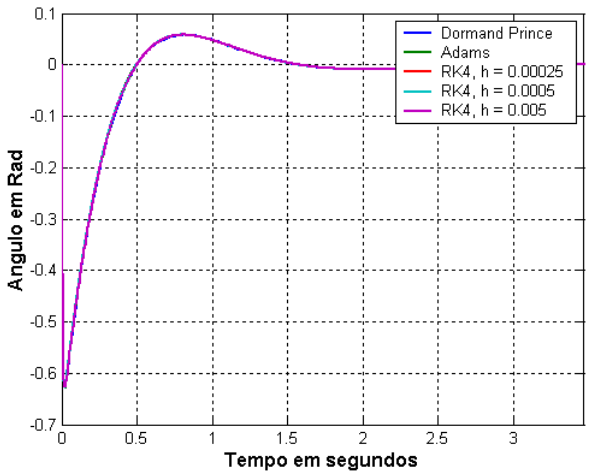
5.2.3 – Simulação numérica do pêndulo do 2º Estudo de Caso

Vamos simular o sistema do pêndulo para o caso do controle estabilizante, para as condições iniciais $\theta_T = 0, \theta_P = 5.4^o, \dot{\theta}_T = 0, \dot{\theta}_P = 0$, e o caso do controle de SwingUp, para a condição inicial $\theta_T = 0, \theta_P = 185.4^o, \dot{\theta}_T = 0, \dot{\theta}_P = 0$. Será utilizado o método Clássico de Runge Kutta de 4ª ordem, para h = 0.005, 0.0005 e 0.00025, comparando com os métodos de passo de tempo variável de Dormand Prince e Adams (de passo corretor e preditor). O "modo de controle" não será utilizado nas simulações em nenhum dos dois casos;

A figura 5.21 apresenta o primeiro caso, com a aplicação do controle estabilizante. Os resultados indicam que qualquer um dos métodos apresentam bons resultados, para essa simulação de curta duração, em que se converge para um ponto de equilíbrio, onde as velocidades tendem a zero.



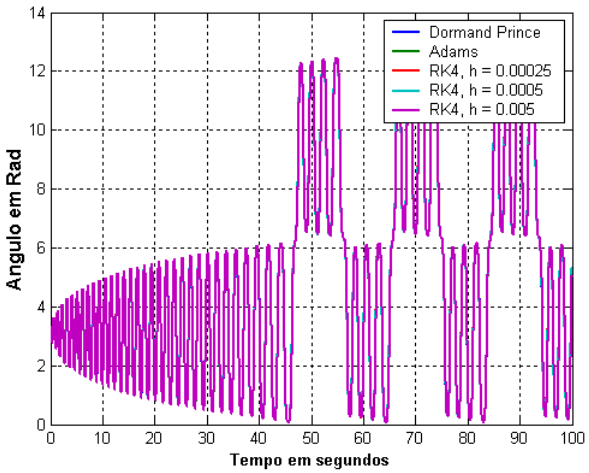
(a) $\theta_P(t)$



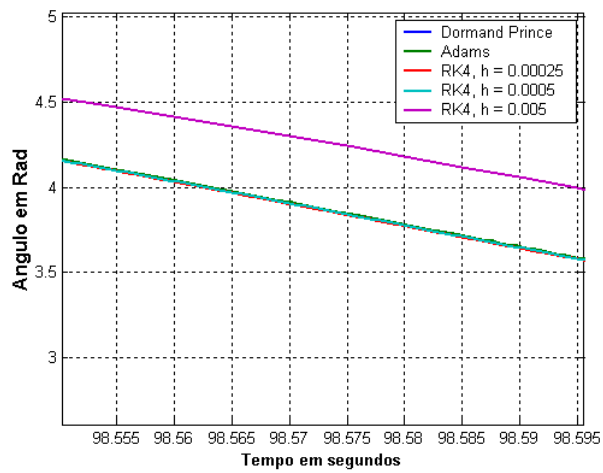
(b) $\dot{\theta}_P(t)$

Figura 5.21 – Controle Estabilizante, para condições iniciais $\theta_T = 0, \theta_P = 5.4^o, \dot{\theta}_T = 0, \dot{\theta}_P = 0$

Para o segundo caso, vamos simular o controle de SwingUp, sem o "modo de cotrole", ou seja, o pêndulo vai ficar oscilando sem nunca ser capturado no seu ponto de equilíbrio instável pelo controle estabilizante. Vamos simular para 100 segundos, e observar a concordância da resposta no tempo para os vários métodos numéricos. As figuras 5.22 e 5.23 apresentam os resultados.

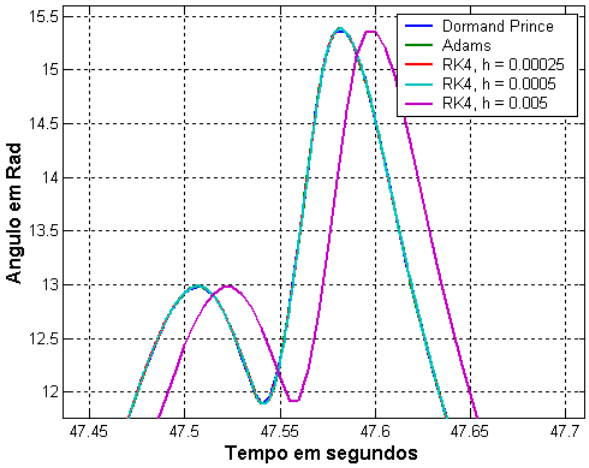


(a) $\theta_P(t)$ – oscilações forçadas pelo algoritmo de SwingUp.

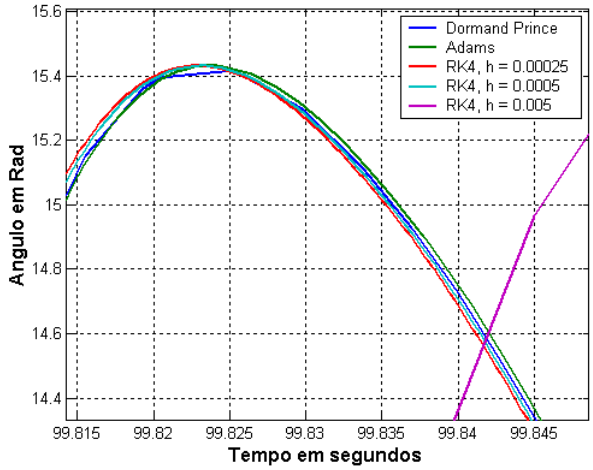


(b) Ampliando o gráfico de $\theta_P(t)$ num intervalo próximo dos 100 segundos

Figura 5.22 –Para h = 0.005, o método de Runge Kutta de 4ª ordem está bastante defasado em relação aos outros métodos depois de 100 segundos.



(a) $\dot{\theta}_p(t)$ próximo de 50 segundos



(b) $\dot{\theta}_p(t)$ próximo de 100 segundos

Figura 5.23 – Velocidade angular do pêndulo. Em 100 segundos, todos os métodos de Runge Kutta Clássico, estão defasados em relação aos métodos de passo variável.

Pelas figuras 5.21, 5.22 e 5.23, pode-se observar que os métodos de passo fixo podem ser utilizados sem grandes problemas em "missões" de curta duração, no qual o sistema converge para um ponto de equilíbrio. No entanto, em sistemas oscilatórios, os erros acumulados ao longo do tempo se tornam um problema que não pode ser tratado uma vez que a simulação já tenha começado, pois não é possível alterar o passo de tempo h dinamicamente. É claro que se o "modo de controle" fosse utilizado, o sistema teria convergido para o ponto de equilíbrio instável e os erros numéricos seriam cancelados.

A seguir é apresentado o código simplificado do módulo RTLinux/Free que simula pêndulo.

/*Código do módulo do RTLinux simplificado*/

```
pthread_t thread;
void * pendulo(void);

void * pendulo(void)
{
pthread_self()->uses_fp = 0; /* to force save/restore */
pthread_setfp_np (pthread_self(), 1);
struct sched_param p;/*struct necessario para alterar a prioridade do thread*/
p . sched_priority = 99;
pthread_setschedparam (pthread_self(), SCHED_FIFO, &p);
double t,u,h,pi,tau,mp,LP,LT,rp,IT,g,a1,a2,d,Rm,Km,Kg;
double cp,sp,a,b,ia,dpt,dp2,G1;
pi = 3.14159265358979;
t=0;              /*tempo de simulação*/
h=0.0005;          /*passo de tempo fixo*/
u=0;             /*sinal de controle(tensão)*/
tau=0;            /* torque */
Rm = 2.6;
Km = 0.00767;
Kg = 14*5;
```

```
mp = 0.14;
LP = 0.43;
LT = 0.21;
rp = LP/2;
IT = 0.0044;
g  = 9.8;
u= 0; /*sinal de controle (tensão)*/
a1 = mp*(LT*LT) + IT;
a2 = mp*(LT*LT);
d  = mp*LT;
int iterador=0;
double thetaT,thetaP,dthetaT,dthetaP,x[4],h_tmp,dx[4][4];
/*Condicoes Iniciais*/
thetaT   =0;
thetaP   =pi*0.03;
dthetaT  =0;
dthetaP  =0;
int cont=0;
int it =0;

pthread_make_periodic_np (pthread_self(), gethrtime(), 500000); /*0.5 ms*/
while(1)
{
       pthread_wait_np ();
       /*Metodo de Runge Kutta para resolucao de EDOs */
       /*y[k+1] = y[k] + (1/6)*(k1+ 2 k2 +2 k3 + k4) */
       x[0]     = thetaT;
       x[1]     = thetaP;
       x[2]     = dthetaT;
       x[3]     = dthetaP;
       it=0;
       for(;it<4;it++)
       {
              if(it < 2)/*nao influencia no calculo de k1*/
              {
                     h_tmp=h/2; /*para o calculo de k2 e k3*/
              }
              else
              {
                     h_tmp=h;/*para o calculo de k4*/
              }

              cp       = cos(x[1]);/*cos(thetaP)*/
              sp       = sin(x[1]);/*sin(thetaP)*/
              a        = a1 - a2*cp*cp;
              b        = sp*cp;
              ia       = 1/a;
              dpt      = x[2]*x[3]; /*dthetaT * dthetaP*/
              dp2      = x[3]*x[3];/*dthetaP^2*/
              G1       = LT*dpt+g;
```

```
            tau      = u*(Km*Kg/Rm) - ((Km*Km)*(Kg*Kg)*x[2])/Rm;
            dx[it][0] = x[2];
            dx[it][1] = x[3];
            dx[it][2]=(ia)*(tau - d *b *G1+ d * rp * sp *dp2 );
            dx[it][3]=(ia/rp)*(-(LT*cp)*tau + a1*sp*G1- d*b*LT * rp *dp2);
            if(it < 3)/*so e necessario para calcular k2 e k3*/
            {
                  x[0]  = thetaT  + h_tmp * dx[it][0];
                  x[1]  = thetaP  + h_tmp * dx[it][1];
                  x[2]  = dthetaT + h_tmp * dx[it][2];
                  x[3]  = dthetaP + h_tmp * dx[it][3];
            }
      }
      /*atualizando os estados*/
      thetaT = thetaT +(h/6)*(dx[0][0]+2*dx[1][0]+2*dx[2][0]+dx[3][0]);
      thetaP = thetaP +(h/6)*(dx[0][1]+2*dx[1][1]+2*dx[2][1]+ dx[3][1]);
      dthetaT= dthetaT+(h/6)*(dx[0][2]+2*dx[1][2]+2*dx[2][2]+dx[3][2]);
      dthetaP= dthetaP+(h/6)*(dx[0][3]+2*dx[1][3]+2*dx[2][3]+dx[3][3]);
      t = t+h;
}

return 0;
}/*Fim da thread pendulo*/
```

5.3 – Discretizando as abordagens de controle

5.3.1 –Controle Digital

Até a seção 5.2 somente foram descritas as abordagens de controle contínuas no tempo, no entanto as abordagens de controle devem ser discretizadas, para poderem ser implementadas e executadas em sistemas computacionais digitais. Um sinal de uma variável física, como posição, velocidade ou aceleração, entre outras, é convertida em um sinal elétricos por transdutores, que será amostrada e então convertida para um sinal digital, que é uma versão quantizada e codificada dessa amostra analógica. Esses sinais digitais são utilizados pelo controlador digital para o cálculo do sinal de controle digital, que é então decodificado por um conversor Digital-Analógico. Um circuito de Hold extrapola o valor dessa amostra digital no sinal analógico até o próximo valor do sinal digital. Um circuito de Hold simples é o de ordem zero, que mantém um valor constante do sinal analógico, até que um novo sinal digital seja atualizado no conversor D-A. A figura 5.24 ilustra um sistema de controle digital de uma entrada e um saída, em que um sinal de tensão correspondente a variável física X(t) é amostrada e digitalizada. Na saído do controlador Digital, o conversor D/A é seguido de um circuito de Hold de ordem zero.

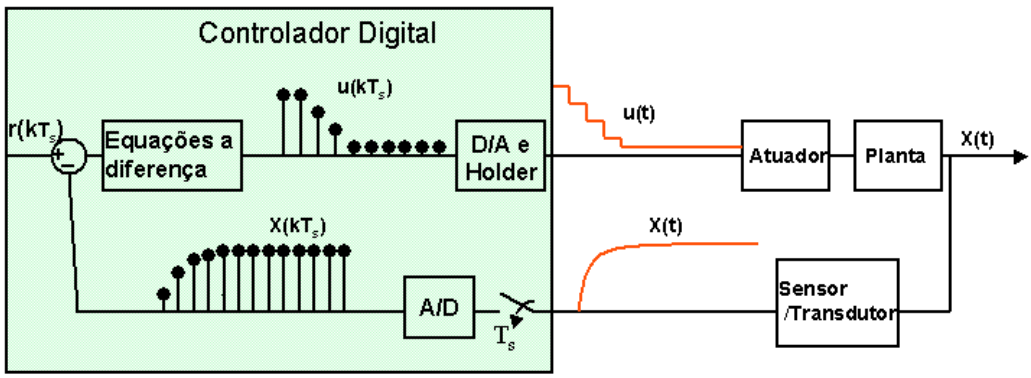


Figura 5.24- Exemplo de sistema de controle digital

Nessa seção vamos apresentar rapidamente o tipo de circuito de Hold que vamos considerar e qual período de amostragem considerar no projeto.

Nesse projeto somente o circuito de Hold de ordem zero é considerado, que vamos chamar de ZOH (*Zero order Hold*). A transformada de Laplace do Hold de ordem zero é dada abaixo:

$$\text{ZOH}(s) = \frac{1 - e^{-sT_s}}{s} \qquad (5\text{-}33)$$

Em um sistema de controle digital, o ZOH altera a dinâmica do sistema, introduzindo um atraso de tempo de $T_s/2$ no sinal de controle, onde T_s é o período de amostragem, e introduzindo também sinais indesejáveis de alta freqüência. A figura 5.25 ilustra o efeito do ZOH.

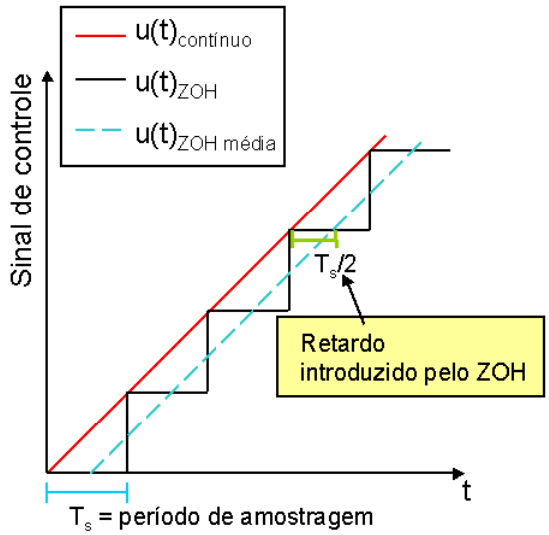


figura 5.25 – Sinal de controle: depois da amostragem, a extrapolação feita pelo ZOH.

5.3.2 – Equivalentes discretos

Para calcular os equivalentes discretos das leis de controle contínuas, vamos apresentar um abordagem baseada em métodos numéricos como Euler Forward e Euler Backward, e também uma forma de representar o sistema no espaço de estados discreto.

- Aproximação por método numéricos

A aproximação por métodos numéricos de integração se baseia na aproximação de equações diferenciais por equações a diferenças, sob a condição de que o período de amostragem é suficiente pequeno para negligenciarmos o erros gerados por essa abordagem. Dois métodos numéricos simples são o método de Forward Euler e o método de Backward Euler, descritos abaixo:

$$\dot{x} \cong \frac{x(kT+T)-x(kT)}{T} \Rightarrow s \cong \frac{z-1}{T} \qquad \text{Forward Euler} \qquad (5\text{-}34)$$

$$\dot{x} \cong \frac{x(kT)-x(kT-T)}{T} \Rightarrow s \cong \frac{z-1}{zT} \qquad \text{Backward Euler} \qquad (5\text{-}35)$$

Dessa forma, um equivalente discreto do compensador contínuo no tempo pode ser obtido utilizando-se uma das aproximações acima.

- Margem ao atraso para sistema discretos

Uma maneira de calcular a margem ao atraso de um sistema com uma planta com função de transferência G(s), e controlador digital com lei de controle C(z), é encontrar a função de transferência da malha L(z) = C(z)G'(z), sendo G'(z) o equivalente discreto da função de G(s) com o ZOH(s), e está descrita na equação abaixo.

$$G'(z) = Z\left[\frac{1-e^{-sT}}{s}G(s)\right] \qquad (5\text{-}36)$$

Com L(z) pode-se obter através do gráfico de Bode a margem de fase e a freqüência de crossover digital ω_c. A margem de atraso pode ser expressa como:

$$\frac{\text{Margem de fase (em rad)}}{\omega_c\,(\text{rad/amostra}) \times f_s\,(\text{amostras/s})}, \text{ onde } f_s \text{ é a freqüência de amostragem} \qquad (5\text{-}37)$$

- Representação no espaço de Estados discreta

Uma representação de um sistema no espaço de estados em sistema contínuos no tempo é dada pela forma abaixo:

$$\dot{\mathbf{X}} = \mathbf{AX} + \mathbf{Bu}$$

$$\mathbf{Y} = \mathbf{CX} + \mathbf{Du}$$

A solução geral para estas equações é diferenciais é da forma:

$$\mathbf{X}(t) = e^{\mathbf{A}(t-t_0)}\mathbf{X}(t_0) + \int_{t_0}^{t} e^{\mathbf{A}(t-\tau)}\mathbf{B}\,u(\tau)\,d\tau \qquad (5\text{-}38)$$

Seja T o período de amostragem. A partir da solução geral, iremos obter a equação a diferenças para um período de amostragem, substituindo t por kT+T e t_0 por kT.

$$\mathbf{X}(\,kT+T\,)=e^{\mathbf{A}T}\mathbf{X}(\,kT\,)+\int_{kT}^{kT+T}e^{\mathbf{A}(T-\tau)}\mathbf{B}\,u(\,\tau\,)\,d\tau \quad (5\text{-}39)$$

Se considerarmos que estamos utilizando um ZOH, temos que o valor de u(τ) não varia durante para kT ≤ τ ≤.kT + T.

$$\mathbf{X}(\,kT+T\,)=e^{\mathbf{A}T}\mathbf{X}(\,kT\,)+u(\,kT\,)\int_{kT}^{kT+T}e^{\mathbf{A}(T-\tau)}\mathbf{B}\,d\tau$$

Para resolver a equação acima, será utilizado o resultado de [43], no qual podemos calcular:

$$F=e^{\mathbf{A}T}=\mathbf{I}+\mathbf{A}T\boldsymbol{\psi} \quad (5.40)$$

$$G=\int_{kT}^{kT+T}e^{\mathbf{A}(T-\tau)}\mathbf{B}\,d\tau=\boldsymbol{\psi}\mathbf{B}T \quad (5.41)$$

$$\boldsymbol{\psi}=\mathbf{I}+\frac{\mathbf{A}T}{2!}+\frac{\mathbf{A}^2T^2}{3!}+\ldots \quad (5.42)$$

Em [43] é descrita uma forma simples para calcular y

$$\boldsymbol{\psi}\approx\mathbf{I}+\frac{\mathbf{A}T}{2}\left(\mathbf{I}+\frac{\mathbf{A}T}{3}\left(\ldots\frac{\mathbf{A}T}{N-1}\left(1+\frac{\mathbf{A}T}{N}\right)\right)\right) \quad (5.43)$$

Com isso obtemos uma representação no espaço de estados na forma:

$$\mathbf{X}(kT+T)=\boldsymbol{\Phi}\mathbf{X}(kT)+\boldsymbol{\Gamma}u(kT) \quad (5.44)$$

$$\mathbf{Y}(kT)=\mathbf{C}\mathbf{X}(kT)+\mathbf{D}u(kT)$$

- Regulador Linear Quadrático

O problema do regulador quadrático para uma planta discreta é o problema de minimização da função de custo quadrática:

$$\mathbf{J}=\sum_{\mathbf{k}=1}^{\infty}\mathbf{x}(\mathbf{k})^{\mathbf{T}}\mathbf{Q}\mathbf{x}(\mathbf{k})+\mathbf{u}(\mathbf{k})^{\mathbf{T}}\mathbf{R}\mathbf{u}(\mathbf{k}) \quad (5\text{-}45)$$

Uma forma de resolver esse problema é obter a solução **S** da equação de Riccati discreta no tempo, para então calcular **K**, a matriz de ganho do controle ótimo **u**(kT) = **Kx**(k):

$$\mathbf{S}=\boldsymbol{\Phi}^{T}[\mathbf{S}-\mathbf{S}\,\boldsymbol{\Gamma}\,\mathbf{R}^{-1}\boldsymbol{\Gamma}^{T}\,\mathbf{S}]\boldsymbol{\Phi}+\mathbf{Q} \quad (5\text{-}46)$$

E com **S** calcular o valor de K:

$$\mathbf{K}=(\boldsymbol{\Gamma}^{\mathbf{T}}\mathbf{S}\boldsymbol{\Gamma}+\mathbf{R})^{-1}(\boldsymbol{\Gamma}^{T}\mathbf{S}\boldsymbol{\Phi}) \quad (5\text{-}47)$$

No octave ou matlab, a função dlqr(F ,G,**Q**,**R**) retorna a solução **S** de (5-46) e o ganho **K**.

- Cálculo do equivalente discreto de $\overline{N}$, descrito nas equações (5-12) e (5-13):

Basta utilizar a seguinte fórmula:

$$\begin{bmatrix}\mathbf{N_x}\\ N_u\end{bmatrix}=\begin{bmatrix}\boldsymbol{\Phi}-\mathbf{I} & \boldsymbol{\Gamma}\\ \mathbf{C} & \mathbf{D}\end{bmatrix}^{-1}\begin{bmatrix}\mathbf{0}\\ 1\end{bmatrix} \quad (5\text{-}48)$$

$$\overline{N}=N_V+\mathbf{K}\mathbf{N_x} \quad (5\text{-}49)$$

F é uma matriz n x n, descrita em (5-40), G é um matriz n x m, descrita em (5-41), $\mathbf{N_x}$ é um vetor 1 x n e N_u é um escalar. O vetor **0** é um vetor 1 x n, com todos os valores iguais a zero e **K** é o vetor de ganhos calculado em (5-47). $\overline{N}$ é utilizado para multiplicar o sinal de referência r(t).

5.3.2.1 – Discretizando as leis de controle: 1º Estudo de Caso

Para discretizar as leis de controle da seção 5.1.2, será utilizada a aproximação por métodos numéricos de Euler Backward. Assim os equivalentes discretos são apresentados na tabela 5-5. O período de amostragem selecionado foi de T = 0.1 s.

Tabela 5.5 – Representação em equações a diferenças para o compensadores discretizados

Função de transferência Da lei de controle contínua	Representação em equações a diferença	Margem ao atraso para o sistema discreto
FeedForward: $C(s) = \frac{V(s)}{e(s)} = \frac{1}{\text{Ganho DC}} = 4.1$	V(k) = 4.1 e(k)	-
P: $C(s) = \frac{V(s)}{e(s)} = \mathbf{K} = 4$	V(k) = K e(k) = 4 e(k)	∞
PI: $C(s) = \frac{K_p s + K_i}{s} = 1 + \frac{4.5}{s}$	$V(k) = K_p e(k) + V(k-1) + K_I T_s e(k)$ = e(k) + V(k-1)+ 0.45 e(k)	$\frac{1.339\text{rad}}{1.07\text{rad/s}} =$ 1.25 segundos
LQR: V(k)= –0.789 i(t) – 0.556 ω(t) + 5.477 q(t) + 6.2350 r(t)	V(k)= –0.709 i(k) –0.506 ω(k) + 5.06 q(k) + 6.0241 r(k)	1.245 segundos em simulação

A figura 5.26 abaixo ilustra o sinal de controle da versão discreta de $C(s) = \frac{V(s)}{e(s)} = \frac{4.5}{s}$, utilizando os métodos de Euler Forward e Euler Backward. O uso do método de Euler Forward resultou em um sistema com menor margem de fase, menos amortecido e com menor margem ao atraso do que com a abordagem de Euler Backward.

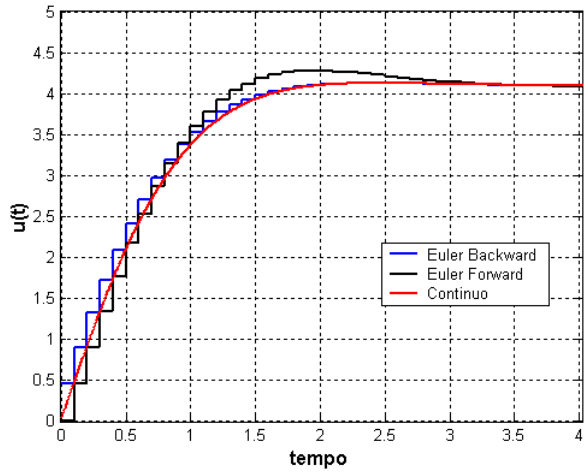


Figura 5.26 –Exemplo do efeito do ZOH no sinal de controle e os dois métodos de integração numéricos aplicados a $C(s) = \frac{V(s)}{e(s)} = \frac{K_i}{s}$

Variando-se o período de amostragem de 10 ms a 400 ms não afeta em muito o desempenho do sistema para C(z) = K = 4 para a entrada ao degrau. Essa faixa de valores de amostragem em que o sistema continua estável favorece o uso de diferentes taxas de transmissões para testar um sistema de comunicação, sem no entanto termos que nos preocupar com a estabilidade do sistema de controle ou a estabilidade numérica.

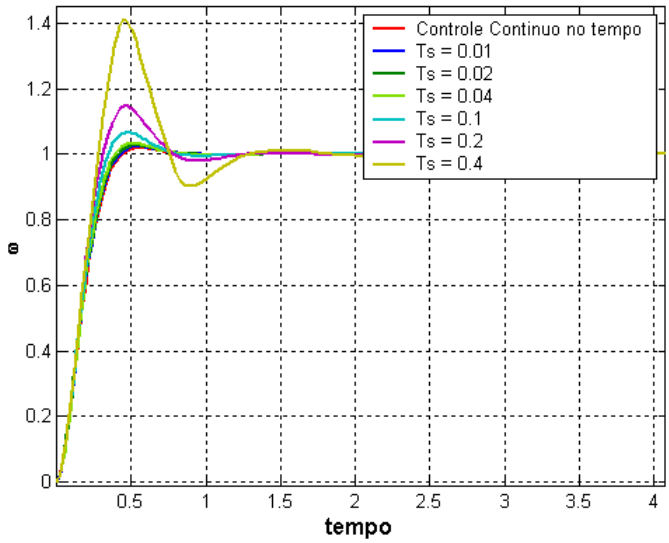
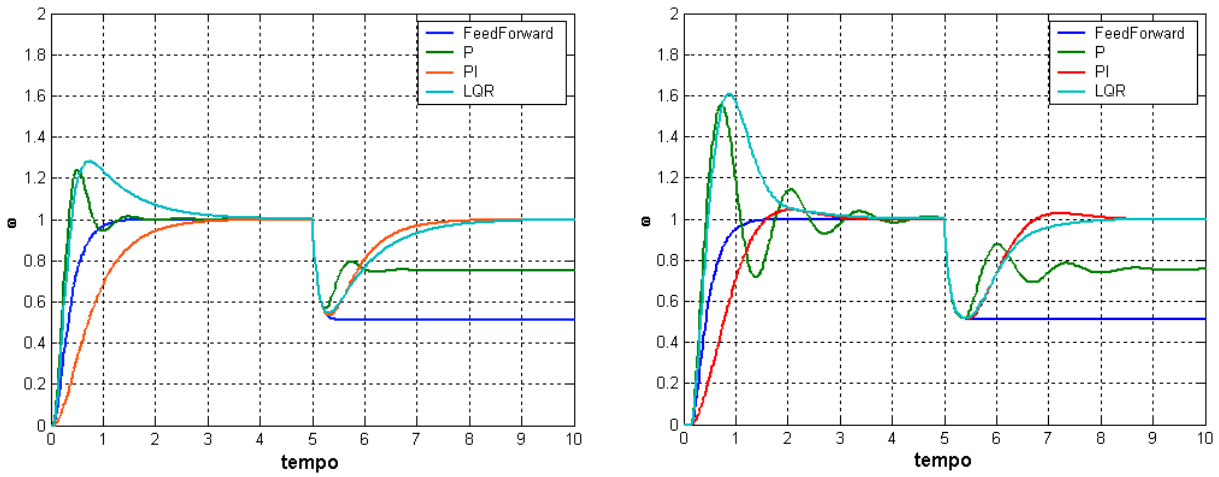


Figura 5.27 – Resultado de $C(z) = \frac{V(z)}{e(z)} = K = 4$, para vários períodos de amostragem.

Como foi descrito na seção 5.2.2, assume-se que o atraso de comunicação será estático e menor que 200 ms. No entanto, a figura 5.26 ilustra como o sistema de controle digital responderia a entrada ao degrau, caso o atraso estático fosse de 300 ms.



(a)Atraso = $\tau_1 + \tau_2 = 100$ ms (b)Atraso = $\tau_1 + \tau_2 = 300$ ms

Figura 5.28 – A saída ω(t), para as várias abordagens de controle, com Ts = 0,1s e atraso de 100 ms e 300 ms.

5.3.2.2 – Discretizando as leis de controle: 2º Estudo de Caso

Para discretizar o "Controle estabilizante" basta obter a representação no espaço de estados que descreve o sistema discreto e utilizar as matrizes Q e R de (5-26) para resolver o problema do Regulador Linear Quadrático e obter o vetor **K**, para obter a lei de controle ótima. O valor de K obtido, para os períodos de amostragem T = 10 ms, 20 ms, 30 ms e 40 ms foi de:

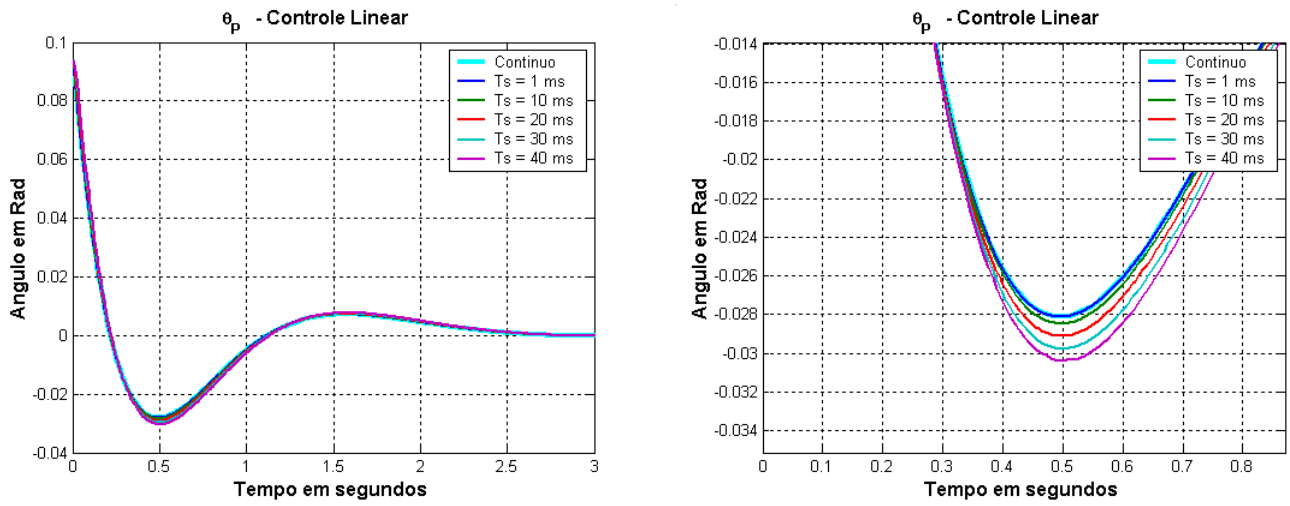
$$T = 40 \text{ ms} \Rightarrow \mathbf{K} = [-0.3607 \quad -8.6042 \quad -0.8489 \quad -1.2970]$$

$$T = 30 \text{ ms} \Rightarrow \mathbf{K} = [-0.4467 \quad -9.6970 \quad -0.9189 \quad -1.5163]$$

$$T = 20 \text{ ms} \Rightarrow \mathbf{K} = [-0.6062 \quad -11.8484 \quad -1.0495 \quad -1.9461]$$

$$T = 10 \text{ ms} \Rightarrow \mathbf{K} = [-0.9880 \quad -17.2081 \quad -1.3633 \quad -3.0133]$$

A figura 5.29 abaixo ilustra a resposta do sistema para vários períodos de amostragem e seus respectivos ganhos **K**, que partem de uma mesma condição inicial, próxima ao ponto de equilíbrio instável, com $\theta_P = 5.4°$ e as outras variáveis de estado iguais a zero. Com essa condição inicial o "modo de controle" já teria comutado o controle para o SwingUp, por isso o bloco "modo de controle" fica desativado nessas simulações.

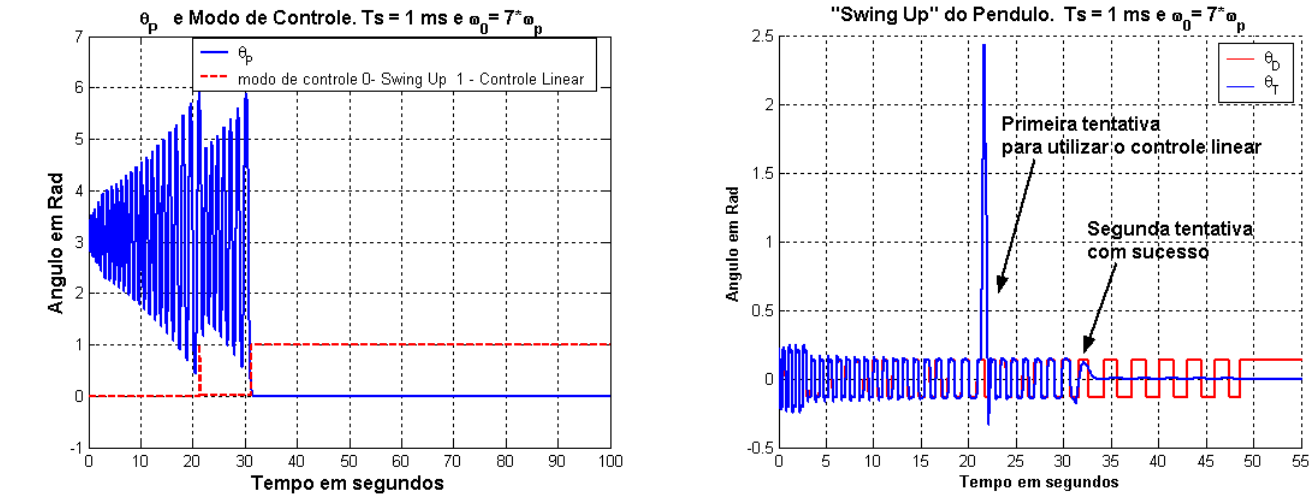


(a) (b)

Figura 5.29 – Controle Estabilizante para diferentes períodos de amostragem.

O bloco "modo de controle" de [40] tem que ter o seu bloco de atraso (0.2/s+0.2) discretizado por um dos métodos numéricos de Euler Forward ou Backward, para obtermos as equações a diferença equivalente.

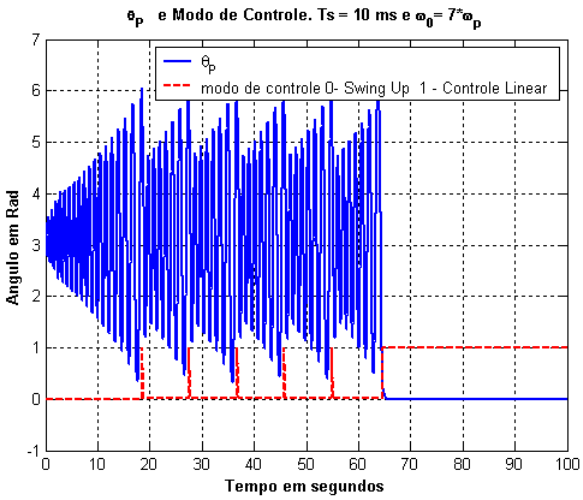
No Controle de SwingUp, a princípio são utilizados os mesmos ganhos KP e KD calculados na seção 5.2.2, mas podemos variar o KP e KD do projeto da seção 5.2, alterando a razão da freqüência natural do sistema a malha fechada entre 5 e 7 ω_o, ω_o a freqüência de oscilação do pêndulo para pequenos sinais descrita em (5-30). Se a oscilação não for suficiente, pode-se aumentar um pouco o ganho KP, para aumentar as amplitude das oscilações. Nos gráficos de 5.30 a 5.33, apresentamos: no gráfico (a) o ângulo $\theta_P(t)$ e a saída do "modo de controle", que é igual a "1" somente quando o controle estabilizante é utilizado; e no gráfico (b) que apresenta o sinal de referência $\theta_D(t)$ e o ângulo da trave $\theta_T(t)$ que rastreia o sinal de $\theta_D(t)$, como descrito no controle de SwingUp da seção 5.2.2.



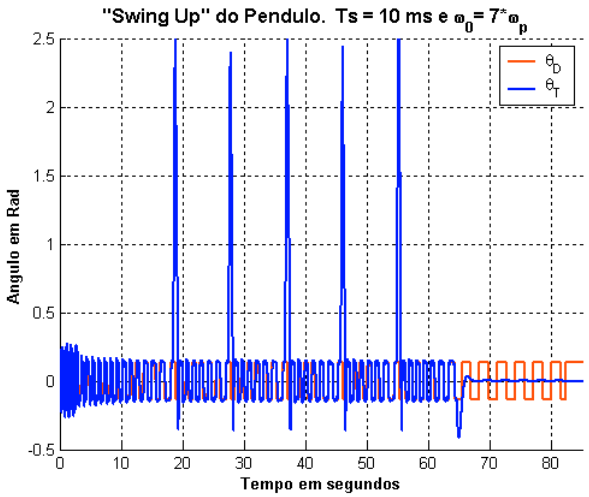
(a) $\theta_P(t)$ em azul, "modo de controle" em vermelho.

(b) $\theta_D(t)$ em vermelho e $\theta_T(t)$ em azul

Figura 5.30- SwingUp do Pêndulo para T = 1 ms. Condição inicial $\theta_T = 0, \theta_P = 185.4^o, \dot{\theta}_T = 0, \dot{\theta}_P = 0$



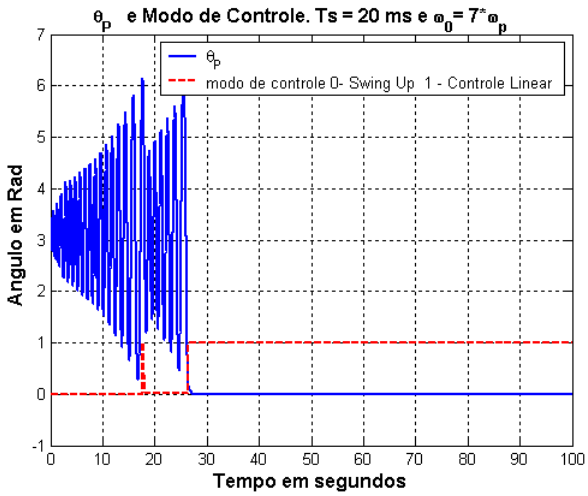
(a) $\theta_P(t)$ em azul, "modo de controle" em vermelho



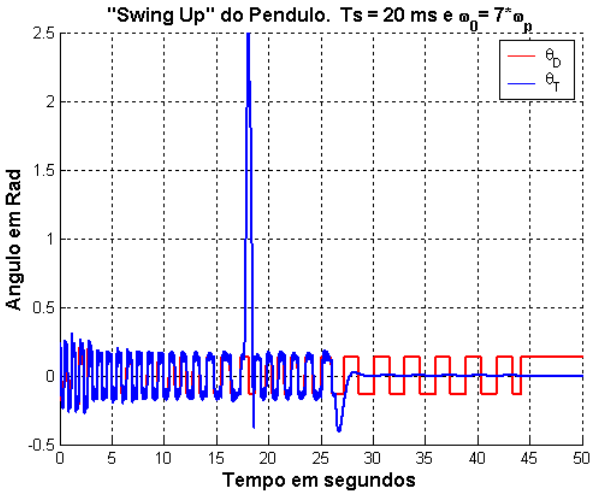
(b) $\theta_D(t)$ em vermelho e $\theta_T(t)$ em azul

Figura 5.31 - SwingUp do Pêndulo para T = 10 ms. Condição inicial

$\theta_T = 0, \theta_P = 185.4^o, \dot{\theta}_T = 0, \dot{\theta}_P = 0$. O "modo de controle" pode ser ajustado para capturar em menos tentativas o pêndulo na posição invertida com o controle estabilizante.



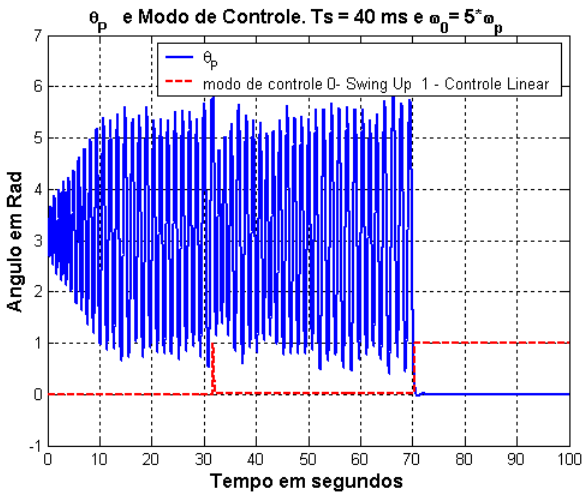
(a) $\theta_P(t)$ em azul, "modo de controle" em vermelho



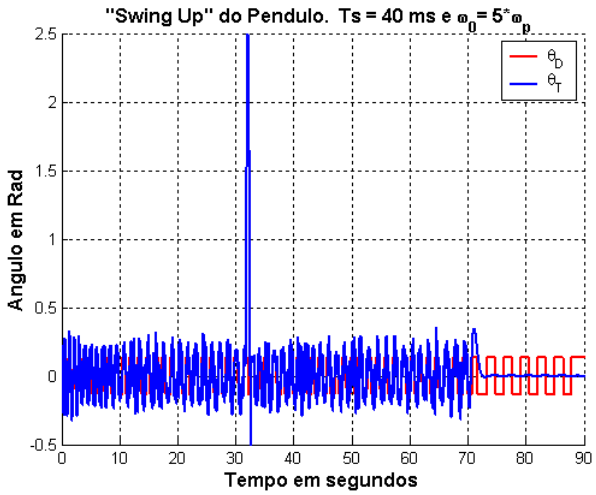
(b) $\theta_D(t)$ em vermelho e $\theta_T(t)$ em azul

Figura 5.32 – Swing Up do Pêndulo para T = 20 ms. Condição inicial

$\theta_T = 0, \theta_P = 185.4^o, \dot{\theta}_T = 0, \dot{\theta}_P = 0$



(a) $\theta_P(t)$ em azul, "modo de controle" em vermelho

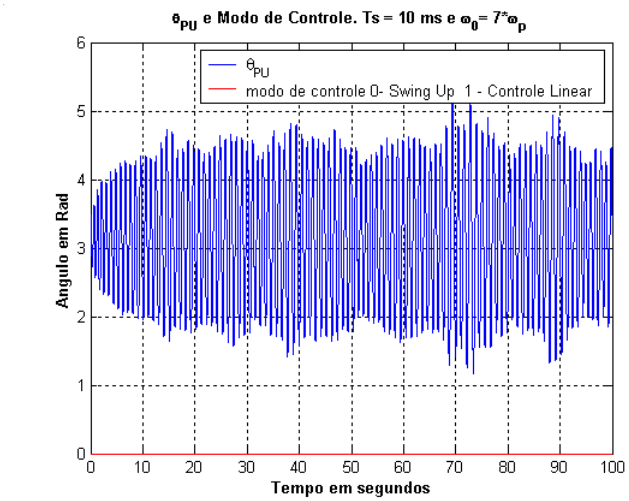


(b) $\theta_D(t)$ em vermelho e $\theta_T(t)$ em azul

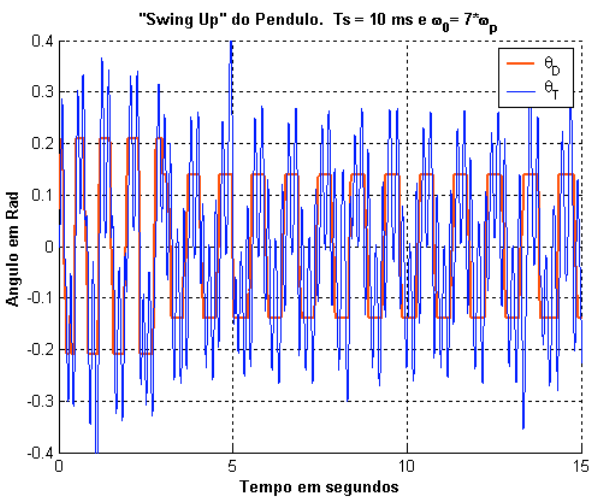
Figura 5.33 - SwingUp do Pêndulo para T = 40 ms. Condição inicial

$$\theta_T = 0, \theta_P = 185.4^o, \dot{\theta}_T = 0, \dot{\theta}_P = 0$$

Na seção 5.2, um problema não abordado foram os efeitos de um atraso variante no tempo em um sistema de controle não linear. Para verificar em simulação os efeitos do atraso variável no tempo no controle de SwingUp e estabilizante, foram feitas várias medições do atraso de comunicação *Bluetooth*, cujos histogramas e caminhos amostrais do atraso de ida e volta estão no capítulo 7, seção 7.2. Vamos utilizar esses dados registrados para gerar a seqüência de atrasos que ilustram a dificuldade em se conseguir o SwingUp do pêndulo sob condições de atraso de comunicação entre 10 ms e 70 ms. As figuras a seguir apresentam o resultado do SwingUp, com atrasos variante no tempo gerados a partir de registros de atrasos reais, para um caso de pouca carga no sistema operacional Linux e um enlace Bluetooth dedicado. Para cada período de amostragem, existe um histograma associado, com diferentes características, como média, pior caso, entre outros, e isso ajuda a entender melhor os efeitos do atraso de comunicação no sistema de controle distribuído, assim como permite justificar a implementação ou não do controle de SwingUp, pois é melhor resolver esse problema com simulações off-line, utilizando métodos numéricos mais sofisticados, do que implementá-lo no sistema proposto com comunicação *Bluetooth*, onde o simulador utiliza métodos numéricos menos estáveis.

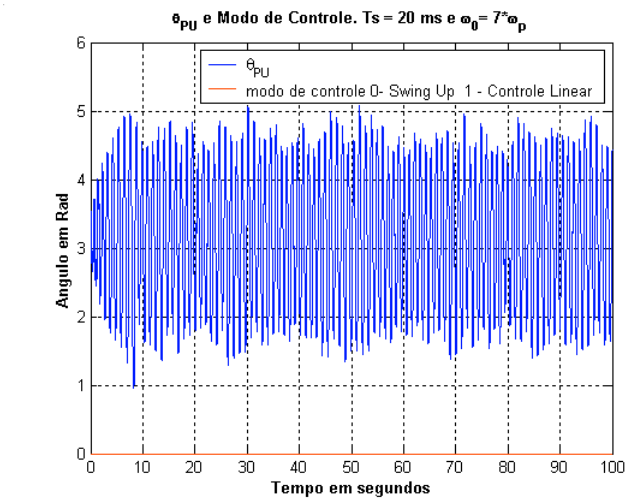


(a) $\theta_P(t)$ em azul, "modo de controle" em vermelho

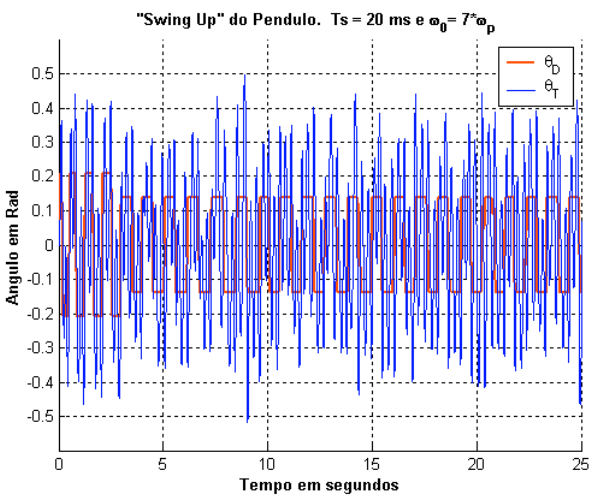


(b) $\theta_D(t)$ em vermelho e $\theta_T(t)$ em azul

Figura 5.34 – Swing Up do Pêndulo para T = 10 ms, com atraso variável descrito na figura 7.37, do cap.7.

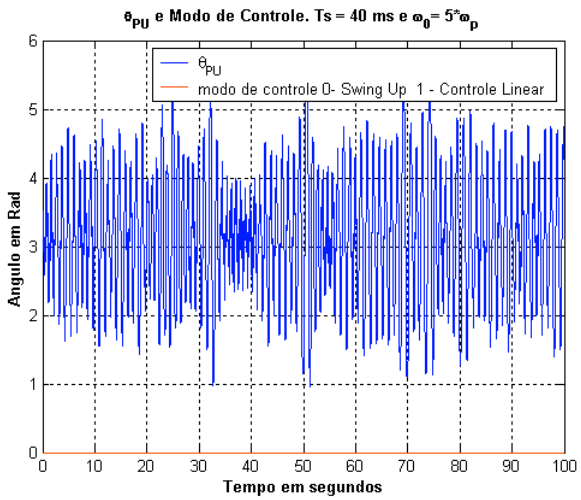


(a) $\theta_P(t)$ em azul, "modo de controle" em vermelho

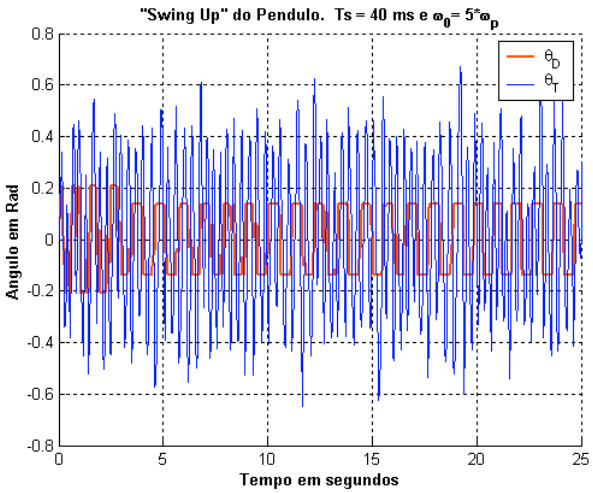


(b) $\theta_D(t)$ em vermelho e $\theta_T(t)$ em azul

Figura 5.35 – Swing Up do Pêndulo para T = 20 ms, com atraso variável descrito na figura 7.38, do cap.7.



(a) $\theta_P(t)$ em azul, "modo de controle" em vermelho



(b) $\theta_D(t)$ em vermelho e $\theta_T(t)$ em azul

Figura 5.36 – Swing Up do Pêndulo para T = 40 ms, com atraso variável descrito na figura 7.40, do cap.7.

Utilizando os mesmos registros de atrasos entre 10 ms e 70 ms, para os respectivos períodos de amostragem, obtemos as simulações a seguir para o controle estabilizante.

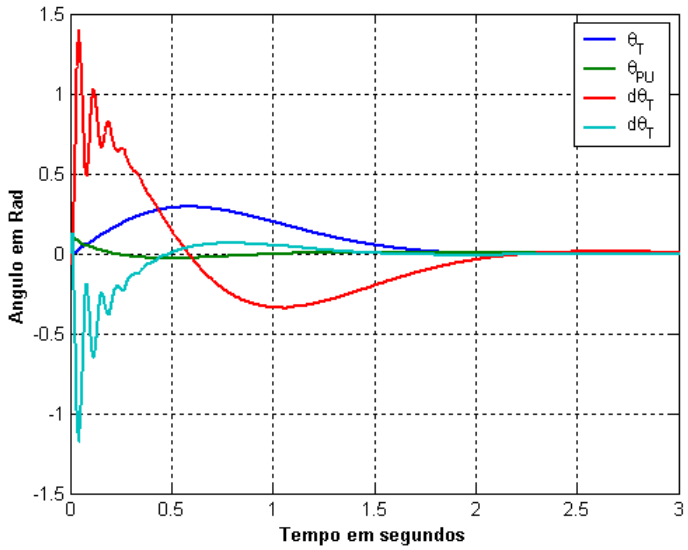


Figura 5.37- Controle estabilizante para T = 10 ms e condições iniciais

$\theta_T = 0, \theta_P = 5.4^o, \dot{\theta}_T = 0, \dot{\theta}_P = 0$. Atraso variável no tempo, com histograma dado pela figura 7.37.

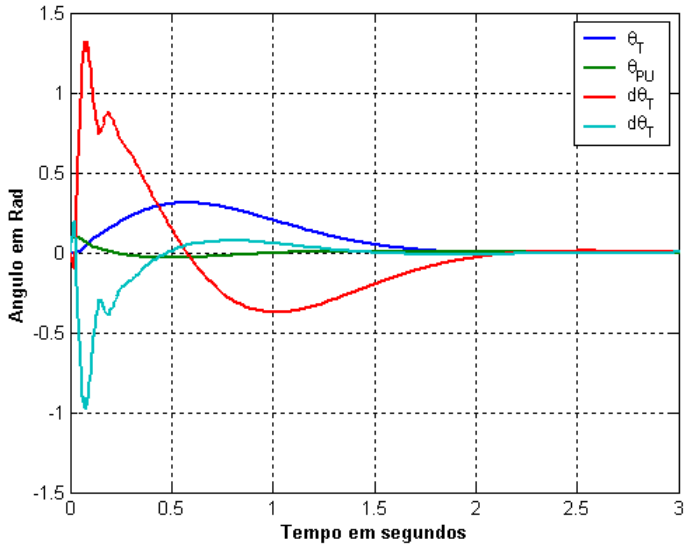


Figura 5.38- Controle estabilizante para T = 20 ms e condições iniciais

$\theta_T = 0, \theta_P = 5.4^o, \dot{\theta}_T = 0, \dot{\theta}_P = 0$. Atraso variável no tempo , com histograma dado pela figura 7.38.

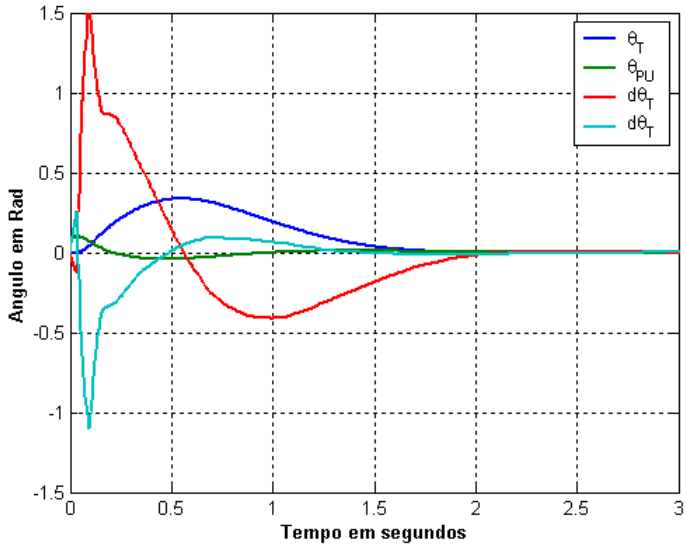


Figura 5.39- Controle estabilizante para T = 40 ms e condições iniciais

$\theta_T = 0, \theta_P = 5.4^o, \dot{\theta}_T = 0, \dot{\theta}_P = 0$. Atraso variável no tempo , com histograma dado pela figura 7.40.

5.4 – Uma abordagem para tornar o atraso de comunicação invariante no tempo

Em sistemas lineares variantes no tempo, muitos dos resultados teóricos para a análise e projeto de sistemas de controle linear invariante no tempo não podem ser utilizadas diretamente. Por exemplo, as análises no domínio da freqüência são aplicáveis somente a sistemas lineares invariantes no tempo, para uma entrada e uma saída.

Por outro lado, podemos representar um sistema linear variante no tempo da seguinte forma no espaço de estados:

$$\dot{\mathbf{x}}(t) = \mathbf{A}(t)\mathbf{x}(t) + \mathbf{B}(t)\mathbf{u}(t) \qquad (5\text{-}50)$$

Para sistemas lineares, discretos e variantes no tempo, temos que :

$$\mathbf{x}(k+1) = \mathbf{\Phi}(k)\mathbf{x}(t) + \mathbf{\Gamma}(k)\mathbf{u}(k) \qquad (5\text{-}51)$$

No entanto, as teorias de controle aplicadas a sistemas lineares invariantes no tempo são mais conhecidas e difundidas do que as teorias para a análise e projeto de sistemas variantes no tempo. Então, para dispormos de um projeto de controle mais simples, apoiado na vasta teoria aplicada a sistemas invariantes no tempo, seria necessário de alguma forma tornar o sistema variante no tempo em invariante.

No caso particular do sistema de controle distribuído, o atraso de comunicação é variante no tempo e aleatório. A figura 5.40 descreve esse sistema e também faz a distinção entre os nós atuador, sensor e controlador pela sua forma de executar. Nesse exemplo, supomos que o sensor executa periodicamente, baseado no agendamento da sua execução em um tempo Ts, e que os nós atuador e o controlador executam tão logo ocorra um evento, como a chegada de um pacote de dado através da rede de comunicação.

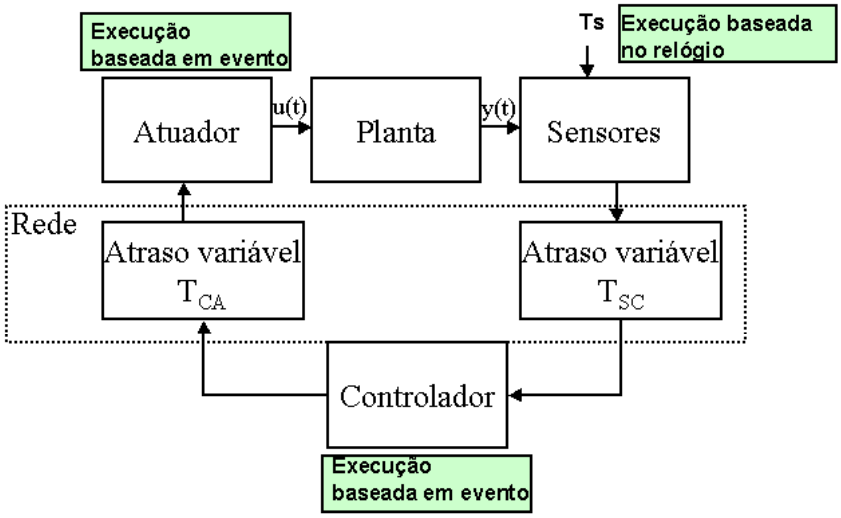


Figura 5.40 – Sistema de Controle Distribuído. Atraso variável no tempo de comunicação.

Para contornar o problema de atraso variável, uma abordagem descrita em [15] é armazenar os pacotes de dados em filas, e apenas utilizar os dados recebidos em tempos agendados maiores que o pior caso de atraso, periodicamente, assumindo que todos os nós (atuador, sensor e controlador) estão sincronizados.

Exemplo: Seja um período de amostragem de Ts segundos e que o atraso máximo do Sensor para o Controlador é de 3 Ts. Pela abordagem descrita acima, não importa que um dado gerado no instante kTs pelo Sensor chegue no Controlador em (k+1)Ts, o controlador somente utilizará o dado em (k+3)Ts. Isso elimina a variabilidade entre os tempos de execução dos pacotes de dados.

A figura 5.41 ilustra a abordagem de [15]:

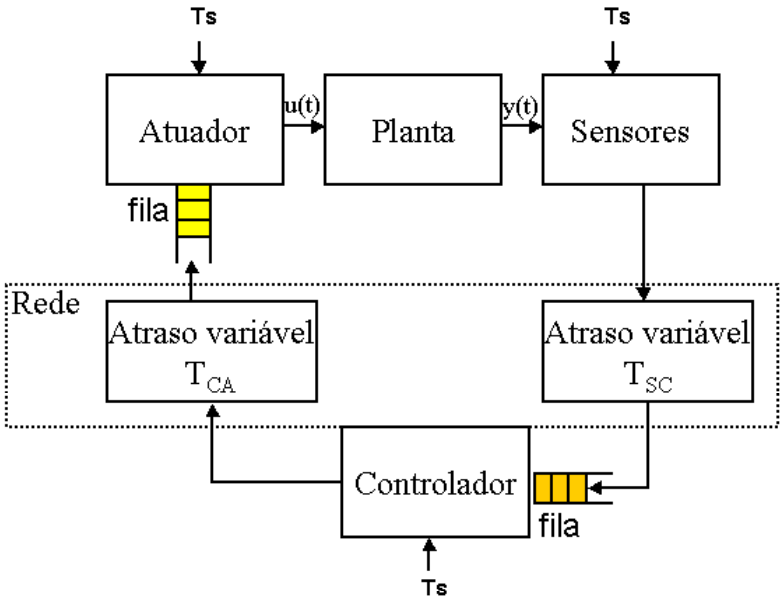


Figura 5.41 – Abordagem de tornar o atraso variável em estático baseado no pior caso. Supõe-se que os nós Controlador, Sensor e Atuador estão sincronizados e executam periodicamente, com período Ts.

Para aplicarmos essa idéia no nosso trabalho, é necessário primeiro conhecer as características do atraso de comunicação. No capítulo 7 serão apresentados vários histogramas para o atraso de comunicação, em função do período de amostragem e da carga do sistema. Vamos analisar um caso, para um período de amostragem de 40 ms. O histograma do atraso de comunicação é dado pela figura 5.42, para o caso em que o sistema operacional Linux está com pouca carga e o enlace *Bluetooth* é dedicado ao sistema de controle. Embora todas as análises sejam feitas para 40 ms, outros períodos de amostragem podem ser utilizados, desde que o respectivo histograma esteja disponível.

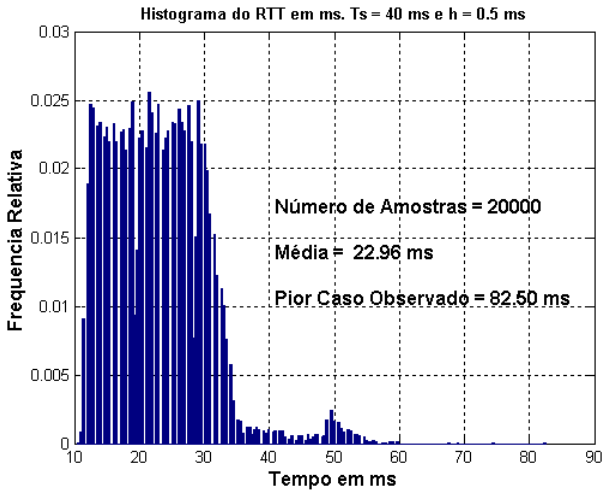


Figura 5.42 – Histograma do atraso de comunicação em ms, para período de amostragem de 40 ms, com pouca carga no sistema operacional e enlace *Bluetooth* dedicado.

Com o histograma da figura 5.42, temos a informação de ida e volta de um pacote de dados, já que o atuador e sensor estão no mesmo nó. Não sabemos sobre o retardo de ida, do sensor para o

controlador, nem o de volta, do controlador para o atuador, mas conhecemos a soma dos dois. Além disso, o atuador e sensor podem ser sincronizados, já que estão no mesmo nó. Uma abordagem, para reduzir a variabilidade do atraso, seria permitir que o atuador utilize um novo dado apenas nos instantes em que nó sensor receber novos dados, a cada período de amostragem. O controlador executa assim que um pacote de dados chegue pelo sistema de comunicação. A figura 5.43 ilustra essa abordagem.

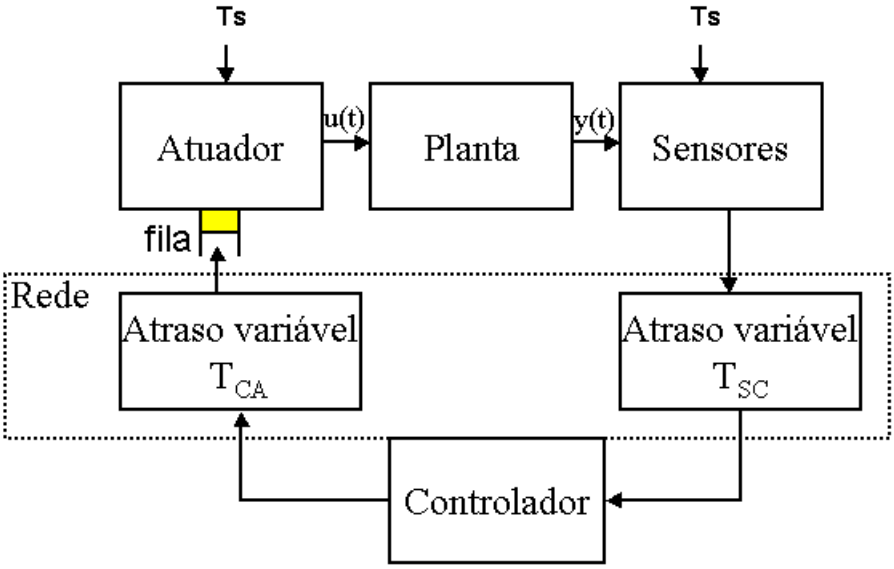


Figura 5.43 – Nó Atuador e Sensor sincronizados. O nó atuador atualiza o sinal de controle u() a cada período de amostragem.

Dessa forma, o novo histograma de atraso deve ser como ilustrado na figura 5.44:

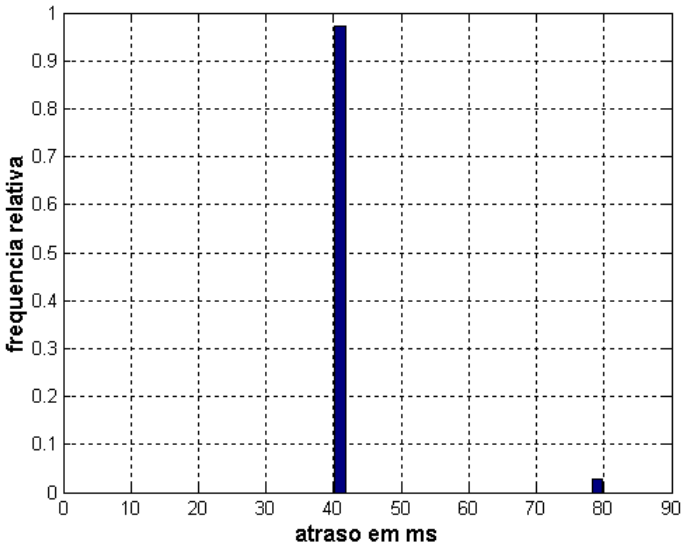


Figura 5.44 – Histograma gerado a partir dos dados da figura 5.42, utilizando a abordagem da figura 5.43. Em 97.26% dos casos, o atraso foi de 40 ms (período de amostragem).

A partir do histograma da figura 5.44, torna-se razoável assumir para o projeto de controle que o sistema possa ser considerado como invariante no tempo, com um atraso estático de 40 ms, embora na realidade o sistema ainda seja variante no tempo. Assume-se que as amostras de atrasos superiores a 40 ms ocorram de forma independente das amostras adjacentes. Não foi considerado o pior caso possível

para fixar o atraso, pois este é muito improvável e poderia ser desprezado nesse caso. Além disso quanto mais segurança temos para garantir que o atraso seja de um determinado valor estático, maior terá que ser o atraso estático que teremos que inserir na malha, e infelizmente o pior caso pode não ser tolerável. O que temos nesse caso é uma troca entre o desempenho do sistema e a segurança para garantir atrasos em um tempo fixo. A partir dessa abordagem, podemos tentar projetar um sistema de controle que compense esse atraso estático, utilizando técnicas de controle linear invariante no tempo.

Um exemplo disso seria aplicar esse método no primeiro estudo de caso. Se for razoável assumir que o atraso é estático, torna-se viável todas as análises e projetos acerca da margem ao atraso estático, como foi descrita na seção 5.1.2.

Na representação no espaço de estados de sistemas discretos, um atraso maior ou igual a um período de amostragem será representado abaixo. O atraso pode ser definido como $RTT_{estático}$ = LTs + mTs, para L = 1,2,3,... e 0≤m≤1, e Ts é o período de amostragem.

$$\mathbf{X}(kT_s + T_s) = \Phi\mathbf{X}(kT_s) + \Gamma_1 u(kT_s - LT_s) + \Gamma_2 u(kT_s - LT_s + T) \quad (5\text{-}52)$$

$$\text{Para: } \Phi = \mathbf{e}^{\mathbf{A}Ts}, \Gamma_1 = \int_{mTs}^{Ts} \mathbf{e}^{\mathbf{A}t}\,\mathrm{B}\mathrm{d}t\,,\ \Gamma_2 = \int_0^{mTs} \mathbf{e}^{\mathbf{A}t}\,\mathrm{B}\mathrm{d}t \quad (5\text{-}53)$$

Vamos dar um exemplo:

Exemplo 2: para $RTT_{estático}$ = 2 Ts + 0.1 Ts e $\mathbf{X} = [x_1]$. Temos que:

$$x_1(kT_s + T_s) = \Phi x_1(kT_s) + \Gamma_1 u(kT_s - 2T_s) + \Gamma_2 u(kT_s - T_s)$$

$$\Phi = e^{aTs}, \qquad \Gamma_1 = \int_{0.1Ts}^{Ts} e^{a\tau}\mathrm{B}\mathrm{d}\tau \quad , \quad \Gamma_2 = \int_0^{0.1Ts} e^{a\tau}\mathrm{B}\mathrm{d}\tau$$

Substituindo:

$$u(kT_s - 2T_s) = x_2$$

$$u(kT_s - T_s) = x_3$$

$$\begin{bmatrix} x_1(T_s(k+1)) \\ x_2(T_s(k+1)) \\ x_3(T_s(k+1)) \end{bmatrix} = \begin{bmatrix} \Phi & \Gamma_1 & \Gamma_2 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1(T_s k)) \\ x_2(T_s k) \\ x_3(T_s k) \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} u(kT_s)$$

Os livros de Franklin e Powell, [43], e Ogata, [44] ensinam como representar um atraso estático arbitrário, mesmo para $RTT_{estático} < T_s$. Agora vamos abordar um exemplo para a aplicação desse representação no espaço de estados para compensar o atraso estático.

No caso do pêndulo invertido, o segundo estudo de caso, com período de amostragem de 40 ms, podemos utilizar a abordagem para reduzir a variabilidade do atraso como descrito na figura 5.43, e analisar o sistema como se o atraso fosse estático e igual a 40 ms. Logo o atraso $RTT_{estático}$ é definido em (5-53) por L = 1 e m = 0. A representação no espaço de estados do sistema discreto é :

$$\mathbf{X}(kT_s + T_s) = \Phi\mathbf{X}(kT_s) + \Gamma u(kT_s - T_s)$$

$$\text{com } \mathbf{X} = [x_1, x_2, x_3, x_4]^T = [\theta_T, \theta_P, \dot{\theta}_T, \dot{\theta}_P]^T$$

Fazendo $u(kT_s - T_s) = x_5(kT_s)$, temos que:

$$\begin{bmatrix} \mathbf{X}(kT+T) \\ x_5(kT+T) \end{bmatrix} = \underbrace{\begin{bmatrix} \boldsymbol{\Phi} & \boldsymbol{\Gamma} \\ 0 & 0 \end{bmatrix}}_{\mathbf{F}} \begin{bmatrix} \mathbf{X}(kT) \\ x_5(kT) \end{bmatrix} + \underbrace{\begin{bmatrix} \mathbf{0} \\ 1 \end{bmatrix}}_{\mathbf{G}} u(kT) \qquad (5\text{-}54)$$

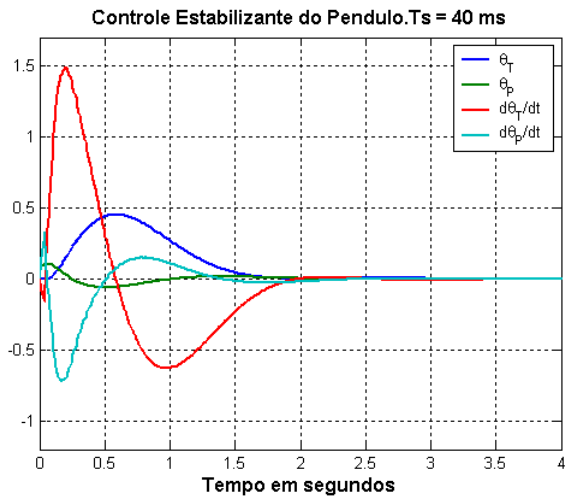
Utilizando as matrizes **F** e **G** de (5-54), e as matrizes **Q** e **R** para resolver o problema do regulador linear quadrático e calcular o vetor de ganhos K do sinal de controle ótimo, podemos considerar explicitamente o atraso de comunicação no projeto de controle. Vamos terminar esse exemplo, calculando o valor de K para as seguinte matrizes **Q** e **R**.

$$\mathbf{Q} = \begin{bmatrix} 0.25 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \mathbf{R} = 0.05$$

A solução do problema LQR resulta em:

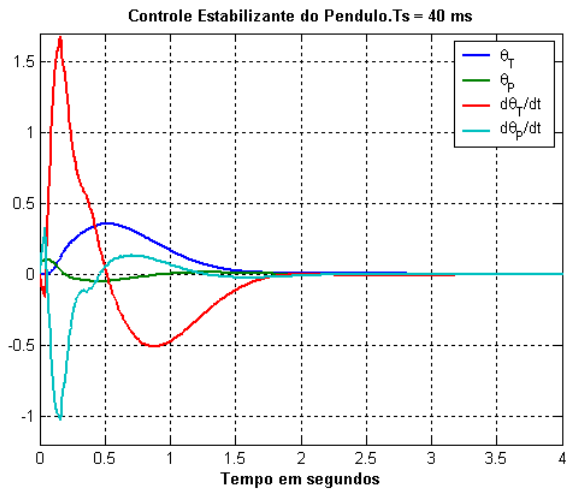
K = [-0.3607 -12.3739 -1.2664 -1.7141 0.7508] (5-55)

A figura 5.45 ilustra os resultados dessa abordagem para as condições iniciais do sistema do pêndulo $\theta_T = 0, \theta_P = 5.4^o, \dot{\theta}_T = 0, \dot{\theta}_P = 0$, como na figura 5.39.



(a) Utilizando compensação ao atraso estático de 40 ms.

K = [-0.3607 -12.3739 -1.2664 -1.7141 0.7508]



(b) Sem considerar o atraso estático de 40 ms no projeto.

K =[-0.3607 -8.6042 -0.8489 -1.2970]

Figura 5.45 – Controle Estabilizante, para período de amostragem de 40 ms, cujo histograma de atrasos é o da figura 5.43. Condições iniciais $\theta_T = 0, \theta_P = 5.4^o, \dot{\theta}_T = 0, \dot{\theta}_P = 0$

Embora os resultados da figura 5.45 (a) sejam um pouco melhores do que os resultados dos gráficos de 5.45 (b) e 5.39, a abordagem de trocar um atraso variante no tempo, por um atraso estático muito maior nem sempre representa uma boa abordagem. Por exemplo, um histograma para Ts = 0.01 está descrito na figura 7.37 do capítulo 7, novamente para um caso sem interferências na comunicação. Se utilizássemos uma abordagem para fixar o atraso em 30 ms, na realidade para 97,02% dos casos, teríamos a seguinte resposta para as condições iniciais $\theta_T = 0, \theta_P = 5.4^o, \dot{\theta}_T = 0, \dot{\theta}_P = 0$.

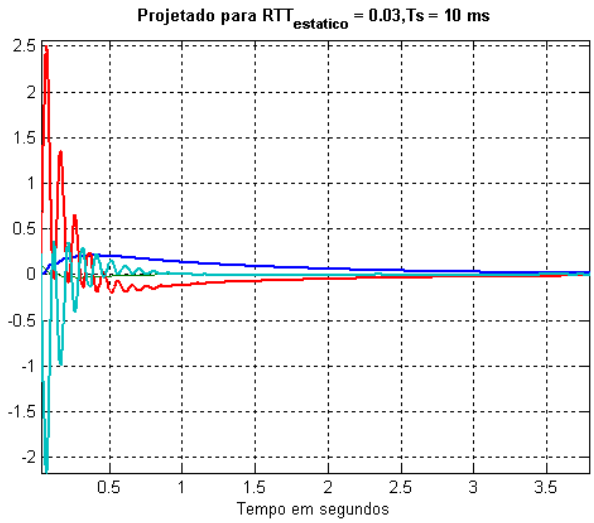


Figura 5.46 – Controle Estabilizante, para um atraso fixo de 30 ms, para período de amostragem de 10 ms. Condições iniciais $\theta_T = 0, \theta_P = 5.4^o, \dot{\theta}_T = 0, \dot{\theta}_P = 0$.

No capítulo 7 serão apresentados histogramas para outros cenários em que o atraso de comunicação varia numa faixa muito maior de tempo, devido a interferência na comunicação por causa da carga do sistema e de compartilhamento do enlace *Bluetooth* com outros tráfegos de dados. Nesses casos, o pior caso chega a atrasos superiores a 500 ms, dependendo da taxa de amostragem, e esse tipo de atraso ocorre em rajadas, afetando vários pacotes de dados consecutivos.

Capítulo 6

Visão Geral do Sistema Implementado

Nesse seção vamos descrever os programas implementados e descrever como os vários processos do Linux e do RTLinux/Free trocam dados para gerar os dados do sistema de monitoramento ou de controle distribuído. No capítulo 5 foram descritos os sistemas físico que devem ser emulados por uma tarefa em tempo real, descrevendo os modelos simplificados de um motor DC e de um pêndulo rotacional invertido. No capítulo 5 também foram abordadas algumas restrições quanto a validade das simulações, principalmente se o atraso de comunicação interferir no controle. No entanto, para aplicações de monitoramento, o cliente que se comunica via *Bluetooth* não faz parte do sistema de controle, e para esses casos o atraso de comunicação não precisa ser considerado como um impecilho.

Vamos descrever primeiro uma visão geral do sistema, para em seguida descrever cada um dos processos que fazem parte deste trabalho, indicando também algumas práticas de programação em linguagem C que são necessárias para o funcionamento do sistema.

6.1 – Visualização Completa do Sistema

Como descrito no capítulo 1, nos cenários desses estudos de caso temos 2 computadores pessoais. Ambos os computadores estão conectados aos seus dispositivos Bluetooth USB, e possuem o *software* necessário para utilizar esse tipo de comunicação sem fio. As aplicações que rodam em cada computador serão descritas a seguir.

6.1.1 – Comunicação entre processos

Um computador possui um sistema operacional Linux, com a versão mais atualizada do Bluez, a implementação em *sotfware* dos protocolos e interfaces do hospedeiro do *Bluetooth*. Um processo no espaço do usuário roda uma aplicação cliente que utiliza interface de socket para acessar aos serviços da camada L2CAP e tentar se comunicar com um servidor em outro máquina que também utilize a comunicação Bluetooth e esteja esperando por conexões numa determinada PSM, um número de identificação dado a cada aplicação que utiliza o L2CAP. Esse programa cliente negocia com o servidor que tipo de serviço quer utilizar, como monitoramento ou testar algum algoritmo de controle, e quando começa a receber dados do servidor irá colocar os dados recebidos em uma fila FIFO, para que outra aplicação, de menor prioridade, faça o registro dos dados ou os imprima sob a forma de gráfico x amostra. Isso é ilustrado na figura 6.1.

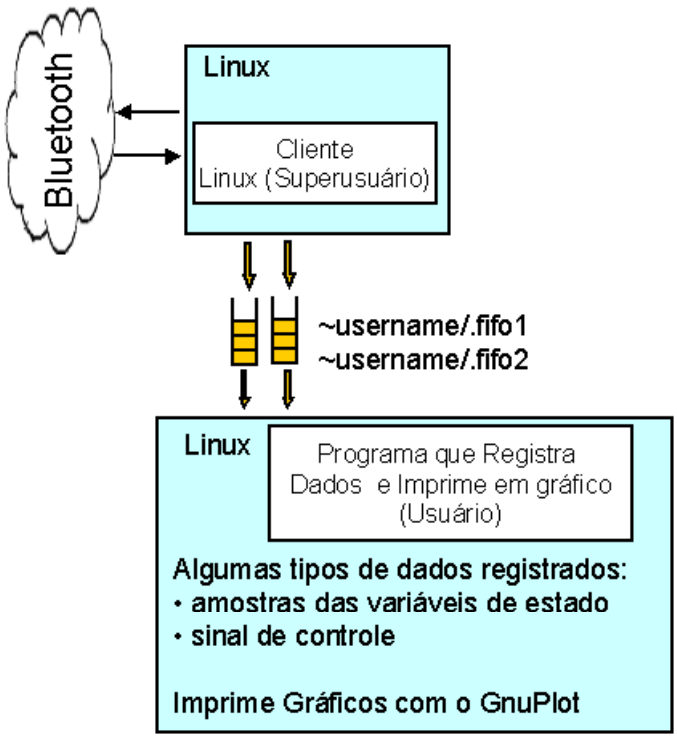


Figura 6.1 – Esboço da comunicação entre processos no PC cliente.

O outro computador possui dois sistema operacionais executando em conjunto: o RTLinux/Free e o Linux, que também possui o Bluez. Inicialmente, uma tarefa em tempo real é executada periodicamente apenas para verificar se algum dado chegou em uma determinada fila RT-Fifo. No Linux, um processo espera por conexões que solicitem conexão através camada L2CAP. Caso essa solicitação chegue, o servidor cria um processo filho para atender a conexão e vai dormir até que o processo filho termine de executar. O processo filho começa a negociar que tipo de aplicação será utiliza, recebendo parâmetros como condições iniciais, período de amostragem e informações como o tipo de abordagem de controle e seus parâmetros. Em seguida a aplicação envia os dados de inicialização para a tarefa do RTLinux/Free, que deve começar a calcular a resolução numérica das equações diferencias ordinárias para as condições iniciais especificadas. A partir daí, os dados com as medidas das variáveis de estado são colocados nas filas RT-Fifo para serem adquiridas pelo processo servidor no Linux. Para realizar outras medidas de interesse, um outro processo do Linux espera por dados em filas RT-Fifo diferentes para registrar ou apresentar em um gráfico amostras de variáveis de estado a cada execução do simulador e não apenas nos períodos de amostragem, amostras do tempo de ida e volta de um pacote, medido pela tarefa em tempo real, ou mesmo o tempo necessário para a tarefa em tempo real executar o método numérico para um passo. Tudo isso é ilustrado na figura 6.2.

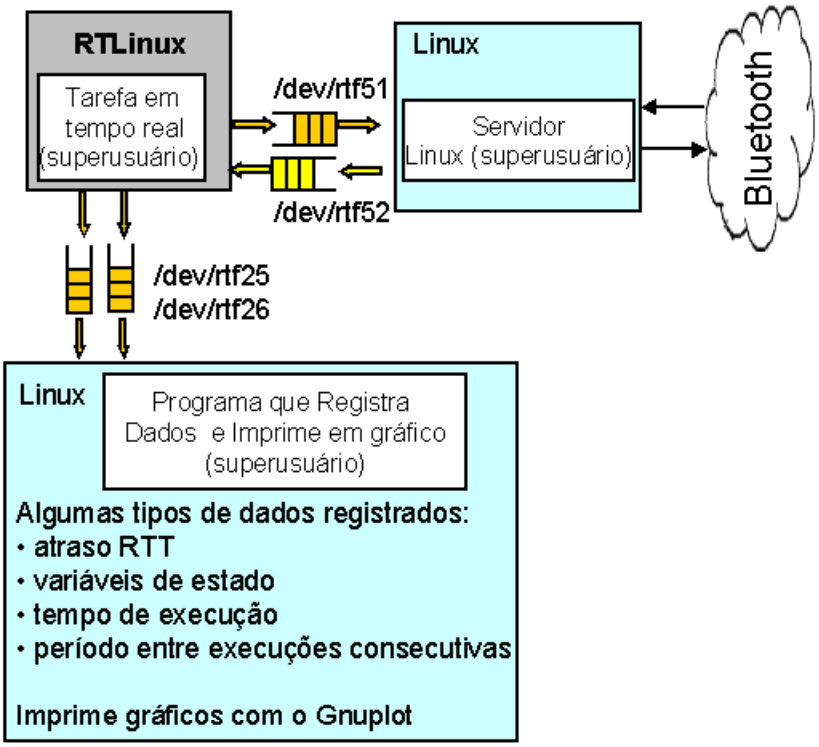


Figura 6.2- Esboço da comunicação entre processo no PC Servidor

Como pode ser percebido, o servidor somente atende um cliente por vez, o que é recomendado para controle distribuído, mas não é recomendado para aplicações de monitoramento, que poderiam estar atendendo a vários usuários, no entanto isso fica para um aprimoramento futuro.

6.1.2 – Estruturas de Dados

As estruturas de dados utilizadas para trocar dados entre o servidor, cliente e a tarefa em tempo real (simulador), vão ser descritas a seguir.

Para a comunicação entre cliente e servidor, são utilizadas duas estruturas de tamanho fixo, com descritas pela figura 6.3. O campo ID, de 16 bits, pode ser utilizado para a identificação do propósito de um determinado pacote, para 256 valores diferentes, o que ajuda aos programas de cliente e servidor a trocar informações sobre a conexão. No entanto, o campo ID pode ser retirado para transmitir dados de controle em um fluxo, sem a necessidade de encapsular os dados. O campo estampa de tempo de 64 bits é utilizado para transportar o tempo em nano segundos, que marca o instante em que o pacote foi transmitido pela tarefa do RTLinux. Esse campo é copiado em todos os pacotes de dados, até que retorne ao RTLinux, onde será contabilizado o atraso de comunicação. O campo Dados serve para transportar os sinais digitais. O campo Dados de Inicialização é utilizado para transportar os dados que configuram a simulação, durante as primeiras trocas de dados entre cliente e servidor.

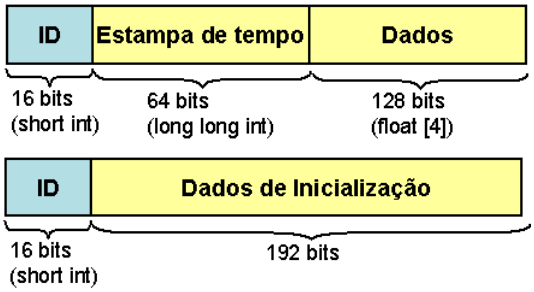


Figura 6.3 – Estruturas de dados utilizada entre o cliente e o servidor

Para a comunicação entre o servidor e a tarefa do RTLinux, são utilizados 2 pacotes, similares aos da figura 6.3, e com as mesmas funções.

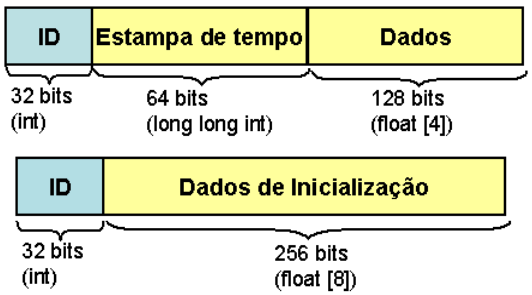


Figura 6.4 – Estrutura de dados utilizada para comunicação entre o servidor e a tarefa em tempo real (simulador)

6.1.3 – Diagramas de tempo

Os diagramas de tempo ilustram em que ordem deve ocorrer a troca de dados entre os processos cliente-servidor-simulador, para que se inicie a troca de dados. As figuras 6.5 e 6.6 ilustram como está sendo feito os procedimentos de inicialização do simulador e início da troca de dados entre o simulador e o cliente, tendo o servidor como intermediário, para dois casos distintos: para a opção de serviço de monitoramento, onde o cliente não participa do sistema de controle, apenas observa os dados; e para a opção de controle distribuído, onde o cliente recebe dados e calcula o sinal digital de controle, que deverá ser enviado de volta ao simulador:

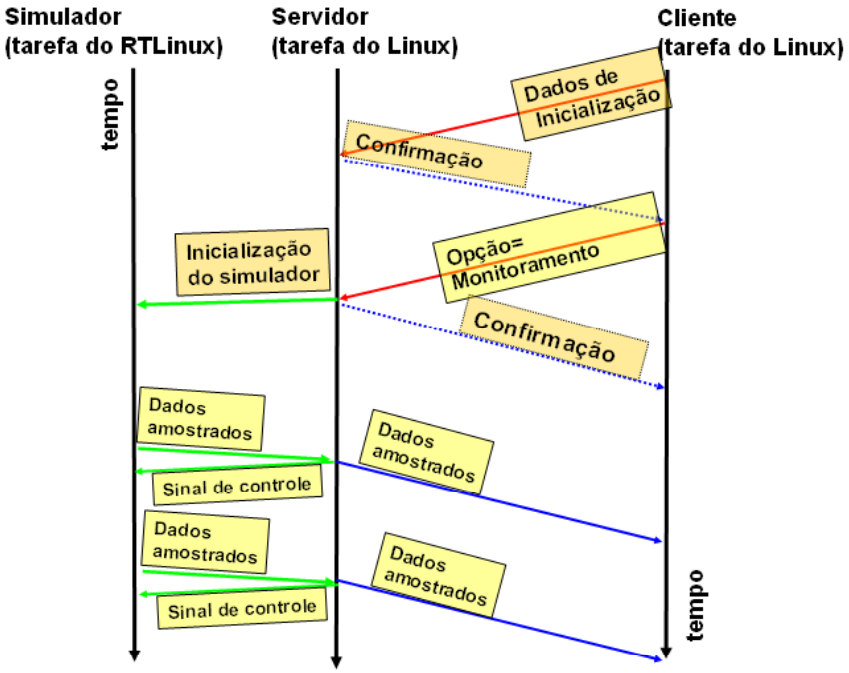


Figura 6.5 – Diagrama de tempo para a configuração do simulador e início de troca de dados para controle distribuído, para serviço de monitoramento.

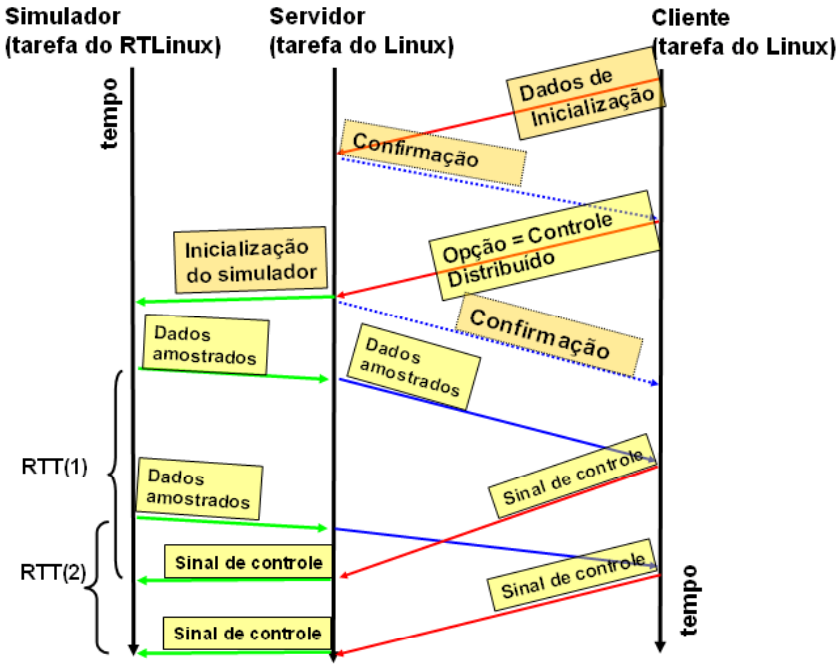


Figura 6.6 – Diagrama de tempo para a configuração do simulador e início de troca de dados para controle distribuído, para serviço de controle distribuído.

6.2 – PC servidor

No PC servidor, vamos ter que utilizar sempre os privilégios de superusuário para inicializar os módulos do RTLinux/Free, e também para poder executar os processos em espaço do usuário do Linux, já que usuários normais não podem acessar as RT-Fifos, que é um espaço em memória compartilhada pelo

módulo do RTLinux/Free e algum processo do Linux. O processo servidor também opera com alto prioridade para executar o mais rápido possível, em detrimento dos outros processos do Linux, e essa alteração de prioridade somente pode ser feito com os privilégios de superusuário. O código para alterar a prioridade e a política do agendador é descrita abaixo:

```
int sched_policy;
struct sched_param sched_param;
sched_param.sched_priority = 1;
sched_setscheduler(0, SCHED_FIFO, &sched_param);
```

6.2.1 – Inicializando o simulador

Para inicializar o simulador, basta utilizar os comandos que carregam os módulos essenciais do RTLinux/Free, e em seguida ir ao diretório onde se localiza o módulo do simulador e inseri-lo no Kernel. A seguir os comando usualmente utilizados:

No diretório do módulo simulador.o
Exemplo de diretório utilizado : /usr/src/rtlinux/simulador/
Faça:
root# rtlinux start
root# rtlinux start simulador

Pode ser um bom procedimento inserir antes do simulador.o, o módulo rtl_debug.o, descrito no capítulo 3, para depuração e evitar que um problema no simulador faça o computador travar.

Existem dois módulo possíveis, um que simula a planta do motor DC descrita na seção 5.2, e outra que simula o sistema do pêndulo invertido descrita na seção 5.2. A tabela 6-1, apresenta o nome do módulo

Tabela 6-1 : Métodos numéricos

Nome do Módulo	Método de Resolução numérica das equações diferenciais ordinárias	Passo de tempo fixo
Motor.o	Runge Kutta de 4ª ordem	1 ms
Pendulo_motor.o	Runge Kutta de 4ª ordem	500 μs

6.2.2 – Inicializando o servidor

Para inicializar o servidor é necessário observar alguns:

- O dispositivo Bluetooth USB está conectado no computador e foi detectado como um dispositivo USB. O driver hci_usb.o usualmente é carregado assim que se conecta o dispositivo Bluetooth USB. Isso pode ser verificado com o comando dmesg no prompt do Linux;

- Todos os módulos necessários do Bluez estão carregados. O l2cap.o, hci_usb.o e bluez.o,. Isso pode ser verificado com o comando lsmod;
- O executável deve ser gerado por um usuário com privilégios de superusuário, do contrário a alteração de prioridade e da política do agendador será negada, e as filas RT-Fifos não estarão acessíveis;
- O servidor foi configurado para escutar na porta PSM correta. Do contrário, nenhum pedido de conexão será encaminhado para o servidor. Nesse trabalho, vamos utilizar dois servidores, um para cada estudo de caso, com PSM iguais a 1032 e 1033, para os primeiro e segundo caso respectivamente. A figura 6.6. ilustra a PSM do primeiro caso;
- No código do servidor, além da PSM, deve-se inicializar 2 parâmetros chamados imtu e omtu. Estes parâmetros descrevem o tamanho máximo do pacote L2CAP em bytes. Deve-se utilizar valores maiores ou iguais aos do maior pacotes de dados que for enviado, do contrário ocorre um bug no programa;

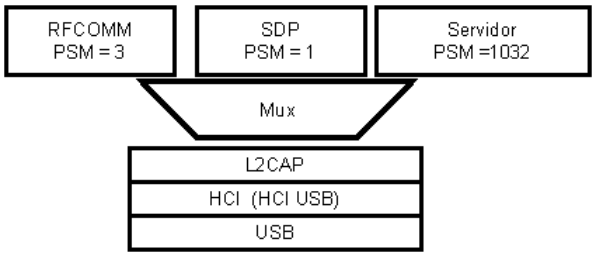


Figura 6.7 – Valor da PSM do Servidor do primeiro caso de estudo. Para o segundo caso, a PSM é igual a 1033.

O programa servidor somente pode ser executado se a tarefa do RTLinux já tiver inicializado as RT-Fifos, senão ocorrerá um erro quando o programa do servidor tentar abrir o arquivo /dev/rtfxx.

6.2.3 – Inicializando o programa Registrador

O programa registrador tem uma função bem simples: adquirir dados da RT-Fifo e armazená-las em arquivo. No entanto, o executável deve ter permissão para ler as RT-fifos. Os dados armazenados em arquivos com o nome da variável amostrada, sempre inserindo o número da amostra ou o tempo associado e o valor dela em cada linha do arquivo. Em seguida chama-se um programa chamado de GnuPlot, através de uma função em C chamada de popen(), para imprimir os gráficos desses dados na tela do computador. Qualquer outra interface gráfica poderia ser utilizada para ler os arquivos de registro e gerar gráficos na tela, no entanto o GnuPlot é bastante simples de se utilizar no Linux. Os gráficos são usualmente atualizados de 5 em 5 segundos, no entanto os arquivos podem ser atualizado numa freqüência muito maior, dependendo do tipo de variável amostrada. O código utilizado para chamar o GnuPlot de um programa em C está descrito no Apêndice A, no programa de nome gravador_pendulo.c ou gravador_motor.c

6.3 – PC cliente

No computador com o PC cliente temos a aplicação que deve ser inicializada com o valor correto da PSM do servidor para o respectivo estudo de caso. Deve-se então ajustar as condições iniciais do sistema simulado, descrever qual a lei de controle utilizada, os ganhos dessa lei de controle, o período de amostragem, entre outros. Depois basta se conectar via *Bluetooth* com um outro computador, que tenha o servidor correto sendo executado sobre a camada L2CAP. Os dados recebidos pela aplicação cliente ou são utilizados para calcular o sinal de controle, no caso de controle distribuído, ou são simplesmente passados a outras aplicações que irão registrar e/ou imprimir os dados na tela para a vizualização, através de filas fifos. O programa que registras os dados é idêntico ao descrito na seção 6.2.3 e não será descrito, contudo o código está no Apêndice, sob o nome de gravador.c ou gravador2.c.

O código para criar filas fifos por um usuário do Linux é:

```
mknod(".fifo11",S_IFIFO | 0644 , 0)
```

Esse comando cria uma fila fifo, que pode ser aberta com um arquivo, no diretório onde foi executado o processo do cliente. Por exemplo, se o cliente está em /user/cliente/, basta abrir o arquivo /user/cliente/.fifo1

Para abrir o arquivo, no entanto, é necessário utilizar o comando open(), corretamente, do contrário o processo pode bloquear esperando que um outro processo leia a fila fifo, o que não é desejável. A maneira correta de utilizar o comando open() é, por exemplo:

```
open(".fifo11", O_RDWR| O_NONBLOCK )
```

Dessa maneira o processo que abrir a fila fifo não irá bloquear, mesmo que a fila fique cheia.

6.3.1 – Utilizando a aplicação cliente

Uma amostra do código está no Apêndice A. Vamos descrever apenas os argumentos das funções abaixo, e para que servem:

```
void cliente_pendulo_motor(char * origem, char * destino,int controle,float ganho[5],float Tamostragem, int setup,float cond_init[4],float tempo)

void cliente_motor(char * origem, char * destino,int control,float Referencia,float ganho[3],float Tamostragem,int setup,float cond_init[2],float tempo)
```

- origem/destino : BD_ADDR dos dispositivos Bluetooth de origem e destino;
- controle – índice que descreve que tipo de abordagem de controle será utilizada. (ex. PI, LQR);
- ganho[] – ganhos do lei de controle;
- Tamostragem – período de amostragem. Será enviado para o simulador;
- setup – descreve se vamos requisitar os serviços de monitoramento ou de controle distribuído;
- cond_init [] – vetor com as condições iniciais do sistema;

- tempo – tempo de duração da simulação em segundos

Também é necessário que o executável seja gerado com privilégios de superusuário, para que se consiga mudar a prioridade do processo para alta e a política do agendador do Linux para ser baseado em prioridades e não de compartilhamento de tempo.

O código está descrito no Apêndice A, em blue_com.c e blue_com.h. Os parâmetros utilizados estão programa cliente_motor.c ou cliente_pendulo.c.

CAPÍTULO 7

Resultados Obtidos

Nesse capítulo serão apresentadas alguns dados registrados durante as simulações em tempo real. Os dados registrados são do tipo:

- as variáveis de estado amostradas;
- registro do tempo de ida e volta (*round trip time*) de um pacote de dados;

Com os registros de atraso dos pacotes de dados de comunicação, o conhecimento das condições iniciais do sistema dinâmico (que é invariante no tempo) e da abordagem de controle utilizada, é possível tentar reproduzir as simulações em tempo real, com simulações que não ocorrem em tempo real. Com isso, pode-se checar por erros de implementação ou por problemas de estabilidade numérica, utilizando-se de métodos numéricos mais sofisticados como o método de Dormand Prince (ode45 no Matlab), para a resolução das equações diferenciais. Para este fim, será utilizado o Simulink (um simulador de sistemas dinâmicos também do Matlab) para gerar dados que possam ser utilizados para comparação. No Simulink, uma função chamada de "*Variable Transport Delay*" recebe como argumento os registros do atraso de comunicação, e com isso atrasa a atualização do sinal de controle aplicado a planta, modelando dessa forma os efeitos do sistema de comunicação no sistema de controle.

Na seção 7.1, vamos apresentar 2 cenários distintos para as simulações do primeiro estudo de caso. Os 2 cenários são os seguintes:

- No primeiro cenário não haverá processos do sistema operacional Linux concorrendo com as aplicações cliente e servidor *Bluetooth*, o que torna o atraso de comunicação via *Bluetooth* bastante reduzido e limitado. O cliente requisita ao servidor pelo serviço de controle distribuído. O caso de monitoramento não será simulado por ser trivial;
- No segundo cenário, duas transmissões diferentes e concorrentes serão feitas a partir da camada L2CAP do *Bluetooth*. Isso significa que o enlace *Bluetooth* terá que compartilhar o seus recursos, aumentando o tempo aumentando o tempo para que um pacote de uma aplicação seja servido pela camada L2CAP. A figura 7.1 ilustra esse cenário. No restante, é o mesmo sistema descrito no primeiro cenário;

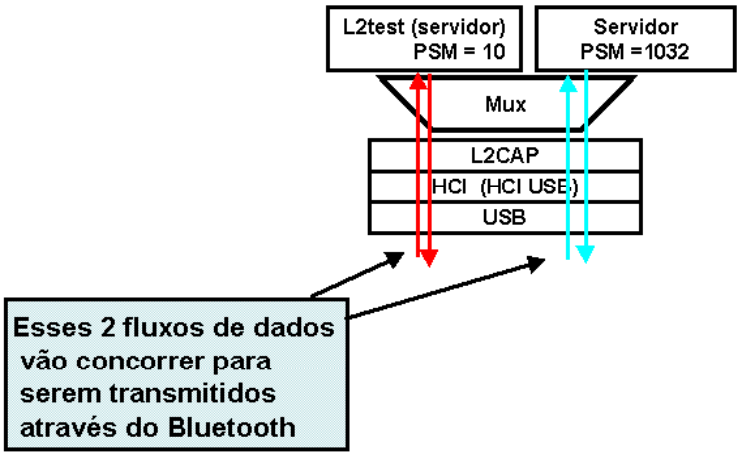


Figura 7.1 – Segundo cenário para as simulações em tempo real do primeiro caso de estudo.

Na seção 7.2, vamos apresentar as simulações em tempo real para o segundo estudo de caso, para outros dois cenários distintos:

- Um cliente remoto configura um servidor através do *Bluetooth* para que este servidor faça o papel de nó controlador, de forma que na mesma máquina estejam todas as funções de um nó controlador, sensor e atuador. O cliente apenas faz o registro dos dados;
- O cliente faz o papel de nó controlador. Evita-se qualquer tipo de interferência que aumente o atraso de comunicação;

7.1 – Resultados na simulação do sistema de 2ª ordem

- Cenário 1: controle distribuído e atraso reduzido devido a pouca concorrência no sistema operacional Linux e na comunicação *Bluetooth*. Em todos as simulações o período de 100 ms foi utilizado. Sempre se assume que i(0) = 0 e ω(0) = 0.

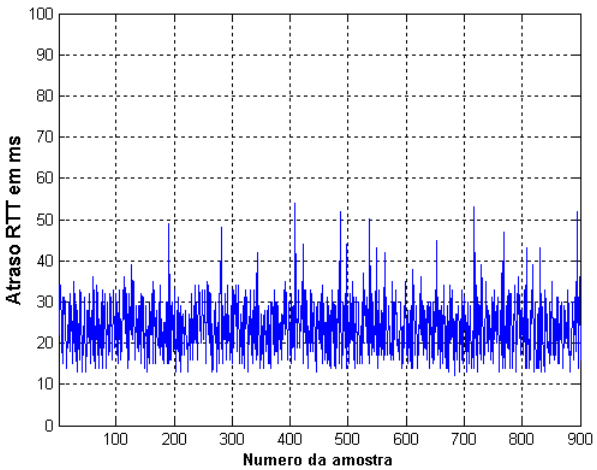


Figura 7.2 – Amostras do atraso de comunicação no primeiro cenário.

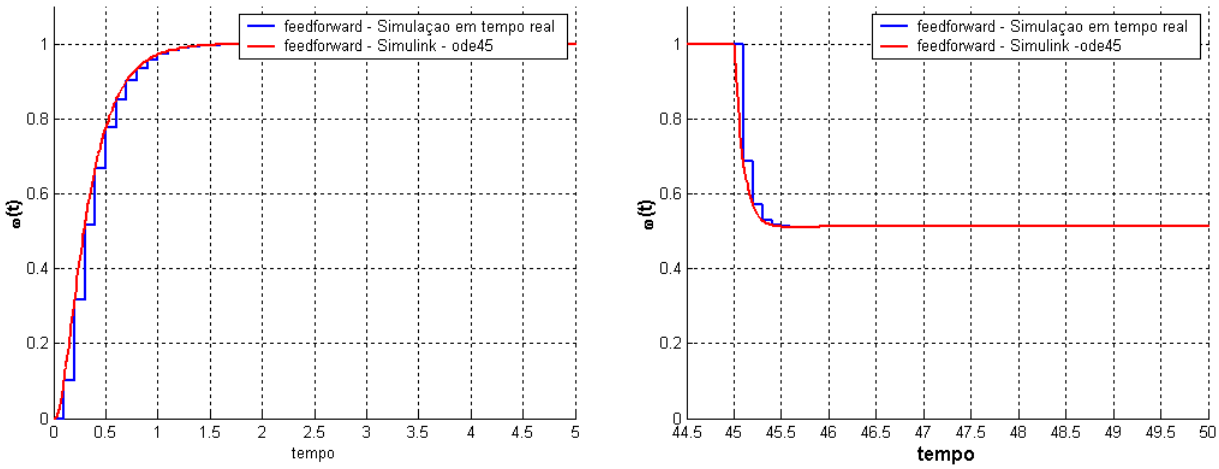


Figura 7.3 –ω(t), para a abordagem de controle a malha aberta (FeedForward). A comparação entre os dados amostrados ω(kT) (em azul, aplicados a um ZOH), e a simulação no Simulink estão em concordância.

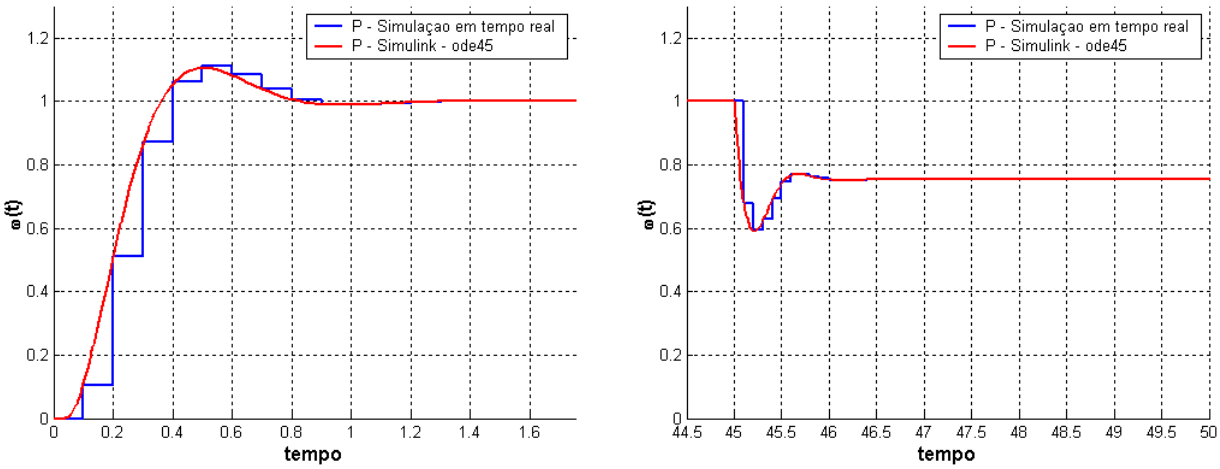


Figura 7.4 –ω(t), para a abordagem de controle Proporcional. A comparação entre os dados amostrados ω(kT) (em azul, aplicados a um ZOH), e a simulação no Simulink estão em concordância.

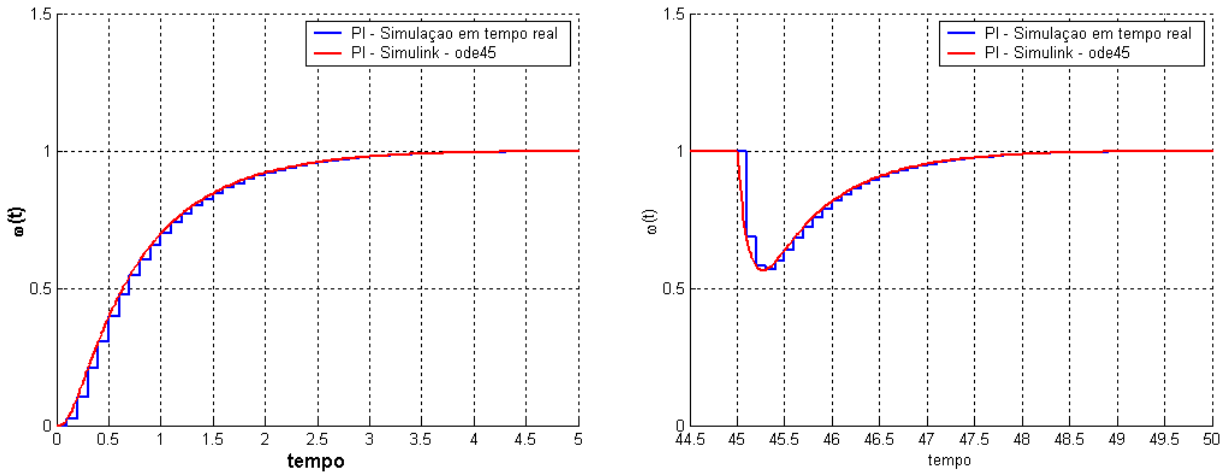


Figura 7.5 –ω(t), para a abordagem de controle com ação PI.. A comparação entre os dados amostrados ω(kT) (em azul, aplicados a um ZOH), e a simulação no Simulink estão em concordância.

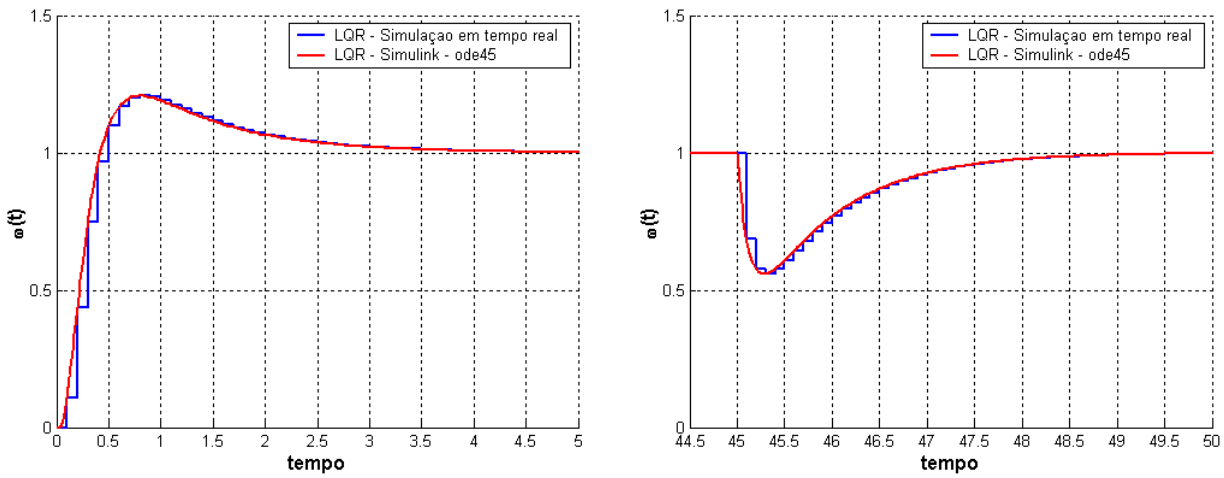


Figura 7.6 –ω(t), para a abordagem de controle com Regulador Linear Quadrático. A comparação entre os dados amostrados ω(kT) (em azul, aplicados a um ZOH), e a simulação no Simulink estão em concordância.

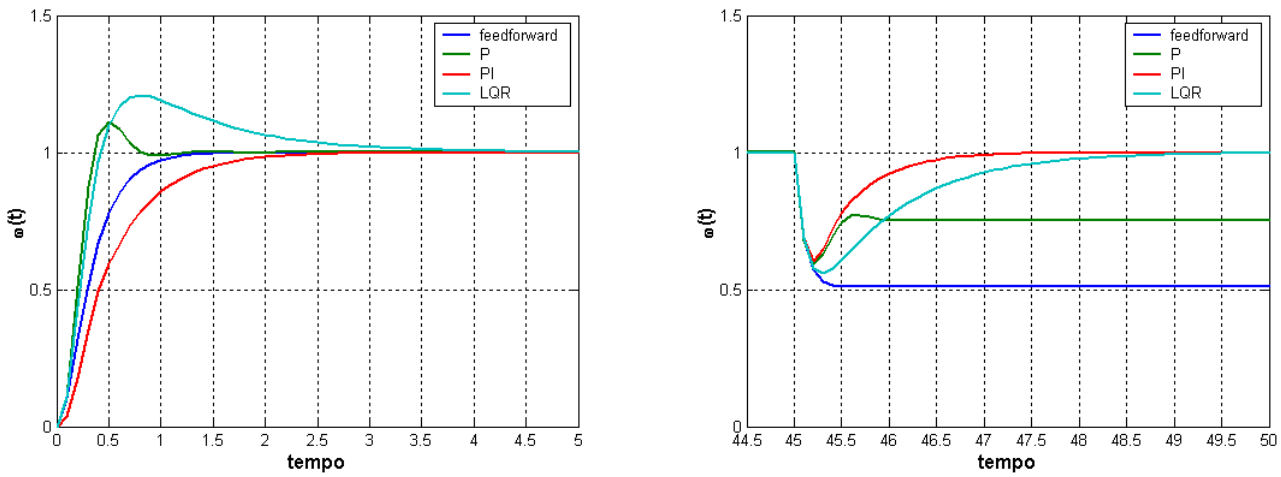


Figura 7.7 –ω(t), obtido no simulador em tempo real para as várias abordagens de controle .

- Cenário 2: Compartilhamento da comunicação Bluetooth com outra aplicação implica em controle distribuído com maior atraso de comunicação.

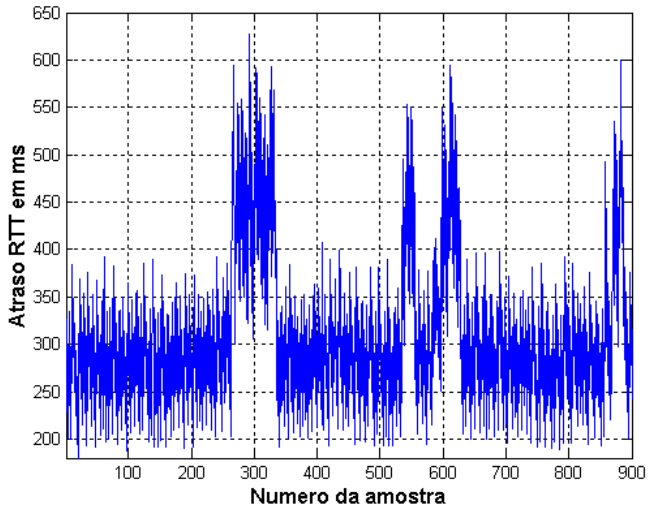


Figura 7.8 – Amostras do atraso de comunicação no segundo cenário.

Infelizmente a verificação nesse segundo cenário ficou prejudicada com um problema no Simulink, no bloco "Variable Transport Delay". Para atrasos da ordem de 300 ms, o atraso que o bloco "Variable Transport Delay" insere no sistema acaba sendo menor do que os registrados, o que compromete a comparação entre simulações. Por exemplo, na figura 7.9, o atraso registrado é de 341 ms, no entanto o sinal de controle já é atualizado 300 ms depois, 41 ms antes que o esperado. Isso faz com que até a resposta do sistema que utiliza o compensador FeedForward, fique um pouco atrasada em relação a resposta do sistema gerada pelo Simulink.

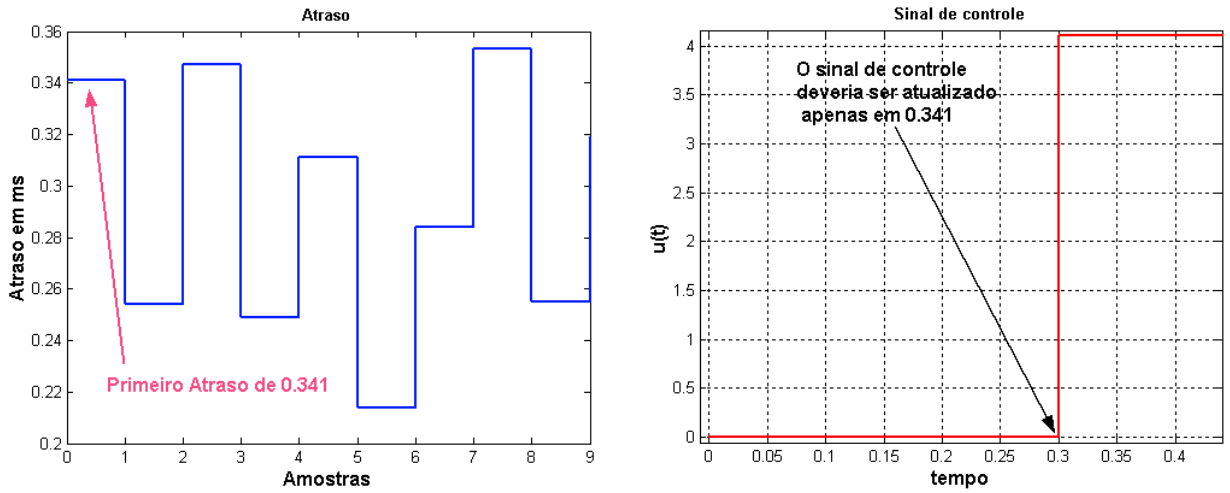


Figura 7.9- Problema detectado na função do Simulink: "Variable Transport Delay". Esse problema não ocorreu para pequenos tempos de atraso.

Mesmo assim, vale conferir os resultados das duas simulações, que embora não sejam exatamente iguais, devem ter respostas semelhantes.

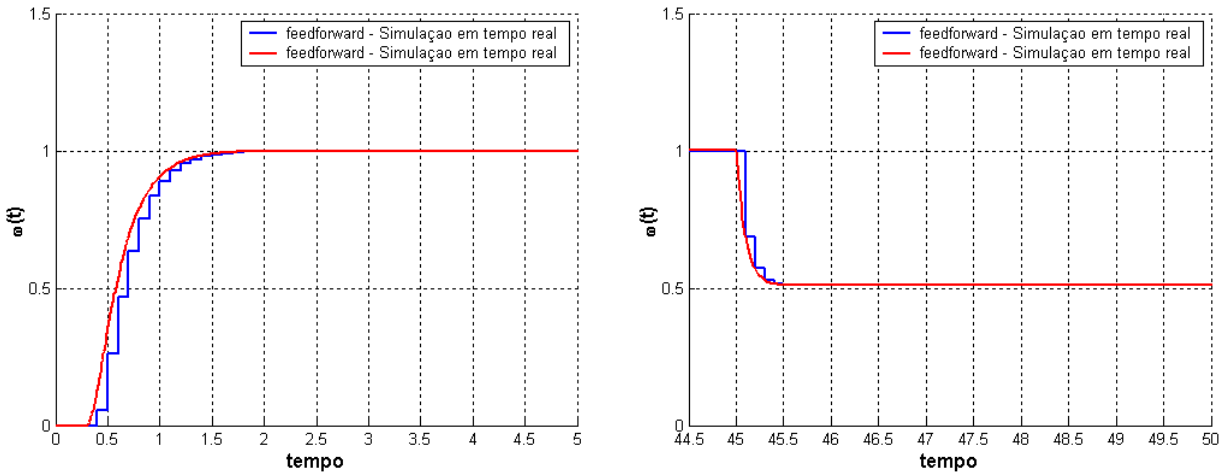


Figura 7.10 –ω(t), para a abordagem de controle FeedForward. Os dados amostrados ω(kT) em azul são aplicados a um ZOH. O problema ilustrado na figura 7.9 permite apenas que se detecte algum erro de implementação mais grosseiro. Os modelo parecem concordar, embora a resolução do Simulink tenha um retardo menor.

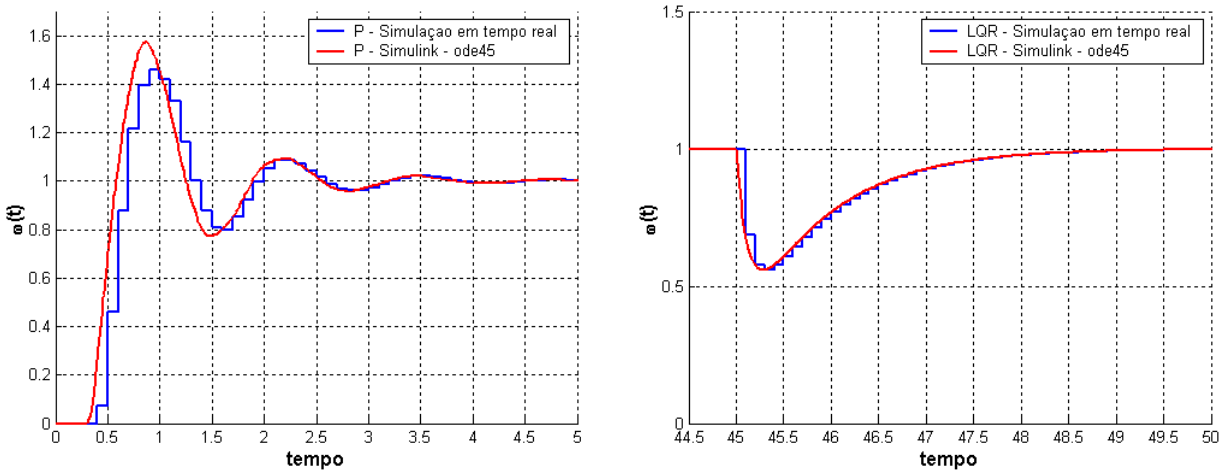


Figura 7.11 –ω(t), para a abordagem de controle Integral. Os dados amostrados ω(kT) em azul são aplicados a um ZOH. O problema ilustrado na figura 7.9 permite apenas que se detecte algum erro de implementação mais grosseiro.

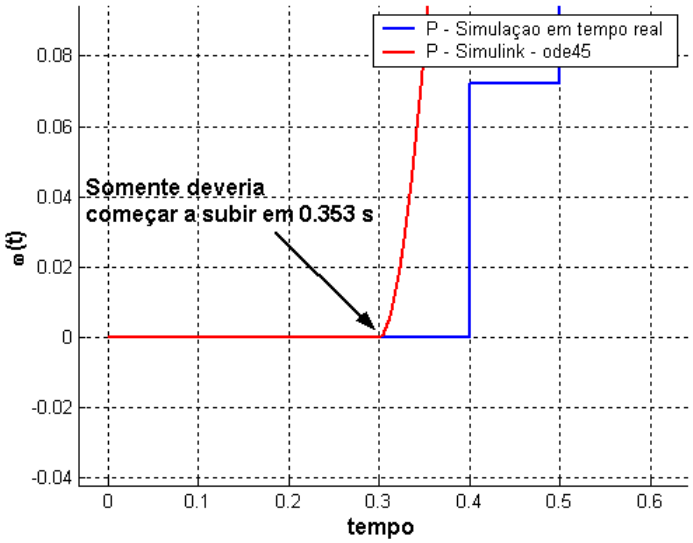


Figura 7.12 – Problema com o bloco do Simulink Variable Delay. O primeiro registro de atraso indica 0.353 segundos antes que a entrada V(kT) fosse atualizada para um valor diferente de zero.

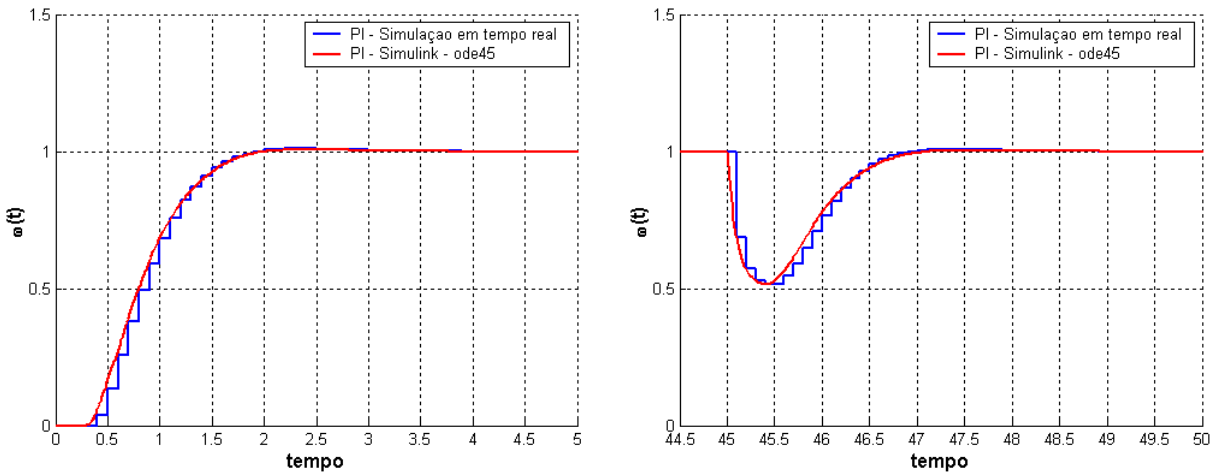


Figura 7.13 –ω(t), para a abordagem de controle PI. Os dados amostrados ω(kT) em azul são aplicados a um ZOH. O problema ilustrado na figura 7.9 permite apenas que se detecte algum erro de implementação mais grosseiro

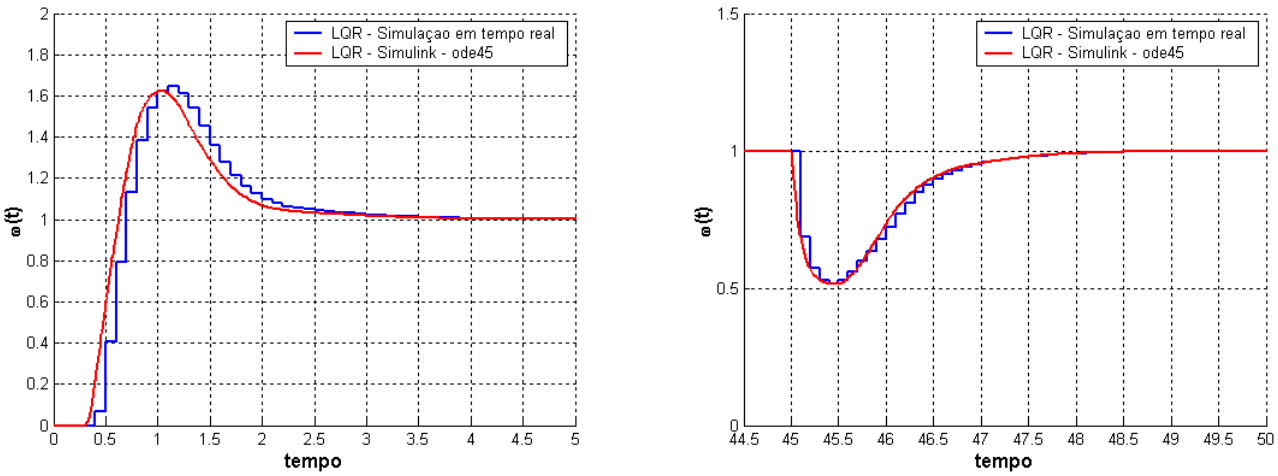


Figura 7.14 – ω(t), para a abordagem de controle LQR. Os dados amostrados ω(kT) em azul são aplicados a um ZOH. O problema ilustrado na figura 7.9 permite apenas que se detecte algum erro de implementação mais grosseiro.

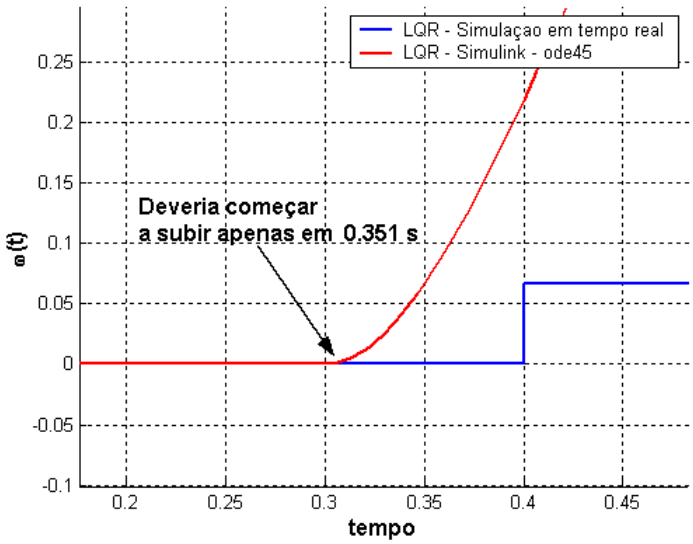


Figura 7.15 – Problema detectado novamente para o caso da figura 7.14. O primeiro registro de atraso indica que o sinal de controle somente deveria ser atualizado em 0.351 segundos.

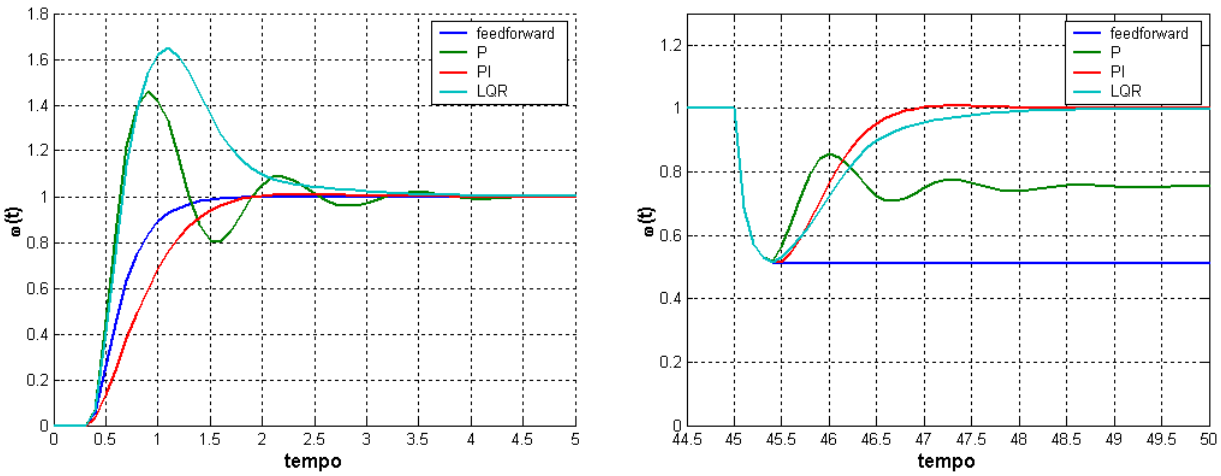


Figura 7.16 –ω(t), para a abordagem para as várias abordagens de controle, no cenário 2..

De maneira geral, o desempenho das abordagens de controle nesses dois cenários está de acordo com o que foi apresentado na seção 5.3.2.1, em que se assumia que o atraso era estático (por exemplo: 100 ms e 300 ms). Na figura 7.15 o resultado foi obtido para um atraso variante no tempo, com distribuição dada pelas figuras 7.40 e 7.30, apresentadas no final deste capítulo.

7.2 – Resultados na simulação do modelo do pêndulo incluindo a dinâmica do motor DC

7.2.1 – Resposta no tempo, sem malha de controle

Vamos simular em tempo real o sistema do pêndulo invertido para as condições iniciais $\theta_T = 0, \theta_P = 5.4^o, \dot{\theta}_T = 0, \dot{\theta}_P = 0$, utilizando o método de Runge Kutta de 4ª ordem com passo de tempo fixo de 500 μs. Com a ajuda do Simulink, vamos comparar os resultados de simulação em tempo real com simulações utilizando métodos numéricos de passo variável. No sistema do pêndulo não será aplicado nenhum sinal de tensão de entrada no sistema, de forma que este deverá oscilar livremente até convergir para o ponto de equilíbrio estável.

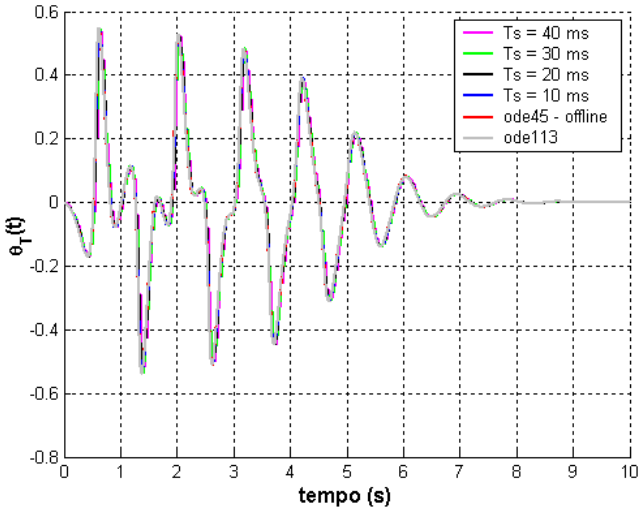


Figura 7.17 – Comparação entre as simulações em tempo real e utilizando o Simulink, com os método ode45 e ode113. Resposta θ_T (kT).

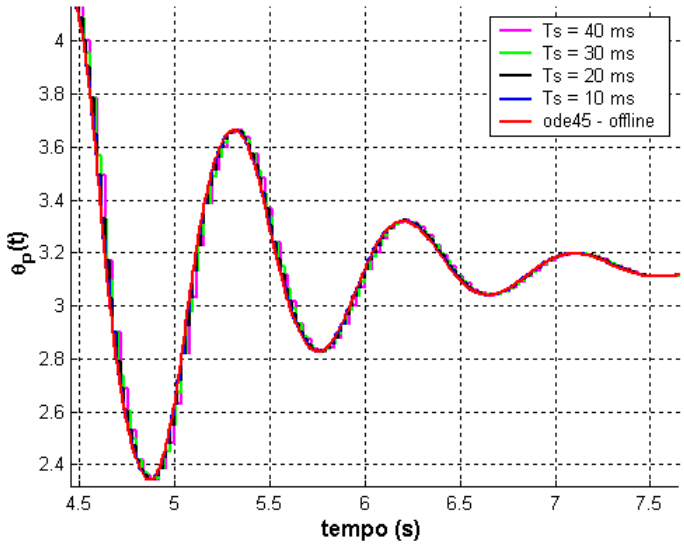


Figura 7.18 – Comparação entre as simulações em tempo real e utilizando o Simulink, com os método ode45 e ode113. Resposta θ_P (kT)

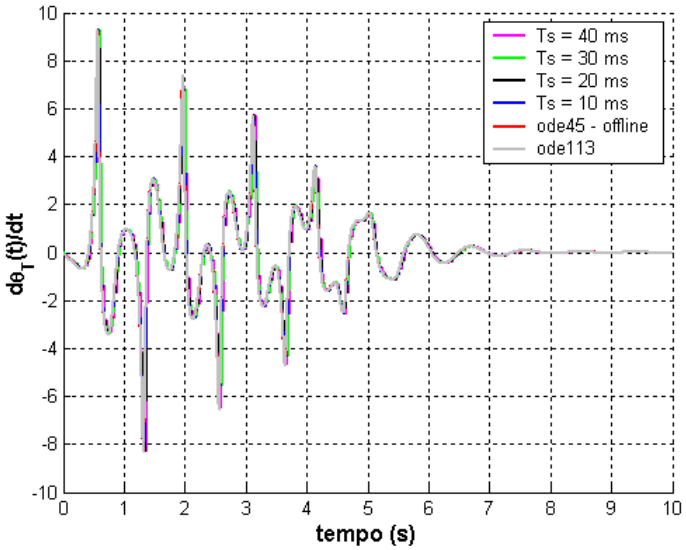


Figura 7.19 – Comparação entre as simulações em tempo real e utilizando o Simulink, com os método ode45 e ode113. Resposta θ_T (kT)

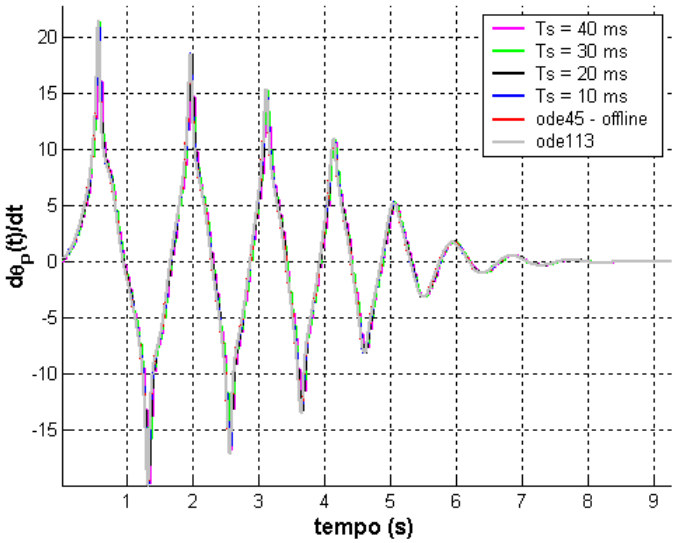


Figura 7.20 – Comparação entre as simulações em tempo real e utilizando o Simulink, com os método ode45 e ode113. Resposta θ_P (kT)

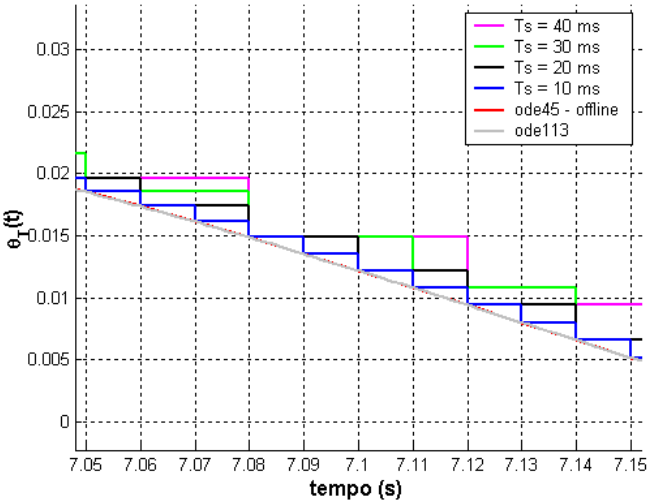


Figura 7.21 – Ampliando a figura 7.20, para visualizar os vários passos de amostragem.

Os resultados apresentados nas figuras 7.17 a 7.21 indicam que os resultados da simulação em tempo real e do Simulink estão em concordância.

7.2.2 – Controle Estabilizante do Pêndulo Invertido

O resultados obtidos com o simulador em tempo real serão comparados aos resultados do Simulink. No Simulink vamos utilizar o bloco Variable Delay para considerar os atrasos de comunicação que foram registrados. Assim poderemos avaliar a estabilidade numérica do simulador em tempo real quando comparado aos resultados obtidos no Simulink. No entanto, os resultados serão divididos em 2 grupos associado a 2 cenários distintos. No primeiro cenário, a comunicação *Bluetooth* é utilizada somente para monitorar os dados, logo o atraso na atualização da lei de controle deve ser muito pequeno, da ordem do período de atualização do próprio simulador em tempo real. Em um segundo cenário, a comunicação via *Bluetooth* será utilizada para o transporte de dados dos sensores e de controle, caracterizando um sistema de controle distribuído, onde o atraso de ida e volta no transporte de um pacote de dados passa a interferir no desempenho do sistema de controle.

Cenário 1. Controle feito pela máquina que hospeda o servidor *Bluetooth*, e o simulador em tempo real. Serão utilizados na simulação os períodos de amostragem de 10 ms, 20 ms e 40 ms. A figura 7.18 apresenta os registros os atrasos de comunicação, que são no mínimo iguais ao período de execução do módulo do RTLinux/Free. Como a comunicação via *Bluetooth* não é utilizada no sistema de controle, e o controle é feito na mesma máquina, espera-se pouco retardo entre o instante que se obtém uma nova amostra dos sensores e o instante em que o atuador é atualizado com um novo sinal de controle.

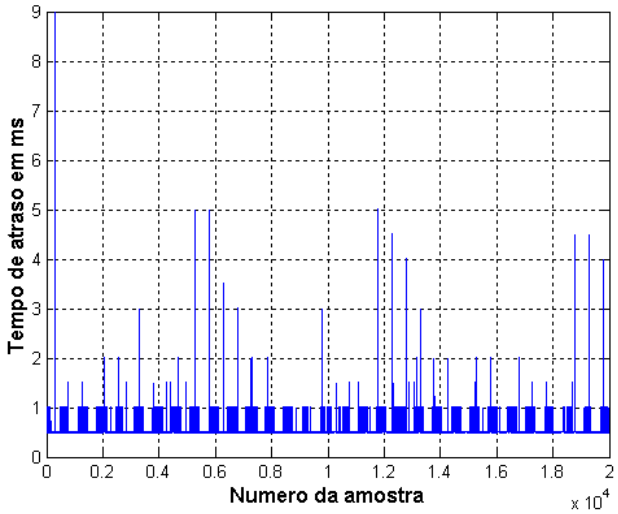


Figura 7.22 - Amostras do atraso de comunicação em ms. Para um período de amostragem de 10 ms.

Cenário 1: Controle na máquina local

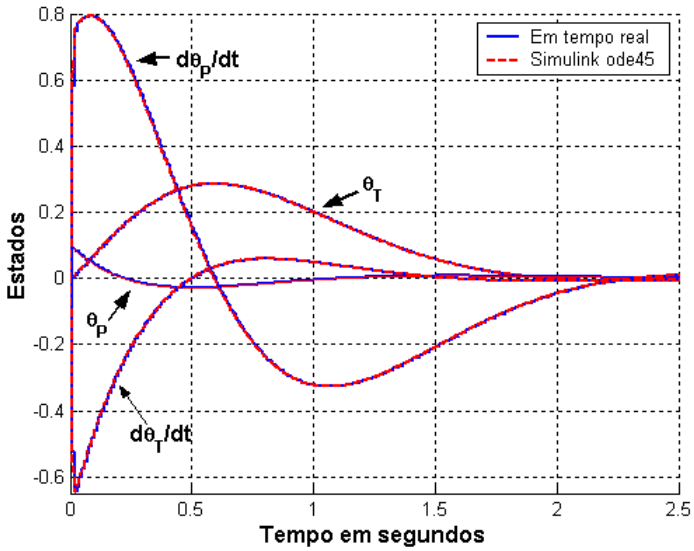


Figura 7.23 – Condições Iniciais $\theta_T = 0, \theta_P = 5.4^\circ, \dot{\theta}_T = 0, \dot{\theta}_P = 0$.

Período de Amostragem = 10 ms e **K** = [-0.9880 -17.2081 -1.3633 -3.0133].

Os dados gerados pela simulação em tempo real e pelo Simulink estão em concordância.

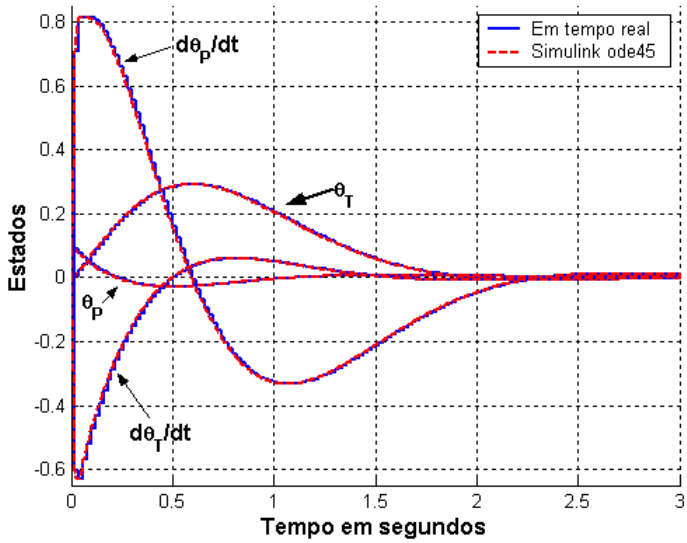


Figura 7.24 – Condições Iniciais $\theta_T = 0, \theta_P = 5.4^o, \dot{\theta}_T = 0, \dot{\theta}_P = 0$.

Período de Amostragem = 20 ms e **K** = [-0.6062 -11.8484 -1.0495 -1.9461]

Os dados gerados pela simulação em tempo real e pelo Simulink estão em concordância.

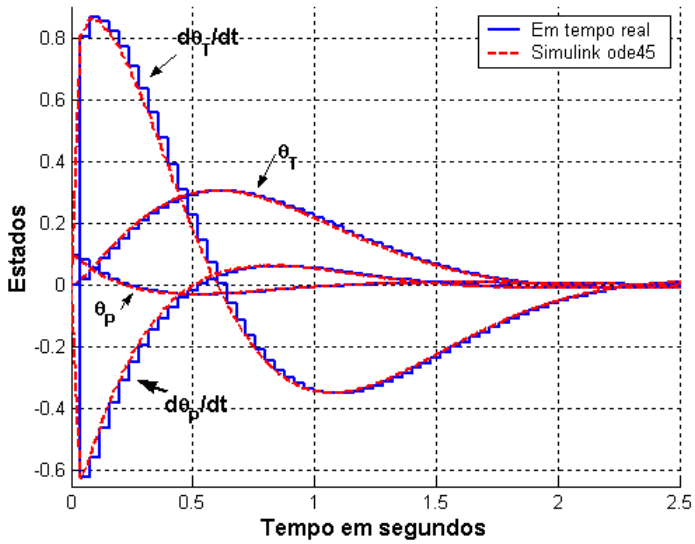


Figura 7.25 – Condições Iniciais $\theta_T = 0, \theta_P = 5.4^o, \dot{\theta}_T = 0, \dot{\theta}_P = 0$.

Período de Amostragem = 40 ms e **K** = [-0.3607 -8.6042 -0.8489 -1.2970]

Os dados gerados pela simulação em tempo real e pelo Simulink estão em concordância.

Cenário 2: Controle Distribuído, com comunicação via *Bluetooth*. Serão utilizados na simulação os períodos de amostragem de 10 ms, 20 ms e 40 ms. A figura 7.26 apresenta os registros de uma simulação em tempo real: os atrasos de comunicação são no mínimo iguais ao período de execução do módulo do RTLinux/Free. Como estamos utilizando a comunicação via Bluetooth, entre computadores remotos, para trocar os dados de controle, espera-se que o atraso de ida e volta de um pacote seja significativo e altere o desempenho do sistema de controle.

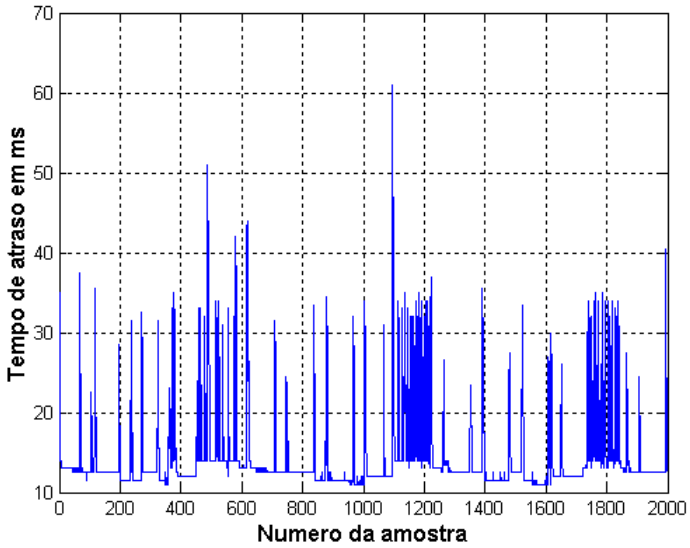


Figura 7.26 – Amostras do atraso de comunicação *Bluetooth* em ms, para um período de amostragem de 10 ms.

Cenário 2 : Controle Distribuído

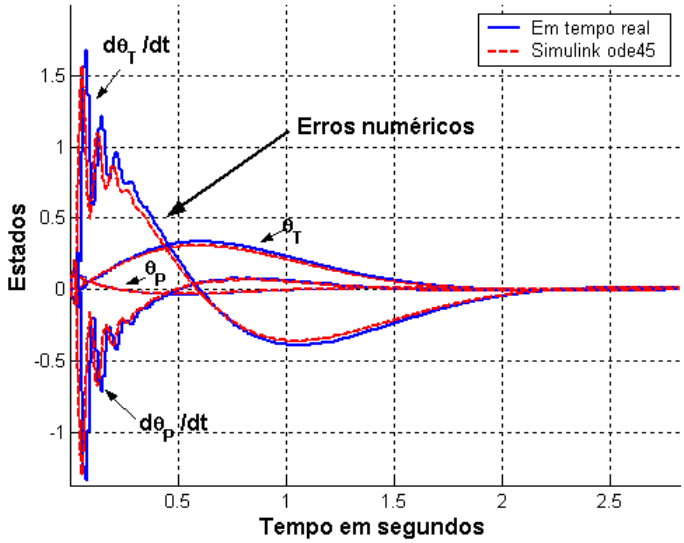


Figura 7.27 – Condições Iniciais $\theta_T = 0, \theta_P = 5.4^o, \dot{\theta}_T = 0, \dot{\theta}_P = 0$

Período de Amostragem = 10 ms e **K** = [-0.9880 -17.2081 -1.3633 -3.0133].

Os dados gerados pela simulação em tempo real e pelo Simulink não estão em concordância. Nesse caso, deve-se investigar qual seria o passo fixo de tempo necessário para que as simulações não comecem a divergir.

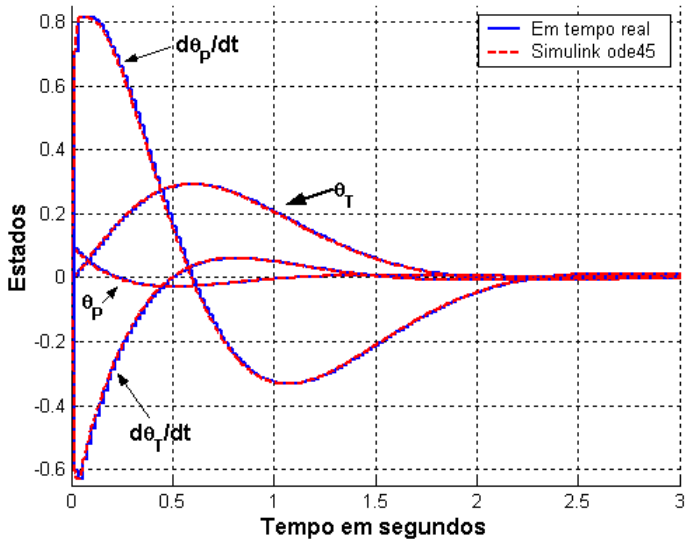


Figura 7.28 – Condições Iniciais $\theta_T = 0, \theta_P = 5.4^o, \dot{\theta}_T = 0, \dot{\theta}_P = 0$.

Período de Amostragem = 20 ms e **K** = [-0.6062 -11.8484 -1.0495 -1.9461]

Os dados gerados pela simulação em tempo real e pelo Simulink estão em concordância.

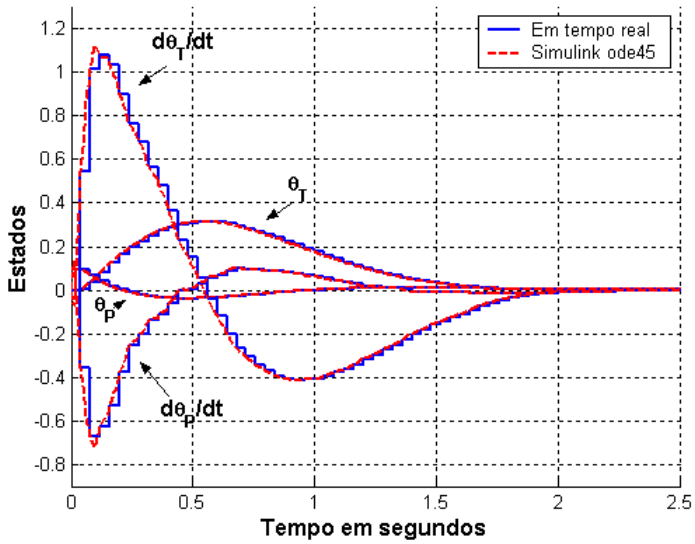


Figura 7.29 – Condições Iniciais $\theta_T = 0, \theta_P = 5.4^o, \dot{\theta}_T = 0, \dot{\theta}_P = 0$.

Período de Amostragem = 40 ms e **K** = [-0.3607 -8.6042 -0.8489 -1.2970]

Os dados gerados pela simulação em tempo real e pelo Simulink estão em concordância.

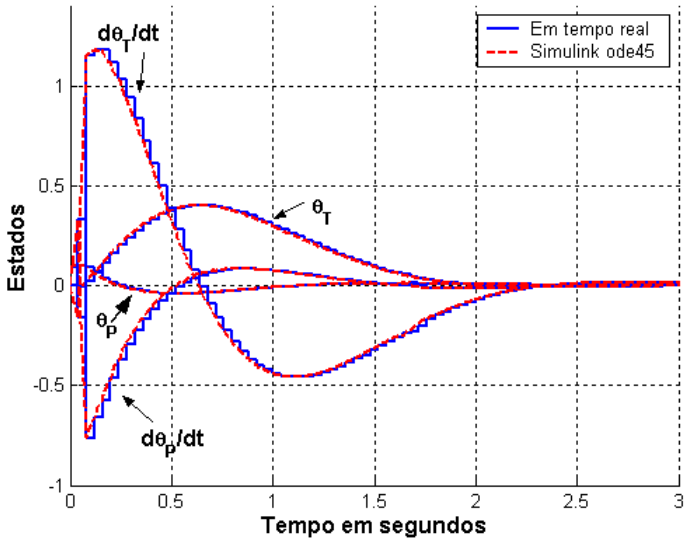


Figura 7.30 – Condições Iniciais $\theta_T = 0, \theta_P = 5.4^\circ, \dot{\theta}_T = 0, \dot{\theta}_P = 0$.

Período de Amostragem = 40 ms e **K** = [-0.3607 -12.3739 -1.2664 -1.7141 0.7508]

Os dados gerados pela simulação em tempo real e pelo Simulink estão em concordância.

7.3 – Medidas do tempo de atraso

Nesse seção, vamos apresentar alguns dados relacionados ao atraso de comunicação. A seguir, serão apresentados alguns motivos para a variabilidade deste atraso por causa do sistema operacional Linux utilizado e por causa do tipo de transporte lógico ACL.

O sistema operacional Linux pode tornar imprevisível o tempo de execução de certas tarefas, que impedem a execução de tarefas prioritárias. Por exemplo, se utilizarmos um programa como o Octave (similar ao Matlab) para manipular matrizes muito grandes, e isso acarretar no uso de Swap, o tempo necessário para que uma tarefa prioritária execute pode ser longo, devido ao acesso ao disco por um processo de menor prioridade (no caso o Octave).

Na comunicação via Bluetooth, utilizar o transporte lógico ACL para transportar um fluxo de dados significa que: os recursos do sistema de comunicação são compartilhados ou são cedidos a outros tipos de fluxo de dados de maior prioridade. Por exemplo, o comando “hcitool inq” do Bluez coloca o módulo Bluetooth para procurar por outros dispositivos, em uma operação de maior prioridade que consome muitos dos recursos para a transmissão de dados via Bluetooth.

7.3.1 – Histogramas de tempo de atraso

- Histogramas do atraso de comunicação, quando há concorrência com 1 processo concorrente pela utilização da comunicação Bluetooth na mesma máquina. No caso, será utilizado uma ferramenta do Bluez chamada de l2test, que serve para verificar a comunicação Bluetooth sobre o protocolo L2CAP, enviando indefinidamente pacotes de dados de um computador ao outro via Bluetooth.

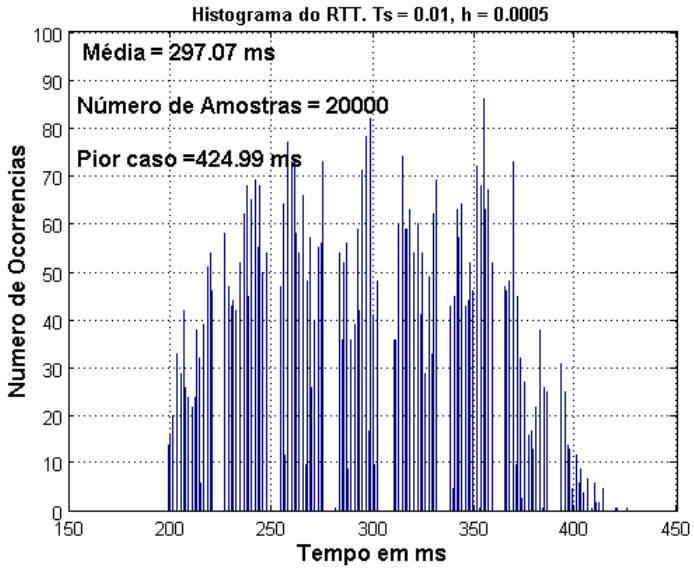


Figura 7.31 - Histograma do Atraso de Comunicação para uma taxa de 100 amostras/s, utilizando o comando l2test do Bluez para gerar um tráfego concorrente no enlace Bluetooth.

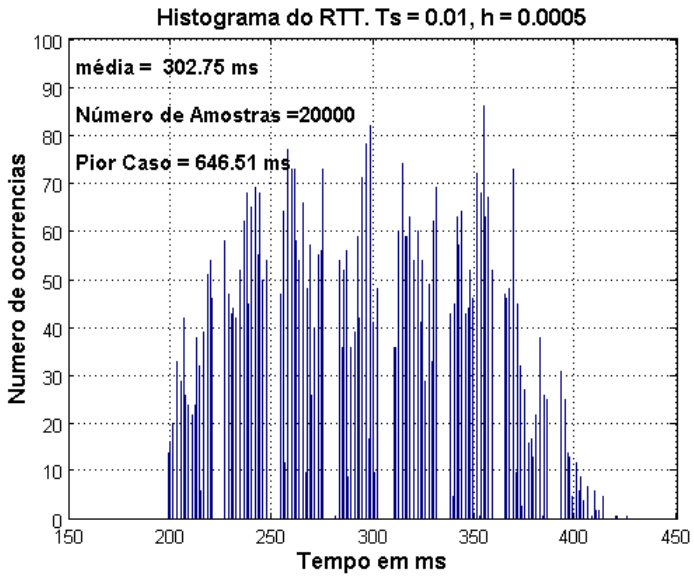


Figura 7.32 - Histograma do Atraso de Comunicação para uma taxa de 25 amostras/s, utilizando o comando l2test do Bluez para gerar um tráfego concorrente no enlace Bluetooth.

- Histogramas do atraso de comunicação, quando há processos do sistema operacional Linux utilizando Swap e/ou utilizando comandos para colocar o módulo Bluetooth em busca de outros dispositivos (Inquiry). A figura 7.27, ilustra como a busca por outros dispositivo aumenta o atraso de comunicação.

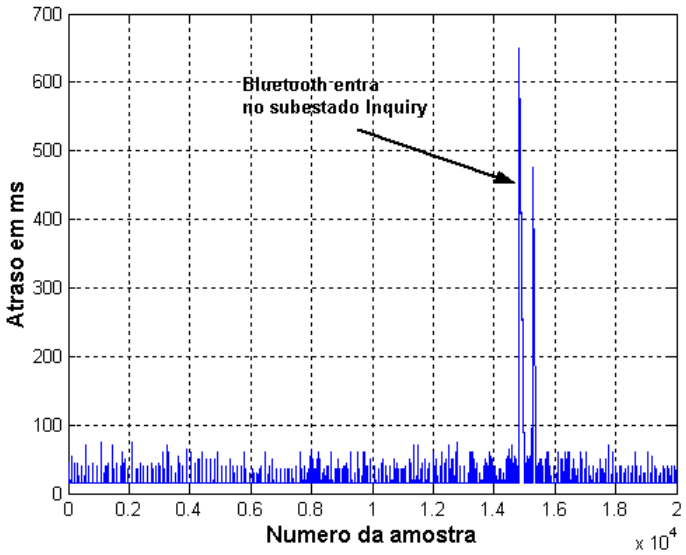
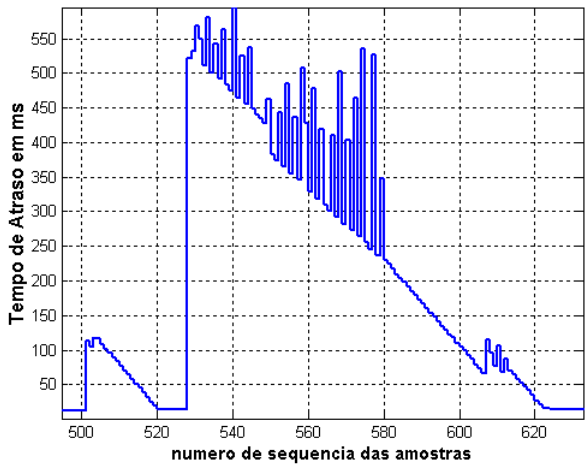
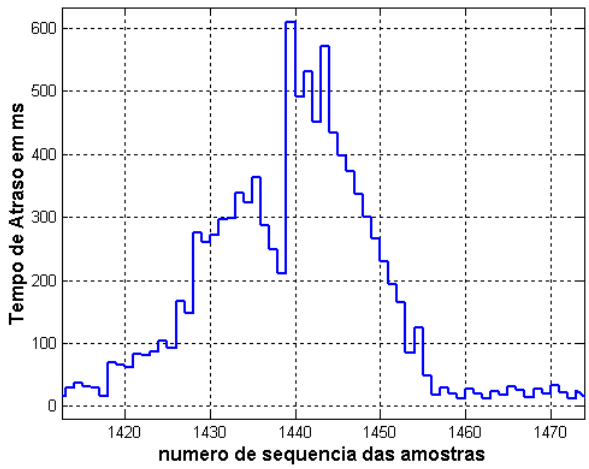


Figura 7.33 – Amostras do Atraso de Comunicação para uma taxa de 100 amostras/s, colocando o módulo Bluetooth no subestado Inquiry.



(a) Amostras de atraso, taxa de 100 amostras/s



(b) Amostras de atraso, taxa de 25 amostras/s

Figura 7.34 – Os piores valores de atrasos ocorrem em rajadas, e não de forma independente.

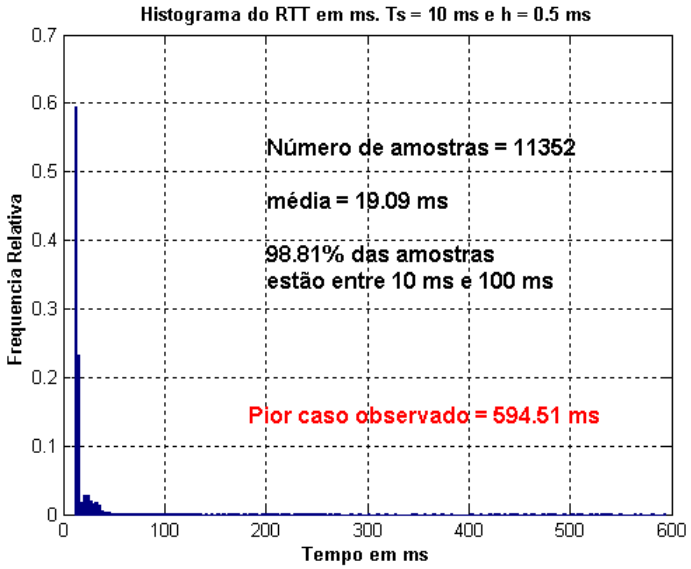


Figura 7.35 -Histograma do Atraso de Comunicação para uma taxa de 100 amostras/s, com carga no sistema Linux e colocando o módulo Bluetooth no subestado Inquiry.

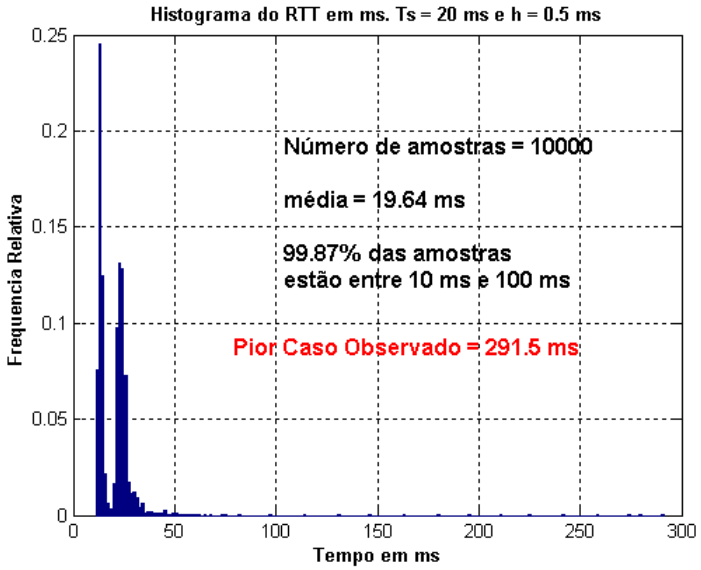


Figura 7.36 - Histograma do Atraso de Comunicação para uma taxa de 50 amostras/s, com carga no sistema Linux e colocando o módulo Bluetooth no subestado Inquiry.

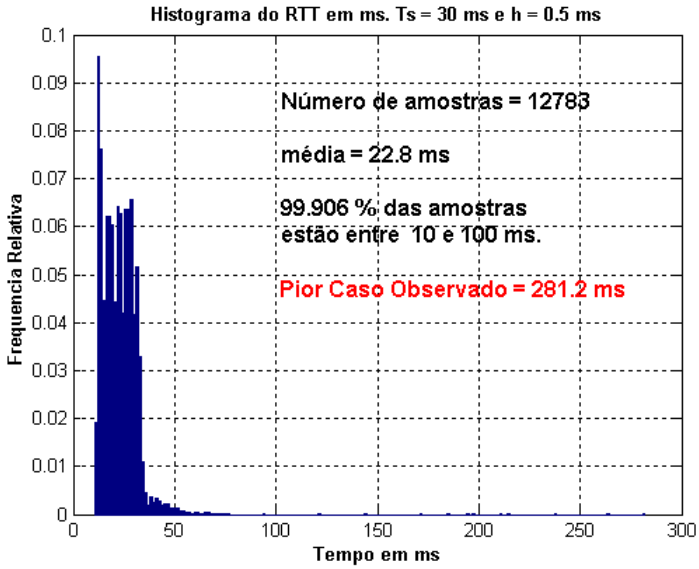


Figura 7.37 - Histograma do Atraso de Comunicação para uma taxa de 33.3 amostras/s, com carga no sistema Linux e colocando o módulo Bluetooth no subestado Inquiry.

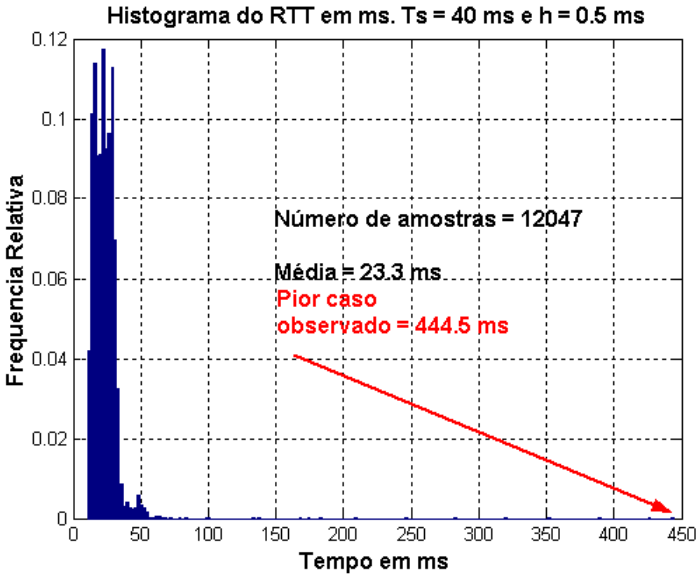


Figura 7.38 - Histograma do Atraso de Comunicação para uma taxa de 25 amostras/s, com carga no sistema Linux e colocando o módulo Bluetooth no subestado Inquiry.

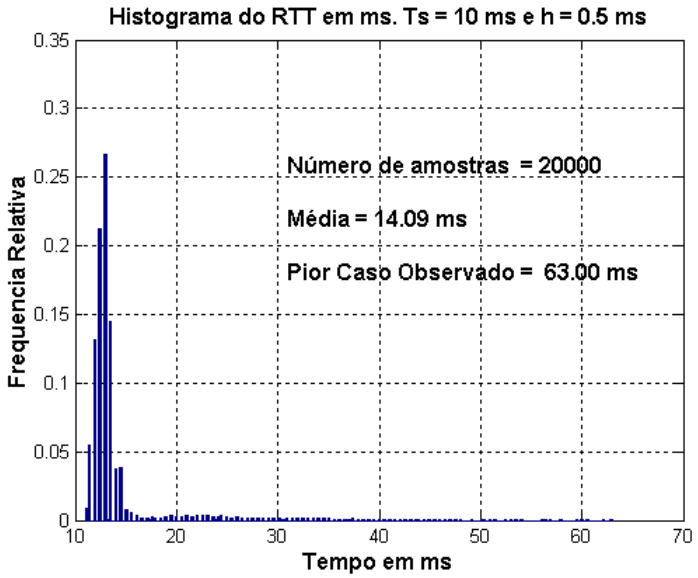


Figura 7.39 - Histograma do Atraso de Comunicação para uma taxa de 100 amostras/s, com pouca carga no sistema Linux e enlace Bluetooth dedicado a comunicação de controle.

- Histogramas do atraso de comunicação, quando há pouca carga no sistema operacional e um enlace Bluetooth dedicado a comunicação Bluetooth;

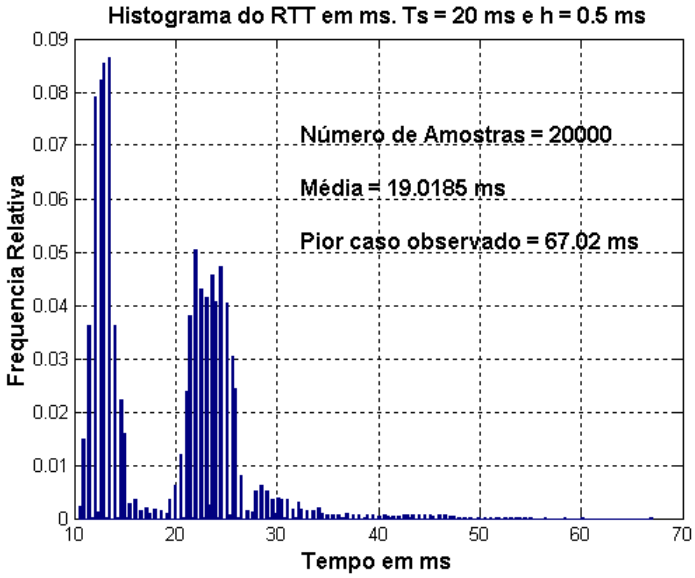


Figura 7.40 - Histograma do Atraso de Comunicação para uma taxa de 50 amostras/s, com pouca carga no sistema Linux e enlace Bluetooth dedicado a comunicação de controle.

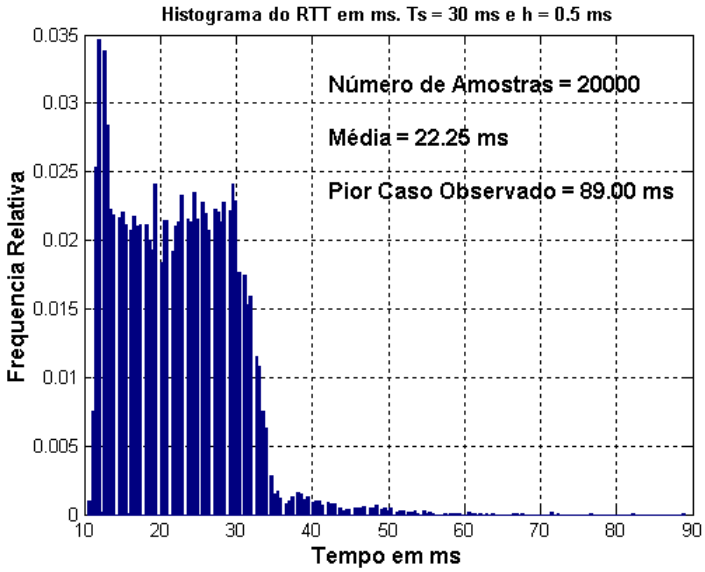


Figura 7.41 - Histograma do Atraso de Comunicação para uma taxa de 33.3 amostras/s, com pouca carga no sistema Linux e enlace Bluetooth dedicado a comunicação de controle.

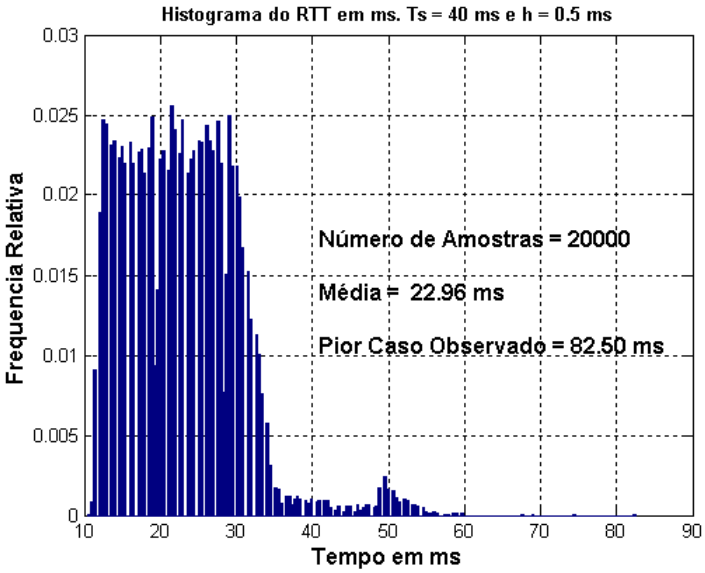


Figura 7.42 - Histograma do Atraso de Comunicação para uma taxa de 25 amostras/s, com pouca carga no sistema Linux e enlace Bluetooth dedicado a comunicação de controle.

CAPÍTULO 8

Conclusão e Trabalhos Futuros

8.1 – Conclusão

O sistema de comunicação *Bluetooth* entre dois computadores pessoais que utilizam o sistema operacional Linux foi implementado corretamente, utilizando-se do modelo de Cliente e Servidor. Pode-se transmitir de um computador ao outro os parâmetros de configuração do simulador e iniciar uma simulação em tempo real. Os pacotes da aplicação, com estampa de tempo, servem para transportar informações como as variáveis de estado, sinais de controle e o tempo de partida, o que facilita a obtenção de medidas importantes, como o atraso de ida e volta de um pacotes de dados.

As tarefas em tempo real do RTLinux cumprem a sua função de emular um sistema dinâmico com o mesmo comportamento no tempo, e isso pode ser observado através da comparação dos resultados da simulação em tempo real e das simulações realizadas no Simulink. Como os resultados são idênticos, salvo quando temos instabilidade numérica, pode-se concluir que a simulação em tempo real sempre executa próxima do tempo correto. Isso também pode ser comprovado através dos histogramas e gráficos de erro no tempo de execução, apresentados no capítulo 3.

No primeiro estudo de caso, foi possível planejar sistemas de controle com uma margem ao atraso razoável, que resultou em um projeto de um sistema de controle robusto a atrasos de comunicação que variam numa faixa entre 10 ms e 700 ms, aproximadamente. É claro que essa maior robustez significou que o sistema teria uma resposta mais lenta, pois o sistema fica superamortecido. Mostrou-se que embora outras abordagens de controle tenham uma resposta mais rápida, com o controle proporcional ou a malha aberta, foi necessário utilizar um integrador no compensador para rejeitar a perturbação de baixa freqüência e tornar o sistema superamortecido para obter as margens ao atraso desejadas.

No caso do pêndulo invertido, haviam muitas restrições em se implementar os algoritmos de controle. O controle de SwingUp do pêndulo é um problema em aberto quando temos atrasos de comunicação variantes no tempo na malha e o controle estabilizante somente pôde ser aplicado no cenário mais restritivo do Bluetooth, em que temos pouca ou nenhuma interferência na transmissão de dados e um atraso de comunicação limitado a valores de apenas algumas dezenas de ms.

No capítulo 7, os histogramas acerca do atraso de ida e volta de um pacote indicam cenários distintos que podem tornar viável ou não algumas das estratégias de controle. Também é possível observar nos histogramas que quando as taxas são maiores, parece haver menor atraso de comunicação. Isso pode ser interpretado como se a prioridade modificada dos processos do Linux, e a maior demanda por transmitir dados, reduzisse o tempo em que outros processos do Linux executassem, reduzindo a variabilidade do atraso e o valor médio desses atrasos. Por outro lado, foi demonstrado que quanto maior a freqüência de amostragem, maior deve ser a rajada e o valor da

atraso em caso de interferência na comunicação, pois muitos pacotes ficarão acumulados em espera em fila.

8.2 – Trabalhos Futuros

Como trabalhos futuros podemos indicar outras investigações feitas na referência em [5], em que se propõe a utilização de transmissões de dados via Bluetooth em que não se garante a integridade dos dados e não são feitas retransmissões, possivelmente utilizando conexões SCO da camada *Baseband* do *Bluetooth*, ao invés de conexões ACL. Nesse caso, propõe-se o projeto de observadores para estimar um valor esperado e assim detectar dados corrompidos nos pacotes.

Outro trabalho a ser feito é implementar o cliente em um sistema computacional portável, como por exemplo um PDA ou um celular, de forma a verificar o desempenho do sistema de comunicação Bluetooth entre dispositivos de menor capacidade computacional.

Também seria interessante modelar o atraso de ida e volta de um pacote de dados transmitidos pelo sistema de comunicação *Bluetooth*. Por exemplo, considerando que o atraso de comunicação seja uma variável aleatória, poderíamos utilizar métodos de estimação estatística e propor uma distribuição que melhor representasse os valores observados de tempos atrasos de ida e volta. Modelo mais sofisticados poderiam utilizar processos estocásticos, como por exemplo Cadeias de Markov, para modelar o atraso de comunicação.

Por último, podemos citar trabalhos como [47]e [48], no qual outras tecnologias de comunicação sem fio são investigadas e comparadas, como a tecnologia IEEE802.11 e o *Bluetooth*.

Referências

[1] BILTRUP, U.; WIBERG, P.A. *Bluetooth in industrial environment. In Proc 2000 IEEE International Workshop on Factory Communication Systems*, pág. 239-247, Porto, Portugal, set., 2000.

[2] ANDERSSON, M.; *Industrial Use of Bluetooth.* Disponível em: <http://www.connectblue.com/>. Acesso em: 5 Janeiro de 2005.

[3] ANDERSSON, *M.; Intelligent Industrial Automation Devices using Bluetooth.* Disponível em: < http://www.connectblue.com/>. Acesso em: 5 Janeiro de 2005.

[4] EKER, J., CERWIN, A. e HORJEL, A.; *Distributed Wireless Control Using Bluetooth.* IFAC Conference on New Technologies for Computer Control, Hong Kong, China, Nov., 2001.

[5] HORJEL, A. *Bluetooth in Control. Master Thesis, Department of Automatic Control, Lund Institute of Technology*, Suécia, jan., 2001. Disponível em: <http://www.control.lth.se/publications/>. Acesso em: 25 Janeiro de 2005.

[6] BLUETOOTH SIG. *Specification of the Bluetooth System Version 1.1 – Core.* v.1, fev., 2001. Disponível em: < http://www.bluetooth.org/>. Acesso em: 5 Janeiro de 2005.

[7] BILLO, E.A. **Uma pilha de protocolos Bluetooth adaptável a aplicação**. Projeto Final de Graduação, UFSC, Florianópolis, SC, Brasil, fev., 2003.

[8] BLUETOOTH SIG. *Qualification Program Reference Document.* fev. , 2002. Disponível em: <http://www.bluetooth.org/>. Acesso em: 5 Janeiro de 2005.

[9] BLUETOOTH SIG. *Specification of the Bluetooth System Version 1.1 – Profiles.* v.2, fev., 2001. Disponível em: < http://www.bluetooth.org/>. Acesso em: 5 Janeiro de 2005.

[10] MILLER, M. **Descobrindo Bluetooth**. Editora Campus Ltda, Brasil, Rio de Janeiro, 2001. p. 4-82.

[11] BLUETOOTH SIG. Specification. *The Official Bluetooth Membership Site.* Disponível em: <http://www.bluetooth.org/spec/>:. Acesso em: 5 de Janeiro de 2005.

[12] ANDERSSON, M.; *IEEE802.11b and Bluetooth in an Industrial Environment.* Disponível em: < http://www.connectblue.com/>. Acesso em: 5 Janeiro de 2005.

[13] POINEXTER, S.E; Heck, B.S. *Using the Web in Your Courses: What can you do? What Should you do?. IEEE Control Systems Magazine*, fev, 1999.

[14] OVERSTREET, J. W.; TZES, A. *An Internet based real-time control engineering laboratory, IEEE Control Systems Magazine*, 19, 1999. n.º 5, p. 19-34.

[15] NILSSON, J. *Real-Time Control Systems with Delays. Master Thesis, Department of Automatic Control, Lund Institute of Technology*, Suécia, jan., 1998. p. 8-20.

[16] TOVAR, E.; VASQUES, F. *Real-Time Fieldbus Communications Using Profibus Networks.* IEEE TRANSACTIONS ON INDUSTRIAL ELECTRONICS, vol. 46, n.6, dez., 1999.

[17] WILLIG, A. Analysis of the Profibus Token Passing Protocol over Wireless Links. Proc. 2002 IEEE International Symposium on Industrial Electronics, L'Aquila, Italy, July 2002.

[18] DRAKE, W.J. *The Distributed Information Revolution and the Global Information Society.* Disponível em : <http://usinfo.state.gov/products/pubs/archive/telecomm/drake.htm>. Acesso em: 9 de ago. 2004.

[19] KUROSE, J.F; ROSS, K.W. **Redes de Computadores e a Internet – Uma Nova Abordagem**. Addison Wesley, 1ª ed, São Paulo, 2003. cap. 2 - 5.

[20] TANENBAUM, A. S. **Redes de Computadores.** Editora Campus, 3ª ed., Rio de Janeiro, 1997. p. 32 - 44.

[21] SACKS, A.G. **Sistema de Gerenciamento de Redes e Processos através de Computadores Portáteis Via Bluetooth.** Projeto Final de Graduação, UFRJ, Rio de Janeiro, RJ, Brasil, mar., 2003.

[22] BLUETOOTH SIG. *Specification of the Bluetooth System Version 1.1 – Core***.** v.1, fev., 2001. p. 126 e p. 146. Disponível em: < http://www.bluetooth.org/>. Acesso em: 5 Janeiro de 2005.

[23] BLUETOOTH SIG. *Specification of the Bluetooth System Version 1.1 – Core***.** v.1, fev., 2001. p. 65. Disponível em: < http://www.bluetooth.org/>. Acesso em: 5 Janeiro de 2005.

[24] USB Implementers Forum. *Universal Serial Bus Specification Revision* 1.1, Set. ,1998. P. 15 – 58. Disponível em : <http://www.usb.org/developers/docs/>. Acesso em 20 Janeiro de 2005.

[25] Zlenovsky, R. e Mendonça, A. PC: Um Guia Prático de *Hardware* e Interfaceamento. MZ Editora Ltda, 2ª ed, Rio de Janeiro, 1999. p. 681-689.

[26] BLUETOOTH SIG. *Specification of the Bluetooth System Version 1.1 – Core*. v.1, fev., 2001. p. 781-796. Disponível em: <http://www.bluetooth.org/>. Acesso em: 5 Janeiro de 2005.

[27] HOLTMANN M. *Bluetooth hardware support for BlueZ.* Disponível em: <http://www.holtmann.org/linux/bluetooth/devices.html>. Acesso em: 20 Jan. 2005.

[28] HOLTMANN M.. *Bluez - The Official Linux protocol Stack.* Disponível em: <http://www.bluez.org/download>. Acesso em: 5 Jan. 2005.

[29] KERNELORG ORGANIZATION. *The Linux Kernel Archives*. Disponível em: <http://www.kernel.org/pub/linux/kernel/v2.4/>. Acesso em: 20 Jan. 2005.

[30] YODAIKEN, V.; *The* **RTLinux** *Manifesto*. **FSMLabs**. Disponível em: <http://www.fsmlabs.com/articles/archive.html >. Acesso em: 22 Jan. 2005.

[31] SVENSSON, M.; *Hard Real-Time Control of an Inverted Pendulum using RTLinux/Free. Master Thesis, Department of Automatic Control, Lund Institute of Technology*, Suécia, fev., 2004. p. 1-15. Disponível em: <http://www.control.lth.se/publications/>. Acesso em: 25 Janeiro de 2005.

[32] BOVET, D. P.; CESATI, M.;. *Understanding the Linux Kernel*, O'Reilly, 2ª ed., Dez., 2002. cap. 3 e 11.

[33] YODAIKEN, V.; *FMSLabs RTLinux Technology for Hard Real Time*. **FSMLabs**. Disponível em: <http://www.fsmlabs.com/articles/archive.html >. Acesso em: 22 Jan. 2005.

[34] BARABANOV, M.; *A Linux-based Real-Time Operating System. Master Thesis*. New Mexico Institute of Mining and Technology, Socorro, New Mexico, 1997. Disponível em: <http://www.fsmlabs.com/articles/archive.html >. Acesso em: 22 Jan. 2005.

[35] FSMLabs (Finite State Machine Labs), Inc. RTLinuxFree. **FSMLabs**. Disponível em: < http://www.fsmlabs.com/products/openrtlinux/>. Acesso em: 10 jan. 2005.

[36] Wu, X.; Figueroa, H.; Monti, A. *Testing Digital Controllers Using Real-Time Hardware in the Loop Simulation*. 2004 IEEE Power Electronics Specialists Conference. Aachen, Alemanha, 2004. p.1 – 2.

[37] Wu, X. ; Figueroa, H. *Real-Time Hardware-In-the-Loop (HIL) Simulation*. REM Lab Projects. Disponível em : <http://vtb.engr.sc.edu/research/reml/projects/rthil.asp>. Acesso em: 12 fev. 2005.

[38] Roque, W.L. **Introdução ao Cálculo Numérico**. Ed. Atlas. 2000. P. 187- 207.

[39] Boyce, W.E e DiPrima, R.C. **Equações Diferenciais Elementares e Problemas de Valores de Contorno**. LTC. 6ª ed., 1998. P. 295- 316.

[40] QUANSER CONSULTING. *Self-Erecting, Rotary Motion Inverted Pendulum*. 1996.

[41] THE MATH WORKS. SISO *Example: the DC Motor:: Building Models*. 2005. Disponível em : <http://www.mathworks.com/access/helpdesk/help/toolbox/control/getstart/buildmo4.html>. Acesso em: 14 de fev. 2005.

[42]OGATA, K. **Engenharia de Controle Moderno.** Prentice Hall do Brasil, 3ª ed., Rio de Janeiro, 1998.

[43] FRANKLIN, G. F.; POWELL, J. D. e Workman, M. L. **Digital Control of Dynamic Systems**, Addison-Wesley Publishing Company, 3ª ed., New York, 1997.

[44] OGATA, K. **Discrete-Time Control Systems**, *Prentice Hall*, 2ª ed., New Jersey, 1994.

[45] CERVIN, A.; Lincoln, B.; Eker, J.; Arzen, K.; Buttazzo, G.. The Jitter Margin and Its Application in the Design of Real-Time Control Systems. **RTCSA**, Gothenburg, Ago., 2004

[46] TANENBAUM, A.S.,WOODHULL,A.S. **Sistemas Operacionais: Projeto e Implementação**. Bookman, 2ª ed., Porto Alegre, 2000. Pág. 70.

[47] PLOPLYS, N. J., KAWKA, P. A., ALLEYNE, A. A., "Closed-Loop Control over Wireless Networks", **IEEE Control Systems Magazine**, Vol. 24, No. 3, June 2004, pp. 58-71

[48] PLOPLYS, N. J., ALLEYNE, A. A., UDP Network Communications for Distributed Wireless Control, **Proceedings of the American Control Conference**, Denver, Colorado June 4-6, 2003

Apêndice A

A1 – Lista de Programas implementados

Arquivos .c

Nome do Arquivo	Descrição
segunda_ordem.c	Módulo do RTLinux/Free que executa periodicamente a cada 1 ms. Utiliza o método de Runge Kutta de 4ª ordem. Simula um sistema dinâmico de 2ª ordem
Pendulo.c	Módulo do RTLinux/Free que executa periodicamente a cada 0.5 ms. Utiliza o método de Runge Kutta de 4ª ordem. Simula um sistema dinâmico de um pêndulo
blue_com_servidor.c	Esse arquivo contêm as funções necessárias para criar um servidor que utiliza a comunicação Bluetooth.
servidor_segunda_ordem.c	É um arquivo principal que inicializa o servidor Bluetooth e espera por conexões para iniciar a simulação do sistema de 2ª ordem
servidor_pendulo.c	É um arquivo principal que inicializa o servidor Bluetooth e espera por conexões para iniciar a simulação do sistema do pêndulo invertido
gravador_servidor_pendulo.c	É um arquivo principal que lê as RT-Fifos, registra os dados em arquivos e imprime estes em um gráfico, utilizando o Gnuplot
plotter.c	Esse arquivo contém as funções que o gravador_servidor_pendulo ou gravador_servidor_segunda_ordem utilizam para registrar os dados e imprimir em um gráfico
blue_com_cliente.c	Esse arquivo contém as funções necessárias para iniciar uma conexão com computadores remotos, através da comunicação sem fio *Bluetooth*
cliente_segunda_ordem.c	É o arquivo principal para um cliente que utiliza o Bluetooth para inicializar a simulação de um sistema de 2ª ordem em um computador remoto, ajustando o período de amostragem, ganhos da lei de controle, estabelece se o controle será distribuído ou não.
cliente_pendulo.c	É o arquivo principal para um cliente que utiliza o Bluetooth para inicializar a simulação de um sistema de um pêndulo invertido em um computador remoto, ajustando o período de amostragem, ganhos da lei de controle, estabelece se o controle será distribuído ou não.

Arquivos .h

Nome do arquivo	Descrição
blue_com_servidor.h	Contém os protótipos das funções de blue_com_servidor.c
blue_com_cliente.h	Contém os protótipos das funções de blue_com_cliente.c
externals.h	Contém os protótipos das funções de plotter.c
tipos.h	Contém a descrição das estruturas de dados utilizadas pelas aplicações e constantes utilizadas para identificar um pacote da aplicação.

Arquivos do tipo Makefile:

Nome do arquivo	Descrição
Makefile_segunda	Arquivo utilizado com o comando “make” para compilar e gerar o módulo do RTLinux/Free do sistema de 2ª ordem
Makefile_pendulo	Arquivo utilizado com o comando “make” para compilar e gerar o módulo do RTLinux/Free do sistema de um pêndulo
Makefile_servidor	Arquivo genérico utilizado com o comando “make” para compilar e gerar um executável de um servidor Bluetooth.
Makefile_cliente	Arquivo genérico utilizado com o comando “make” para compilar e gerar um executável de um cliente Bluetooth.
Makefile_gravador	Arquivo genérico utilizado com o comando “make” para compilar e gerar um executável de um programa que registra dados e imprime na tela utilizando o Gnuplot

A2 – Código Fonte

Serão apresentados os códigos fontes de pendulo.c, blue_com_servidor.c, blue_com_cliente.c, cliente_pendulo.c e plotter.c, gravador_servidor_pendulo.c, tipos.h, Makefile_pendulo e Makefile_servidor. Os demais arquivos podem ser requisitados, através do email ehwatanabe@aol.com, ou estarão em anexo com a cópia digital a ser entregue para o DEL.

pendulo_motor.c

```
/* Simulacao */
#include <math.h>
#include <time.h>
#include <signal.h>
#include <rtl.h>
#include <rtl_fifo.h>
#include <rtl_time.h>
#include <pthread.h>
#include <rtl_sched.h>
#include <rtl_debug.h>
#include "tipos.h"

pthread_t thread;
void * start_pendulo(void);

void * start_pendulo(void)
{
	pthread_self()->uses_fp = 0; /* to force save/restore */
	pthread_setfp_np (pthread_self(), 1);
	struct sched_param p;	/*struct necessario para alterar a prioridade do
thread*/
	p . sched_priority = 99;/*prioridade */
	pthread_setschedparam (pthread_self(), SCHED_FIFO, &p); /*ajustando
parametros*/

	/*pacote de dados*/
	packet_sim packet51, packet52;
	packet_init_sim packet_init;

	/*dados do simulador*/
	double t,u,h,pi,tau;
	pi = 3.14159265358979;
	t=0;		/*tempo de simulaÃ§ao*/
	u=0;		/*sinal de controle(tensao)*/
	tau=0;		/*torque*/
	double mp,LP,LT,rp,IT,g;
	double a1,a2,d;
	double Rm,Km,Kg;
	Rm = 2.6;
	Km = 0.00767;
	Kg = 14*5;
	mp = 0.14;
	LP = 0.43;
	LT = 0.21;
	rp = LP/2;
	IT = 0.0044;
	g  = 9.8;
	u= 0; /*sinal de controle (tensao)*/
	a1 = mp*(LT*LT) + IT;
	a2 = mp*(LT*LT);
	d  = mp*LT;
	double u2=0;
	double thetaT,thetaP,dthetaT,dthetaP;
	double x[4],h_tmp;
	double dx[4][4];
	double cp,sp,a,b,ia,dpt,dp2,G1;
	/*Condicoes Iniciais*/
	thetaT   =0;
	thetaP   =pi*0.03;
	dthetaT  =0;
	dthetaP  =0;

	/*Amostragem e temporizadores*/
	double amostragem = 99; /*Nao e o periodo de amostragem. e o iterador que
verifica o momento de amostrar*/
	double periodo_amostragem = 0.04;
	int atraso_fixo=1;
```

```
	/*variaveis de comandos e estado do simulador */
	short int estado = 0;

	/*dados para observaÃ§ao*/
	 int cont=0;
	int enviados=0;
	int recebidos=0;
	int it =0;

	/*dados para medir o rtt*/
	hrtime_t elapsed_time,partida,chegada; /*devolve o tempo em nanosegundos*/


	/*periodo do simulador*/
	double iterador=0;
	int arredonda;
	double offset = 8;
	h=0.0005;          /*passo de integracao*/
	pthread_make_periodic_np (pthread_self(), gethrtime(),500000); /*500 micro
s*/
	partida = clock_gethrtime(CLOCK_REALTIME);
	chegada = clock_gethrtime(CLOCK_REALTIME);
	while(1)
	{
		iterador++;
		partida = chegada;
		pthread_wait_np ();
		if(estado ==0)
		{
		 if((rtf_get(53,&packet_init,sizeof(packet_init))) > 0)
		 {
		     if(packet_init.ID == INICIAR)
		     {
				rtl_printf("Inicio de transmissao para o Servidor
Bluetooth\n");
				rtl_printf("Inicio de Simulacao\n");
				estado = 1;
				amostragem = iterador; /*enviando imediatamente a
informacao*/

	periodo_amostragem=(double)packet_init.payload[0];/*periodo de amostragem*/
				atraso_fixo = (int)packet_init.payload[1];/*tipo de
controle*/
				arredonda = (int) ((periodo_amostragem/h)+0.5);
				offset = (double) arredonda;
				u     = 0.0;
				u2=0;
				tau   = 0.0;
				estado = 1;
				thetaT   = (double)packet_init.payload[2];
				thetaP   = (double)packet_init.payload[3];
				dthetaT  = (double)packet_init.payload[4];
				dthetaP  = (double)packet_init.payload[5];
				 cont=0;
				enviados=0;
				recebidos=0;
			}
		       cont=0;
		  }
		 }
		if(amostragem == iterador)  /*tipicamente, a cadas 40 ms*/
		{
			/*rtl_printf("t = %d \n ",(int)(1000*t));*/
			/*rtf_put(40,&t,sizeof(t));
			rtf_put(39,&thetaT,sizeof(thetaT));
			rtf_put(38,&thetaP,sizeof(thetaP));*/
```

```
                    if(atraso_fixo == 1)
                    {
                          u=u2;
                    }
                    amostragem = iterador + offset; //Aqui se ajusta o periodo de
amostragem;
                    //rtl_printf("amostragem %d e  iterador %d, estado %d
",(int)amostragem,(int)iterador,(int)estado);
                    if(estado == 1)
                    {
                          enviados = enviados+1;
                          if(isnan(thetaP) > 0)
                          {
                                packet51.ID = NaNError;
                                u     = 0.0;
                                u2=0;
                                tau   = 0.0;
                                estado = 0;
                                rtl_printf("Fim de transmissao para o Servidor
Bluetooth1\n");
                                rtl_printf("Fim de Simulacao1\n");
                                thetaT   = (double)packet_init.payload[2];
                                thetaP   = (double)packet_init.payload[3];
                                dthetaT  = (double)packet_init.payload[4];
                                dthetaP  = (double)packet_init.payload[5];
                                cont=0;
                                enviados=0;
                                recebidos=0;
                                rtf_flush(38);
                                rtf_flush(39);
                                rtf_flush(40);
                                rtf_flush(51);
                                rtf_flush(52);
                                rtf_flush(53);
                          }
                          else
                          {
                                packet51.ID = DADOS;
                                packet51.timestamp=(long
long)clock_gethrtime(CLOCK_REALTIME);
                                packet51.payload[0]= (float)thetaT;
                                packet51.payload[1]= (float)thetaP;
                                packet51.payload[2]= (float)dthetaT;
                                packet51.payload[3]= (float)dthetaP;
                                /*rtl_printf("dados enviados %d \n",enviados);*/
                          }
                          rtf_put(51,&packet51,sizeof(packet51));
                    }
              }

              if(estado == 1)
              {
                    if((rtf_get(52,&packet52,sizeof(packet52))) > 0)
                    {
                          cont = 0;
                           /*rtl_printf("chegou!! \n");*/
                          if(packet52.ID == DADOS )
                          {
                                recebidos= recebidos+1;
                                chegada = clock_gethrtime(CLOCK_REALTIME);
                                elapsed_time = chegada - packet52.timestamp;
                                /*rtl_printf("rtt = %Ld \n",(long
long)elapsed_time);*/
                                rtf_put(40,&elapsed_time,sizeof(elapsed_time));

                                u2 = (double) packet52.payload[0];
                                if(atraso_fixo == 0)
                                {
                                      u =u2;
```

```
                        }
                    }
                    else
                    {
                        if(packet52.ID == REINICIAR)
                        {
                            u     = 0.0;
                            u2=0;
                            tau   = 0.0;
                            estado = 0;
                            rtl_printf("Reiniciando Simulacao.\n");
                            thetaT   = (double) packet_init.payload[2];
                            thetaP   = (double)packet_init.payload[3];
                            dthetaT  = (double)packet_init.payload[4];
                            dthetaP  = (double)packet_init.payload[5];
                            cont=0;
                            enviados=0;
                            recebidos=0;
                            rtf_flush(38);
                            rtf_flush(39);
                            rtf_flush(40);
                            rtf_flush(51);
                            rtf_flush(52);
                            rtf_flush(53);
                        }
                    }
                }
                else
                {
                    cont = cont +1;
                    if(cont > 40000 && estado == 1)/*timeout -> reset
simulation*/
                    {
                        u     = 0.0;
                        u2=0;
                        tau   = 0.0;
                        estado = 0;
                        rtl_printf("Fim de transmissao para o Servidor
Bluetooth3\n");
                        rtl_printf("Fim de Simulacao3\n");
                        thetaT   = (double) packet_init.payload[2];
                        thetaP   = (double) packet_init.payload[3];
                        dthetaT  = (double) packet_init.payload[4];
                        dthetaP  = (double) packet_init.payload[5];
                        cont=0;
                        enviados=0;
                        recebidos=0;
                        rtf_flush(38);
                        rtf_flush(39);
                        rtf_flush(40);
                        rtf_flush(51);
                        rtf_flush(52);
                        rtf_flush(53);
                    }
                }
            }
            if(estado == 1)
            {
                    /*Metodo de Runge Kutta para resolucao de EDOs */
                    /*y[k+1] = y[k] + (1/6)*(k1+ 2 k2 +2 k3 + k4) */

                    x[0]     = thetaT;
                    x[1]     = thetaP;
                    x[2]     = dthetaT;
                    x[3]     = dthetaP;
                    it=0;
                    for(;it<4;it++)
                    {
```

```
                                        if(it < 2)/*nao influencia no calculo de
k1*/
                                        {
                                              h_tmp=h/2; /*para o calculo de k2 e
k3*/
                                        }
                                        else
                                        {
                                              h_tmp=h;/*para o calculo de k4*/
                                        }

                                        cp       = cos(x[1]);/*cos(thetaP)*/
                                        sp       = sin(x[1]);/*sin(thetaP)*/
                                        a        = a1 - a2*cp*cp;
                                        b        = sp*cp;
                                        ia       = 1/a;
                                        dpt      = x[2]*x[3]; /*dthetaT * dthetaP*/
                                        dp2      = x[3]*x[3];/*dthetaP^2*/
                                        G1       = LT*dpt+g;
                                        tau      = u*(Km*Kg/Rm) -
((Km*Km)*(Kg*Kg)*x[2])/Rm;
                                        /*embora u seja considerado constante
durante na resolucao numerica, o torque tem
                                        que ser atualizado em cada iteracao pelos
novos valores de thetaT*/
                                        dx[it][0] = x[2];
                                        dx[it][1] = x[3];
                                        dx[it][2]  = (ia)   *(         tau - d *b
*G1  + d * rp * sp *dp2 );
                                        dx[it][3]  = (ia/rp)*(-(LT*cp)*tau +
a1*sp*G1  - d*b*LT * rp *dp2);

                                        if(it < 3)/*so e necessario para calcular
k2 e k3*/
                                        {
                                              x[0]  = thetaT  + h_tmp * dx[it][0];
                                              x[1]  = thetaP  + h_tmp * dx[it][1];
                                              x[2]  = dthetaT + h_tmp * dx[it][2];
                                              x[3]  = dthetaP + h_tmp * dx[it][3];
                                        }
                                  }
                                  /*atualizando os estados*/
      thetaT   = thetaT + (h/6) * (dx[0][0] + 2*dx[1][0] + 2*dx[2][0] + dx[3][0]);
      thetaP   = thetaP + (h/6) * (dx[0][1] + 2*dx[1][1] + 2*dx[2][1] + dx[3][1]);
      dthetaT  = dthetaT+ (h/6) * (dx[0][2] + 2*dx[1][2] + 2*dx[2][2] + dx[3][2]);
      dthetaP  = dthetaP+ (h/6) * (dx[0][3] + 2*dx[1][3] + 2*dx[2][3] + dx[3][3]);
      t = t+h;
            }
      }
      rtl_printf("Terminou!!\n");
      return 0;
}

int fifo_handler1(unsigned int fifo)/*handler*/
{
      packet_sim msg;
      int r;
      r = rtf_get(fifo, &msg, sizeof(msg));
      r = rtf_put(fifo, &msg, sizeof(msg));
      return 0;
}

int fifo_handler2(unsigned int fifo)/*handler para */
{
      packet_init_sim msg;
      int r;
      r = rtf_get(fifo, &msg, sizeof(msg));
      r = rtf_put(fifo, &msg, sizeof(msg));
      return 0;
```

```
}

int init_module(void) {

       rtf_destroy(38); /*escrebe thetaP amostrado*/ /*gravador*/
       rtf_destroy(39); /*escrebe thetaT amostrado*/ /*gravador*/
       rtf_destroy(40); /*escreve t amostrado*/ /*gravador*/
       rtf_destroy(51); /*escreve para servidor_bluetooth*/
       rtf_destroy(52); /*le para servidor bluetooth*/
       rtf_destroy(53); /*canal de inicializacao do simulador*/
       rtf_create(38, 1000);
       rtf_create(39, 1000);
       rtf_create(40, 4000);
       rtf_create(51, 4000);
       rtf_create(52, 4000);
       rtf_create_handler(52, fifo_handler1);
       rtf_create(53, 1000);
       rtf_create_handler(53, fifo_handler2);
       return pthread_create(&thread, NULL,(void *)start_pendulo,(void *) NULL);
}

void cleanup_module(void) {
       printk("%d\n", rtf_destroy(38));
       printk("%d\n", rtf_destroy(39));
       printk("%d\n", rtf_destroy(40));
       printk("%d\n", rtf_destroy(51));
       printk("%d\n", rtf_destroy(52));
       pthread_delete_np (thread);
}
```

blue_com_servidor.c

```
/*blue_com*/

#include "blue_com.h"
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <syslog.h>
#include <string.h>
#include <errno.h>
#include <signal.h>
#include <sys/time.h>
#include <time.h>
#include <sys/poll.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <arpa/inet.h>
#include <resolv.h>
#include <netdb.h>
#include <asm/types.h>
#include <asm/byteorder.h>
#include <bluetooth/bluetooth.h>
#include <bluetooth/l2cap.h>

void float2buffer(float input,char* buffer,int pos_mem){
  //a verificao sobre a alocacao de memoria e feita antes da chamada dessa funcao
  //pos_mem deve ser um valor inteiro nao negativo, movendo de 4 em 4 bytes para
  //copiar o conteudo de um float para o buffer[pos_mem*4]
  memcpy(&buffer[pos_mem*4],&input, sizeof(input));
}
float buffer2float(char*buffer,int pos_mem){
    float output;
    memcpy(&output,&buffer[pos_mem*4], sizeof(output));
    return output;
}
```

```
float tv2fl(struct timeval tv)
{
       return (float)(tv.tv_sec*1000.0) + (float) (tv.tv_usec/1000.0);
}
void packet2buffer(packet_blue packet,char * buffer)
{
    memcpy(&buffer[0],&packet, sizeof(packet));
}
packet_blue buffer2packet(char*buffer)
{
    packet_blue output;
    memcpy(&output,&buffer[0], sizeof(output));
    return output;
}

void servidor_pendulo_motor(bdaddr_t origem)
{
  struct l2cap_options opts;
  int s,opt,imtu,omtu,psm;
  imtu = 130;
  omtu = 130;
  psm  = 1033;
  struct sockaddr_l2 host;
  struct sockaddr_l2 target;
  unsigned char buf1[50],buf2[50];
  int comando;
  comando=1;/*no futuro definir uma constante como START_SIM para identificar o
comando*/
  char str_or[18];
  int s1;
  int num_bytes;
  int conexao = 0;
  packet_blue pacote_blue_s,pacote_blue_r;
  packet_sim pacote_sim_r,pacote_sim_w;
  packet_init_sim pacote_init;
  int num;
  /*Abrindo arquivos descritores para leitura*/
  int  rtcom51,rtcom52,rtinit; /* rtcom51 -> ler  ,rtcom52 -> escrever, rtinit ->
ajuste inicial*/
  int n =0;
  int cont=0;
  int teste_sel =1;
  fd_set rfds;
  ba2str(&origem,str_or);
  if((s= socket(PF_BLUETOOTH,SOCK_SEQPACKET,BTPROTO_L2CAP)) < 0)
  {
    perror("Can't create a socket.");
    exit(1);
  }
  memset(&host,0,sizeof(host));
  host.l2_family = AF_BLUETOOTH;
  baswap(&host.l2_bdaddr,strtoba(str_or));
  host.l2_psm = htobs(psm);
  if( (bind(s, (struct sockaddr *) &host,sizeof(host)))  < 0)
  {
    perror("Can't bind socket.");
    exit(1);
  }
  opt = sizeof(opts);
  if (getsockopt(s, SOL_L2CAP, L2CAP_OPTIONS, &opts, &opt) < 0)
  {
       perror("Can't get default L2CAP options.");
        exit(1);
  }
  /* Set new options */
  opts.omtu = omtu;
  opts.imtu = imtu;
  if (setsockopt(s, SOL_L2CAP, L2CAP_OPTIONS, &opts, opt) < 0)
  {
```

```
       perror("Can't set default L2CAP options.");
        exit(1);  }

  opt = sizeof(opts);
  if (getsockopt(s, SOL_L2CAP, L2CAP_OPTIONS, &opts, &opt) < 0)
  {
       perror("Can't get default L2CAP options.");
        exit(1);
  }
  /*syslog(LOG_INFO, "Connected [imtu %d, omtu %d, flush_to %d]\n", opts.imtu,
opts.omtu, opts.flush_to);*/
fprintf( stderr,"[imtu %d, omtu %d, flush_to %d]\n", opts.imtu, opts.omtu,
opts.flush_to);
  if( (listen(s,10)) < 0)
  {
    perror("Can't listen.");
    exit(1);
  }
  int len = sizeof(host);
  int status;
  float Tamostragem;
  float cond_iniciais[4];
  float auxiliar[2];
  float controle;
  int maximo1 =0;
  float ganho[4];
  float sinal_controle;
  while(1)
  {

       if( ( s1 = accept(s, (struct sockaddr *) &target,&len) ) < 0)
       {
       perror("Can't accept.");
       exit(1);
       }
       else
              conexao =1;

       if(fork())
       {
              close(s1);
              wait(&status);/*faz o processo pai dormir, assim so uma conexao Ã©
atendida por              vez*/
              continue;
       }
       close(s);

       if( (recv(s1,buf1,sizeof(buf1),0)) < 0)
       {
                     perror("Recv Failed");
              exit(1);
       }
       pacote_blue_r = buffer2packet(buf1);
       if(pacote_blue_r.ID2 != SETUP_INIT )
       {
              fprintf( stderr,"Nao completou o ajuste da amostragem\n");
              exit(0);
       }
       else
       {
              Tamostragem = pacote_blue_r.carga[0];
              controle    = pacote_blue_r.carga[1];
              cond_iniciais[0] = pacote_blue_r.carga[2];
              cond_iniciais[1] = pacote_blue_r.carga[3];

       memcpy(auxiliar,&pacote_blue_r.timestamp2,sizeof(pacote_blue_r.timestamp2));
              cond_iniciais[2] = auxiliar[0];
              cond_iniciais[3] = auxiliar[1];
       }
```

```
pacote_blue_s.ID2 = SETUP_INIT_ACK;
packet2buffer(pacote_blue_s,buf2);
if((num = send(s1,buf2,sizeof(buf2),0)) <= 0)
{
	perror("Send failed!!!");
	exit(1);
}

fprintf(stderr, "Connected ...\n");
/*Cuidando da comunicacao com o simulador*/

if( (recv(s1,buf1,sizeof(buf1),0)) < 0)
{
		perror("Recv Failed");
	exit(1);
}
pacote_blue_r = buffer2packet(buf1);

if ((rtcom51 = open("/dev/rtf51", O_RDONLY)) < 0)
{
	fprintf(stderr, "Error opening /dev/rtf51\n");
	exit(1);
}
if ((rtcom52 = open("/dev/rtf52", O_WRONLY)) < 0)
{

	fprintf(stderr, "Error opening /dev/rtf52\n");
	exit(1);
}
if ((rtinit = open("/dev/rtf53", O_WRONLY)) < 0)
{
	fprintf(stderr, "Error opening /dev/rtf53\n");
	exit(1);
}
 if(pacote_blue_r.ID2 == SETUP_DCS )
 {
	pacote_blue_s.ID2 = SETUP_DCS_ACK;
	packet2buffer(pacote_blue_s,buf1);
	if((num = send(s1,buf1,sizeof(buf1),0)) <= 0)
	{
		perror("Send failed!!!");
		exit(1);
	}

	pacote_init.ID = INICIAR;
	pacote_init.payload[0] = Tamostragem;
	pacote_init.payload[1] = controle;
	pacote_init.payload[2] = cond_iniciais[0];
	pacote_init.payload[3] = cond_iniciais[1];
	pacote_init.payload[4] = cond_iniciais[2];
	pacote_init.payload[5] = cond_iniciais[3];
	if( (write(rtinit,&pacote_init,sizeof(pacote_init))) < 0)
	{
		fprintf(stderr,"Nao escreveu no arquivo rtf53");
	}
	maximo1 = maximo(s1,rtcom52,rtcom51,rtinit);
	while(conexao)
	{
		fprintf(stderr,"entrei no loop \n");
		FD_ZERO(&rfds);
		FD_SET(rtcom51,&rfds);
		FD_SET(s1,&rfds);

		if( (teste_sel =select( maximo1+1,&rfds,NULL,NULL,NULL)) < 0)
		{
				printf("Select Error \n");
		}
		if(FD_ISSET(rtcom51,&rfds)) //chegou dados em rtf51
```

```
                  {
                    if( (n = read(rtcom51,&pacote_sim_r,sizeof(pacote_sim_r))) <
0)
                          {
                                fprintf(stderr,"Nao leu o arquivo rtf51");
                          }
                          fprintf(stderr,"recebi \n");
                          /*gettimeofday(&start,NULL);*/
                          if(pacote_sim_r.ID == DADOS)
                          {
                                pacote_blue_s.ID2       = DADOS;
                                pacote_blue_s.timestamp2 = pacote_sim_r.timestamp;
                                pacote_blue_s.carga[0] =  pacote_sim_r.payload[0];
                                pacote_blue_s.carga[1] =  pacote_sim_r.payload[1];
                                pacote_blue_s.carga[2] =  pacote_sim_r.payload[2];
                                pacote_blue_s.carga[3] =  pacote_sim_r.payload[3];
                                packet2buffer(pacote_blue_s,buf1);
                                if( (num_bytes = send(s1,buf1,sizeof(buf1),0)) <
0)
                                {
                                      perror("Send Error.");
                                      exit(1);
                                }
                                else
                                {
                                      cont =cont+1;
                                      fprintf(stderr,"transmissoes
feitas=%d",cont);
                                }
            fprintf(stderr,"thetaT =%f, thetaP =%f, dthetaT = %f, dthetaP =%f\n",
      pacote_blue_s.carga[0],pacote_blue_s.carga[1],pacote_blue_s.carga[2],pacote_b
      lue_s.carga[3]);
                          }
                          if(pacote_sim_r.ID == NaNError)
                          {
                                      perror("NaN Error");
                                      exit(0);
                          }
                  } /*fim do ISSET(rtcom52)*/
            if(FD_ISSET(s1,&rfds)) //chegou dados via socket bluetooth
            {
                  if( (num_bytes = recv(s1,buf2,sizeof(buf2),0)) < 0)
                  {
                        perror("Read Error.");
                        exit(1);
                  }
                  else
                  {
                        pacote_blue_r = buffer2packet(buf2);
                        pacote_sim_w.ID = DADOS;
                        pacote_sim_w.payload[0] = pacote_blue_r.carga[0];
                        pacote_sim_w.payload[1] = pacote_blue_r.carga[1];
                        pacote_sim_w.payload[2] = pacote_blue_r.carga[2];
                        pacote_sim_w.payload[3] = pacote_blue_r.carga[3];
                        pacote_sim_w.timestamp  = pacote_blue_r.timestamp2;
                        if( (write(rtcom52,&pacote_sim_w,sizeof(pacote_sim_w)))<
0)
                        {
                              fprintf(stderr,"Nao escreveu no arquivo rtf52");
                        }

                   }
               }
            }

      }
      if(pacote_blue_r.ID2 == SETUP_MONITOR )
      {
            pacote_blue_s.ID2 = SETUP_MONITOR_ACK;
```

```
		ganho[0] = pacote_blue_r.carga[0];
		ganho[1] = pacote_blue_r.carga[1];
		ganho[2] = pacote_blue_r.carga[2];
		ganho[3] = pacote_blue_r.carga[3];
		packet2buffer(pacote_blue_s,buf1);
		if((num = send(s1,buf1,sizeof(buf1),0)) <= 0)
		{
			perror("Send failed!!!");
			exit(1);
		}
		pacote_init.ID = INICIAR;
		pacote_init.payload[0] = Tamostragem;
		pacote_init.payload[1] = controle;
		pacote_init.payload[2] = cond_iniciais[0];
		pacote_init.payload[3] = cond_iniciais[1];
		pacote_init.payload[4] = cond_iniciais[2];
		pacote_init.payload[5] = cond_iniciais[3];

		if( (write(rtinit,&pacote_init,sizeof(pacote_init))) < 0)		{
			fprintf(stderr,"Nao escreveu no arquivo rtf");
		}
		maximo1 = maximo(s1,rtcom52,rtcom51,rtinit);
		while(conexao)
		{
			fprintf(stderr,"entrei no loop \n");
			FD_ZERO(&rfds);
			FD_SET(rtcom51,&rfds);

			if( (teste_sel =select( maximo1+1,&rfds,NULL,NULL,NULL)) < 0)
			{
				printf("Select Error \n");
			}
			if(FD_ISSET(rtcom51,&rfds)) //chegou dados em rtf51
			{
				if((n=read(rtcom51,&pacote_sim_r,sizeof(pacote_sim_r)))<
0)
				{
					fprintf(stderr,"Nao leu o arquivo rtf51");
				}
sinal_controle = -(pacote_sim_r.payload[0]*ganho[0]+
pacote_sim_r.payload[1]*ganho[1] + (pacote_sim_r.payload[2]*ganho[2]) +
pacote_sim_r.payload[3]*ganho[3]);
				pacote_sim_w.ID = DADOS;
				pacote_sim_w.payload[0] = sinal_controle;
				pacote_sim_w.timestamp  = pacote_sim_r.timestamp;
				if( (write(rtcom52,&pacote_sim_w,sizeof(pacote_sim_w)))<
0)
				 {
					fprintf(stderr,"Nao escreveu no arquivo rtf52");
				}
				if(pacote_sim_r.ID == DADOS)
				{
					pacote_blue_s.ID2        = DADOS;
					pacote_blue_s.timestamp2 = pacote_sim_r.timestamp;
					pacote_blue_s.carga[0] =  pacote_sim_r.payload[0];
					pacote_blue_s.carga[1] =  pacote_sim_r.payload[1];
					pacote_blue_s.carga[2] =  pacote_sim_r.payload[2];
					pacote_blue_s.carga[3] =  pacote_sim_r.payload[3];
					packet2buffer(pacote_blue_s,buf1);

					if( (num_bytes = send(s1,buf1,sizeof(buf1),0)) <
0)
					{
						perror("Send Error.");
						exit(1);
					}
					else
					{
						cont =cont+1;
```

```
                                        fprintf(stderr,"transmissoes
feitas=%d",cont);
                                }
                                fprintf(stderr,"thetaT =%f, thetaP =%f, dthetaT =
                        %f, dthetaP =%f\n",
                        pacote_blue_s.carga[0],pacote_blue_s.carga[1],pacote_blue
                        _s.carga[2],pacote_blue_s.carga[3]);
                        }
                        if(pacote_sim_r.ID == NaNError)
                        {
                                perror("NaN Error");
                                exit(0);
                        }
                } /*fim do ISSET(rtcom52)*/
          }
      }
      exit(0);
     }
  close(s1);
  close(s);
 }

 int maximo(int s1,int s2,int s3,int s4)
 {
       int aux1,aux2;
       if( s1 > s2 )
         aux1 = s1;
       else
         aux1 = s2;

       if( s3 > s4 )
          aux2 = s3;
       else
          aux2 = s4;

       if(aux1 > aux2)
              return aux1;
       else
              return aux2;
 }
```

BLUE_COM_CLIENTE.C

```
#include "blue_com_cliente.h"
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <syslog.h>
#include <string.h>
#include <errno.h>
#include <signal.h>
#include <sys/time.h>
#include <time.h>
#include <sys/poll.h>
#include <sys/socket.h>

#include <netinet/in.h>
#include <netinet/tcp.h>
#include <arpa/inet.h>
#include <resolv.h>
#include <netdb.h>

#include <asm/types.h>
#include <asm/byteorder.h>
#include <sys/types.h>
#include <sys/stat.h>
```

```
#include <bluetooth/bluetooth.h>
#include <bluetooth/l2cap.h>

 int do_connect(char *origem,char *servidor,int PSM) //retirado do arquivo l2test.c
 {
       struct sockaddr_l2 remote_addr, local_addr;
       struct l2cap_options opts;
       int s,opt,imtu,omtu,psm;
       imtu = 130;
       omtu = 130;
       psm  = PSM;
       fprintf(stderr,"\n Criando Socket ... \n");
       if ((s = socket(PF_BLUETOOTH, SOCK_SEQPACKET, BTPROTO_L2CAP)) < 0) {
              syslog(LOG_ERR, "Can't create socket. %s(%d)", strerror(errno),
errno);
              return -1;
       }
       memset(&local_addr, 0, sizeof(local_addr));
       local_addr.l2_family = AF_BLUETOOTH;
       baswap(&local_addr.l2_bdaddr,strtoba(origem));
       if (bind(s, (struct sockaddr *) &local_addr, sizeof(local_addr)) < 0) {
              syslog(LOG_ERR, "Can't bind socket. %s(%d)", strerror(errno), errno);
              exit(1);
       }

       /* Get default options */
       opt = sizeof(opts);
       if (getsockopt(s, SOL_L2CAP, L2CAP_OPTIONS, &opts, &opt) < 0) {
              syslog(LOG_ERR, "Can't get default L2CAP options. %s(%d)",
strerror(errno), errno);
              return -1;
       }

       /* Set new options */
       opts.omtu = omtu;
       opts.imtu = imtu;
       if (setsockopt(s, SOL_L2CAP, L2CAP_OPTIONS, &opts, opt) < 0) {
              syslog(LOG_ERR, "Can't set L2CAP options. %s(%d)", strerror(errno),
errno);
              return -1;
       }
       memset(&remote_addr, 0, sizeof(remote_addr));
       remote_addr.l2_family = AF_BLUETOOTH;
       baswap(&remote_addr.l2_bdaddr, strtoba(servidor));
       remote_addr.l2_psm = htobs(psm);

/*fprintf( stderr,"Connected [imtu %d, omtu %d, flush_to %d ]\n",opts.imtu,
opts.omtu, opts.flush_to);*/

       if (connect(s, (struct sockaddr *)&remote_addr, sizeof(remote_addr)) < 0 ) {
              syslog(LOG_ERR, "Can't connect. %s(%d)", strerror(errno), errno);
              close(s);
              return -1;
       }

       opt = sizeof(opts);
       if (getsockopt(s, SOL_L2CAP, L2CAP_OPTIONS, &opts, &opt) < 0) {
              syslog(LOG_ERR, "Can't get L2CAP options. %s(%d)", strerror(errno),
errno);
              close(s);
              return -1;
       }

       /*syslog(LOG_INFO, "Connected [imtu %d, omtu %d, flush_to %d]\n",
              opts.imtu, opts.omtu, opts.flush_to);*/
       fprintf( stderr,"Connected [imtu %d, omtu %d, flush_to %d]\n",opts.imtu,
opts.omtu, opts.flush_to);
       return s;
}
```

```
void cliente_pendulo_motor(char * origem, char * destino,int controle,float
ganho[5],float Tamostragem, int setup,float cond_init[4],float tempo)
{

   /*FILE *shell;
   static char * delete_fifos = "rm .fifo11 .fifo12 .fifo13 .fifo14 .fifo15";
   shell = popen(delete_fifos, "w");*/

  if(Tamostragem < 0.0005)
  {
       fprintf( stderr,"\namostragem deve ser maior ou igual que 10 ms.\n");
       exit(0);
  }
  if((controle > 1) || (controle < 0  ) )
  {
       fprintf( stderr,"\n variavel controle = {0,1}.\n");
       exit(0);
  }

  criar_fifos_pendulo();


  int  fd1,fd2,fd3,fd4,fd5; /*thetaT,thetaP,dthetaT,dthetaP,controle,erro*/

  /*SE UTILIZAR O_WRONLY ou O_RDONLY, o open() vai bloquear ate que o outro
programa que vai ler ou
  escrever acessar a FIFo CUIDADO!!!!!  Usar | O_NONBLOCK para nao bloquear quando
a fila ficar cheia*/
  if ((fd1 = open(".fifo11", O_RDWR| O_NONBLOCK )) < 0) /*corrente*/
  {
             fprintf(stderr, "Error opening .fifo11 \n");
             exit(1);
  }
  if ((fd2 = open(".fifo12", O_RDWR| O_NONBLOCK )) < 0) /*w*/
  {
             fprintf(stderr, "Error opening .fifo12 \n");
             exit(1);
  }
  if ((fd3 = open(".fifo13", O_RDWR| O_NONBLOCK )) < 0) /*controle*/
  {
             fprintf(stderr, "Error opening .fifo13 \n");
             exit(1);
  }
  if ((fd4 = open(".fifo14", O_RDWR| O_NONBLOCK )) < 0) /*erro*/
  {
             fprintf(stderr, "Error opening .fifo14 \n");
             exit(1);
  }
  if ((fd5 = open(".fifo15", O_RDWR | O_NONBLOCK)) < 0) /*erro*/
  {
             fprintf(stderr, "Error opening .fifo15 \n");
             exit(1);
  }

  unsigned char buf1[50],buf2[50];
  int s;
  fprintf(stderr,"\n Vou conectar  \n");
  int PSM = 1033;
  if((s= do_connect(origem,destino,PSM)) < 0)
  {
    perror("Can't create a socket.");
    exit(1);
  }
  int num=0;
  signal(SIGINT,quit);


 float u;
```

```
 u=0;

 packet_blue pct1,pct2;
 float auxiliar[2];
 float t=0;
 /*setup*/
  pct2.ID2 = SETUP_INIT;
  pct2.carga[0] = Tamostragem;
  pct2.carga[1] = (float) controle;
  pct2.carga[2] = cond_init[0]; /*condicao inicial thetaT*/
  pct2.carga[3] = cond_init[1];/*condicao inicial thetaP*/

  auxiliar[0]   = cond_init[2];/*condicao inicial dthetaT*/
  auxiliar[1]   = cond_init[3];/*condicao inicial dthetaP*/
  memcpy(&pct2.timestamp2,auxiliar,sizeof(auxiliar)); /*copia dois floats (32 bits)
para o long long (64 bits)*/
  packet2buffer(pct2,buf1);

  if((num = send(s,buf1,sizeof(buf1),0)) <= 0) /*estou desperdicando espaco. No
futuro criar pacote de tamanho variavel*/
  {
       perror("Send failed!!!");
       exit(1);
  }
  if( (num = recv(s,buf2,sizeof(buf2),0)) < 0)
  {
              perror("Recv Failed");
       exit(1);
  }
  pct1 = buffer2packet(buf2);
  if(pct1.ID2 != SETUP_INIT_ACK )
  {
              perror("Setup Amostragem Failed");
       exit(1);
  }

 if(setup == SETUP_DCS)
 {
       pct2.ID2      = SETUP_DCS;
       pct2.carga[0] = ganho[0];
       pct2.carga[1] = ganho[1];
       packet2buffer(pct2,buf1);
       if ( (num = send(s,buf1,sizeof(buf1),0)) <= 0)
       {
              perror("Send failed!!!");
              exit(1);
       }
       if( (num = recv(s,buf2,sizeof(buf2),0)) < 0)
       {
                    perror("Recv Failed");
                    exit(1);
       }
       pct1 = buffer2packet(buf2);
       if(pct1.ID2 != SETUP_DCS_ACK )
       {
                    perror("Setup DCS Failed");
                    exit(1);
       }
       else
       {
          /*DCS*/
          while(t < tempo)
          {
              t = t +Tamostragem;
              if( (num = recv(s,buf2,sizeof(buf2),0)) < 0)
              {
                           perror("Recv Failed");
                           exit(1);
              }
```

```
            pct1 = buffer2packet(buf2);
            if(controle == 1)
            {
                  u = -(ganho[0]*pct1.carga[0] + ganho[1]*pct1.carga[1] +
ganho[2]*pct1.carga[2] +ganho[3]*pct1.carga[3] + ganho[4]*u);
            }
            else
            {
                  u = -(ganho[0]*pct1.carga[0] + ganho[1]*pct1.carga[1] +
ganho[2]*pct1.carga[2] +ganho[3]*pct1.carga[3]);
            }

            pct2.ID2 = DADOS;
            pct2.carga[0]=u;
            pct2.timestamp2 = pct1.timestamp2;
            packet2buffer(pct2,buf1);
            fprintf(stderr,"\n Vou enviar  \n");

            if ( (num = send(s,buf1,sizeof(buf1),0)) <= 0)
            {
                  perror("Send failed!!!");
                  exit(1);
            }
            fprintf(stderr,"\n Enviei= %d  \n",num);

            log_dados_pendulo(pct1.carga,u,fd1,fd2,fd3,fd4,fd5);
            fprintf(stderr,"thetaT = %f,thetaP= %f,dthetaT= %f,dthetaP= %f,u=
%f\n",pct1.carga[0],pct1.carga[1],pct1.carga[2],pct1.carga[3],u);
        }
      }
  }
  if(setup == SETUP_MONITOR)
  {
  /*os ganhos sao enviados. O servidor pode usa-los para controlar a planta e
enviar os dados.
    Para parar de receber dados, usar o sinal gerado por ctrl-c.
  */
      pct2.ID2 = SETUP_MONITOR;
      pct2.carga[0] = ganho[0];
      pct2.carga[1] = ganho[1];
      pct2.carga[2] = ganho[2];
      pct2.carga[3] = ganho[3];
      packet2buffer(pct2,buf1);
      if ( (num = send(s,buf1,sizeof(buf1),0)) <= 0)
      {
            perror("Send failed!!!");
            exit(1);
      }
      if( (num = recv(s,buf2,sizeof(buf2),0)) < 0)
      {
                  perror("Recv Failed");
                  exit(1);
      }
      pct1 = buffer2packet(buf2);
      if(pct1.ID2 != SETUP_MONITOR_ACK )
      {
                  perror("Setup Monitor Failed");
                  exit(1);
      }
      else
      {     int cont=0;
            while(t< tempo)
            {
                  cont = cont +1;
                  if( (num = recv(s,buf2,sizeof(buf2),0)) < 0)
                  {
                              perror("Recv Failed");
                              exit(1);
                  }
```

```
                    pct1 = buffer2packet(buf2);
                      log_dados_pendulo(pct1.carga,u,fd1,fd2,fd3,fd4,fd5);
                    fprintf(stderr,"thetaT = %f,thetaP= %f,dthetaT= %f,dthetaP=
%f,u= %f\n",pct1.carga[0],pct1.carga[1],pct1.carga[2],pct1.carga[3],u);

              }
        }
  }

  close(s);
}/*fim do cliente pendulo motor*/

void criar_fifos_pendulo()
{
  if(mknod(".fifo11",S_IFIFO | 0644 , 0) !=0)  /*criando fifos*/
  {

       perror("fila11 nao criada!!");
  }
  if(mknod(".fifo12",S_IFIFO | 0644 , 0)!=0)
  {

       perror("fila12 nao criada!!");
  }
  if(mknod(".fifo13",S_IFIFO | 0644 , 0)!=0)
  {

       perror("fila13 nao criada!!");
  }
  if( mknod(".fifo14",S_IFIFO | 0644 , 0)!=0)
  {

       perror("fila14 nao criada!!");
  }
  if( mknod(".fifo15",S_IFIFO | 0644 , 0)!=0)
  {

       perror("fila15 nao criada!!");
  }
}

 void log_dados_pendulo(float dados[4],float u,int fd1, int fd2,int fd3,int fd4,int
fd5)
 {
              if( (write(fd1,&dados[0],sizeof(dados[0]))) < 0)
              {
                           fprintf(stderr,"Nao escreveu no arquivo .fifo11");
              }
              if( (write(fd2,&dados[1],sizeof(dados[1]))) < 0)
              {
                     fprintf(stderr,"Nao escreveu no arquivo .fifo12");
              }
              if( (write(fd3,&dados[2],sizeof(dados[2]))) < 0)
              {
                           fprintf(stderr,"Nao escreveu no arquivo .fifo13");
              }
              if( (write(fd4,&dados[3],sizeof(dados[3]))) < 0)
              {
                           fprintf(stderr,"Nao escreveu no arquivo .fifo14");
              }
              if( (write(fd5,&u,sizeof(u))) < 0)
              {
                           fprintf(stderr,"Nao escreveu no arquivo .fifo15");
              }
 }

void float2buffer(float input,char* buffer,int pos_mem){
  //a verificao sobre a alocacao de memoria e feita antes da chamada dessa funcao
  //pos_mem deve ser um valor inteiro nao negativo, movendo de 4 em 4 bytes para
```

```
  //copiar o conteudo de um float para o buffer[pos_mem*4]
  memcpy(&buffer[pos_mem*4],&input, sizeof(input));
}

float buffer2float(char*buffer,int pos_mem){
    float output;
    memcpy(&output,&buffer[pos_mem*4], sizeof(output));
    return output;
}

float tv2fl(struct timeval tv)
{
      return (float)(tv.tv_sec*1000.0) + (float) (tv.tv_usec/1000.0);
}

void packet2buffer(packet_blue packet,char * buffer)
{
    memcpy(&buffer[0],&packet, sizeof(packet));

}

packet_blue buffer2packet(char*buffer)
{
    packet_blue output;
    memcpy(&output,&buffer[0], sizeof(output));
    return output;
}


 void quit()
 {
      exit(0);
 }
```

PLOTTER.C

```
/*CÓDIGO OBTIDO EM http://www.cs.cf.ac.uk/Dave/C/subsection2_22_25_3.html*/

#include <signal.h>
#include "externals.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>

 float tv2fl(struct timeval tv)
 {
      return (float)(tv.tv_sec*1000.0) + (float) (tv.tv_usec/1000.0);
 }


 void PlotOne(FILE *plot1,FILE *plot2,char * startplot1,char * startplot2)
 {
   fprintf(plot1, "%s", startplot1);
   fflush(plot1);
   fprintf(plot2, "%s", startplot2);
   fflush(plot2);
 }

 void RePlot(FILE *plot1,char * replot)
 {
   fprintf(plot1, "%s", replot);
   fflush(plot1);
 }

void plot(float y1, float y2,float y3,float y4,float i,FILE *fp1,FILE *fp2,FILE
*fp3,FILE *fp4)
```

```
{

    fprintf(fp1,"%f %f\n",i,y1);
    fprintf(fp2,"%f %f\n",i,y2);
    fprintf(fp3,"%f %f\n",i,y3);
    fprintf(fp4,"%f %f\n",i,y4);

    fflush(fp1);
    fflush(fp2);
    fflush(fp3);
    fflush(fp4);
}
void plot_rtt(float y1,float i,FILE *fp1)/*,FILE *plot1,FILE *plot2,char *
startplot1,char *startplot2)*/
{

    /* escreve no arquivo o indice e o retardo*/
    fprintf(fp1,"%f %f\n",i,y1);

    fflush(fp1);/*joga no grafico*/
}

void Imprime(FILE *plot1,char * startplot1)
{
   fprintf(plot1, "%s", startplot1);
   fflush(plot1);
}

void StopPlot(FILE *plot1,FILE *plot2)
{
  pclose(plot1);
  pclose(plot2);
}

void RemoveDat(char * deletefiles,FILE * ashell)
{
   ashell = popen(deletefiles, "w");
   exit(0);
}
```

gravador_servidor_pendulo.c

```
#ifdef HAVE_CONFIG_H
#include <config.h>
#endif
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <fcntl.h>
#include <unistd.h>
#include "externals.h"

int main()
{
  FILE *plot1;
  FILE *fp1;
  /*registrar o atraso RTT */
  if ( ((fp1 = fopen("rtt.dat","w")) == NULL)  )
  {
    printf("Error can't open rtt.dat\n");
    exit(1);
  }
  static char * startplot1  = "plot 'rtt.dat' with steps\n";
  static char * command1    = "/usr/bin/gnuplot> dump1";
  static char * deletefiles = "rm rtt.dat";
```

```
  static char * set_term    = "set terminal x11\n";

  plot1 = popen(command1, "w");/*abre a linha de comandos e chama o gnuplot*/
  fprintf(plot1, "%s", set_term);
  fflush(plot1);
  if (plot1 == NULL)
     exit(2);
  /*Abrindo arquivos descritores para leitura*/
  int  RTT;
  if ((RTT = open("/dev/rtf40", O_RDONLY)) < 0) {
       fprintf(stderr, "Error opening /dev/rtf40\n");
              exit(1);
  }
   long long rtt;
   int j   = 0;
   int pode_plotar = 0;
   for(;j < 20000;j++)/*recolhe 20000 amostras*/
   {
         if( (read(RTT,&rtt,sizeof(rtt))) < 0)
         {
                     fprintf(stderr,"Nao leu o arquivo rtf40");
         }
         plot_rtt(rtt,j,fp1);
         if(pode_plotar == j)
         {
             Imprime(plot1,startplot1);
             pode_plotar = pode_plotar + 25;
         }

   }
   int c=0;
   c=getchar();
   pclose(plot1);
  return EXIT_SUCCESS;
}
```

tipos.h

```
/*tipos.h*/

typedef struct{
       short int ID;
       long long int timestamp;
       float payload[4];
}packet_sim;

typedef struct{
       short int ID;
       float payload[8];
}packet_init_sim;


typedef struct{
       char ID2;
       long long int timestamp2;
       float carga[4];
}packet_blue;

#define PARAR 255
#define INICIAR 1
#define DADOS 2
#define SETUP_INIT 3
#define SETUP_INIT_ACK 4
#define SETUP_DCS 5
#define SETUP_DCS_ACK 6
#define SETUP_MONITOR 7
#define SETUP_MONITOR_ACK 8
#define REINICIAR 40
#define NaNError 51
```

Makefile_pendulo

```
#Cria módulo .o do RTLinux/Free
all: pendulo_motor.o

include /usr/src/rtlinux/rtl.mk

ifeq ($(ARCH),ppc)
CFLAGS += -mhard-float
endif

pendulo_motor.o: pendulo_motor.c
	$(CC) ${INCLUDE} ${CFLAGS} -g -c pendulo_motor.c -o pendulo_tmp.o
	ld -r -static pendulo_tmp.o -o pendulo_motor.o -L/usr/lib -lm
	rm -f pendulo_tmp.o


clean:
	rm -f *.o

include /usr/src/rtlinux/Rules.make
```

Makefile_servidor_bluetooth

```
#Makefile do projeto servidor_pendulo_motor
CC=gcc
LD=gcc
CFLAGS= -Wall -c -O2
LFLAGS= -Wall -lbluetooth -lsdp -f -o
OBJS= blue_com.o servidor_pendulo_motor.o
EXECS=servidor_pendulo_motor
LIBLINK = -L LIBDIR
INCLUDE = -I /usr/include/bluetooth/
.c.o:
	$(CC) $(CFLAGS) $(LIBLINK) $(INCLUDE)  $<
all:  servidor_pendulo_motor
servidor_pendulo_motor: $(OBJS)
	$(LD) $(LFLAGS) $@ $(OBJS) $(LIBLINK) $(INCLUDE)
clean:
	rm -f $(OBJS) $(EXECS)
```

Apêndice B

B1 – Instalando o Bluez no Linux 2.4.20 (distribuição Red Hat 9.0)

O Red Hat 9.0, uma distribuição Linux gratuita (a última da Red Hat antes do Fedora), tem um instalador bastante amigável que permite não apenas instalar o suporte ao Bluez como módulos do Kernel, como também instala aplicativos e ferramentas que auxiliam ao usuário na administração dos dispositivos *Bluetooth*. Dessas ferramentas em nível do usuário, 2 conjuntos são indispensáveis para um usuário utilizar o *Bluetooth*: o bluez-libs e o bluez-utils. Estes dois pacotes de software devem ser instalados.

Entretanto, o Bluez é um *software* livre em contínuo melhoramento e atualização, o que significa que melhorias e consertos devem ser procurados na página da Internet do Bluez, em [28]. Na página do Bluez, na seção *Downloads*, é necessário obter 3 coisas para melhorar o Bluez que já veio com uma distribuição do Linux:

- *Patch* (conserto) mais recente para o Bluez, na versão de Kernel utilizada;
- A versão mais recente do bluez-libs;
- A versão mais recente do bluez-utils;

A seguir, será indicada a seqüência para instalação adequada do Bluez, no Red Hat 9.0.

Assume-se que a distribuição Red Hat 9.0 já tenha sido instalada com sucesso, com suporte a USB e sem qualquer suporte ao *Bluetooth*. O que será feito é aplicar o patch (patch-2.4.20-mh17) e recompilar o Kernel 2.4.20. Este Kernel não é o 2.4.20-8 que vem com o Red Hat. Em seguida serão instalados os pacotes do bluez-libs e bluez-utils. Para iniciar o procedimento, assume-se há conexão com a Internet, que se tenha os privilégios de superusuário (root) e que o dispositivo Bluetooth USB esteja desconectado.

Passo 1: Obtendo os arquivos necessários.

Na página da Internet [29], o Kernel do Linux versão 2.4.20 pode ser obtido. Um arquivo pode ser o linux-2.4.20.tar.gz. Coloque o Kernel 2.4.20 no seguinte caminho:

Diretório : /usr/src/

Com o comando :

```
root#   tar –zxvf linux-2.4.20.tar.gz
```

um novo diretório será criado, contendo o kernel 2.4.20 :

Diretório criado: /usr/src/linux-2.4.20/

Na página do Bluez, em [28], os seguintes arquivos foram obtidos: o patch-2.4.20-mh17.tar.gz, o bluez-libs-2.8.tar.gz e o bluez-utils- 2.8.tar.gz. Coloque o patch-2.4.20-mh17.tar.gz no mesmo do kernel 2.4.20, no diretório /usr/src/linux-2.4.20/ e faça

Diretório : /usr/src/linux-2.4.20/

```
root#   tar –zxvf patch-2.4.20-mh17.tar.gz
```

Passo 2: Aplicando o patch, configurando a instalação do Kernel

Aplique o patch no kernel 2.4.20, no diretório /usr/src/linux-2.4.20/, e limpe antigos arquivos .o, com os comandos:

Diretório : /usr/src/linux-2.4.20/

```
root# patch –p1 < patch-2.4.20-mh17
root# make mrproper
```

Com isso, correções foram aplicadas ao Bluez do kernel 2.4.20 e ele está pronto para ser configurado. Ainda no diretório /usr/src/linux-2.4.20/ vamos configurar o Kernel, utilizando os comandos make menuconfig ou make xconfig. Esses comandos geram um arquivo .config, que contém as opções do kernel, como o suporte ao Bluetooth. Entretanto, configurar esse arquivo pode ser um tanto frustrante dado o número de opções possíveis. Uma boa maneira de começar esse trabalho, é ir até o diretório /boot/ e copiar o arquivo config-2.4.20-8, que é o arquivo de configuração da instalação do Red Hat, para o diretório /usr/src/linux-2.4.20/:

Diretório : /boot/

```
root# cp config-2.4.20-8 /usr/src/linux-2.4.20/
```

Diretório : /usr/src/linux-2.4.20/

```
root# cp config-2.4.20-8 .config
```

Agora, quando o comando make menuconfig ou make xconfig forem utilizados, não haverá tantos detalhes para se configurar. Faça então:

Diretório : /usr/src/linux-2.4.20/

```
root# make menuconfig
```

ou

```
root# make xconfig
```

Uma interface gráfica será aberta. Procure os items Bluetooth Support e Bluetooth Device Drivers, e faça as seguintes opções:

Opções no arquivo .config: n – não instalar, y – instalar no kernel, m – instalar como módulo

```
# # Bluetooth support#
CONFIG_BLUEZ=m
CONFIG_BLUEZ_L2CAP=m
CONFIG_BLUEZ_SCO=m
CONFIG_BLUEZ_RFCOMM=m
CONFIG_BLUEZ_RFCOMM_TTY=y
CONFIG_BLUEZ_BNEP=m
CONFIG_BLUEZ_BNEP_MC_FILTER=y
CONFIG_BLUEZ_BNEP_PROTO_FILTER=y
```

```
# # Bluetooth device drivers #
CONFIG_BLUEZ_HCIUSB=m
CONFIG_BLUEZ_HCIUSB_SCO=y
CONFIG_BLUEZ_HCIUART=m
CONFIG_BLUEZ_HCIUART_H4=y
CONFIG_BLUEZ_HCIUART_BCSP=y
CONFIG_BLUEZ_HCIUART_BCSP_TXCRC=y
CONFIG_BLUEZ_HCIDTL1=m
CONFIG_BLUEZ_HCIBT3C=m
CONFIG_BLUEZ_HCIBLUECARD=m
CONFIG_BLUEZ_HCIBTUART=m
CONFIG_BLUEZ_HCIVHCI=m
```

O importante é instalar os módulos (com as opções m ou y) do Bluez, L2CAP e SCO, através do CONFIG_BLUEZ, CONFIG_BLUEZ_L2CAP e CONFIG_BLUEZ_SCO.

São também essenciais os drivers do HCI USB, para que o sistema suporte os dispositivos *Bluetooth* USB. Coloque as opções m ou y no CONFIG_BLUEZ_HCIUSB e CONFIG_BLUEZ_HCIUSB_SCO (necessário se quisermos transportar dados de Audio SCO pelo HCI USB).

Passo 3: Compilando o Kernel

Depois da configuração, basta fazer:

Diretório : /usr/src/linux-2.4.20/

```
root# make dep
root# make bzImage
root# make modules_install
```

O último passo pode ser feito com:

Diretório : /usr/src/linux-2.4.20/

```
root# make install
ou
root# cp arch/i386/boot/bzImage   /boot/newKernel
```

Se utilizarmos o comando “make install”, basta em seguida reiniciar o computador, pois o Boot Loader (responsável por carregar o Kernel) já estará configurado. Se utilizarmos a segunda opção (copiar manualmente o bzImage para o /boot/), é preciso configurar o Boot Loader (Lilo ou Grub). Para isso, confirme em que partição dos discos rígidos está mapeado o “/” (raiz do sistema de arquivos) e o “/boot”. Para saber onde o “/” está instalado, utilize o comando:

```
root# df –a
```

Em seguida vá para o diretório /etc/ e edite o grub.conf ou lilo.conf.

Exemplo de configuração do /etc/grub.conf: Esse é um exemplo bem específico e não se aplica de maneira geral. Nesse trabalho foi utilizado o Grub como administrador de Boot. Em um dos computadores pessoais existem 2 discos rígidos, o primeiro é utilizado pelo sistema operacional Windows e o segundo para o Linux. A saída do comando df indica que o “/” está instalado no hdb1 (segundo disco rígido, primeira partição). A partição do “/boot” não é existente e deve ser mapeada na mesma partição do “/”. Logo no arquivo /etc/grub.conf devem ser adicionadas as seguintes linhas :

Adicionar no Arquivo: /etc/grub.conf

```
Title Red Hat (2.4.20-mh17)
root(hd1, 0)
kernel    /boot/newKernel    ro root=/dev/hdb1
```

Não foi apresentado todo o conteúdo do arquivo /etc/grub.conf. O comando root(hd1,0) indica onde carregar a partição “/boot”, hd1 indica que é o segundo disco rígido (hd – hard disk , 1 - segundo disco rígido), e o valor 0 indica que é a primeira partição. O comando “kernel /boot/newKernel ro root=/dev/hdb1”, carrega o Kernel indicado em /boot/newKernel. O ro root indica com hdb1 que a partição onde está a raiz do sistema de arquivos “/”.

Agora devemos reiniciar o computador e verificar se não houve algum problema na configuração do Boot Loader (Lilo ou Grub) ou na configuração do Kernel. Se tudo ocorrer bem, haverá mais uma opção de sistema : um Linux Kernel 2.4.20 com suporte ao *Bluetooth*. Falta instalar os programas de usuário.

Passo 4: Instalando o bluez-libs e o bluez-utils e testando a conectividade

No diretório onde estão os arquivos bluez-libs-2.8.tar.gz e o bluez-utils-2.8.tar.gz, faça:

```
root# tar –zxvf  bluez-libs-2.8.tar.gz
root# tar –zxvf  bluez-utils- 2.8.tar.gz
```

Entre no diretório bluez-libs-2.8, e faça:

```
root# ./configure
root# make
root# make install
```

Depois de instalar o bluez-libs, repita a operação para o bluez-utils-2.8.

Agora os comando hciconfig e hcitool devem estar disponíveis. Para torná-los disponíveis para um usuário, utilize o comando "which" para descobrir os diretórios aonde foram instalados e faça:

Diretório: /usr/sbin/

```
root# chmod   u+sx   hciconfig
```

Diretório: /usr/bin/

```
root# chmod   u+sx   hcitool
```

Agora temos que carregar os módulos Bluetooth no Kernel. Os arquivos bluez.o, l2cap.o, hci_usb.o e sco.o devem ser carregados no Kernel, como descritos na seção 2.4.1. A maneira manual de se realizar esse procedimento é usar de privilégio de superusuário (root) e fazer (assume-se que os módulos do USB, usb-core.o e usb-uhci.o (ou uhci.o ou ohci.o), já estão carregados):

```
root# modprobe bluez
root# modprobe hci_usb
root# modprobe l2cap
root# modprobe sco
```

A maneira de automatizar esse procedimento para carregar módulos do Kernel é alterar o arquivo /etc/modules.conf e inserir as seguintes linhas:

Inserir no Arquivo : /etc/modules.conf

```
alias  net-pf-31       bluez
alias  bt-proto-0      l2cap
alias  bt-proto-2      sco
alias  bt-proto-3      rfcomm
alias char-major-10-250   hci_vhci
```

Depois de alterar o módules.conf, faça:

```
root# demod –a
```

Por problemas desconhecidos, os módulos l2cap.o, sco.o e rfcomm.o nunca foram automaticamente carregados, ao contrário dos outros módulos do Bluez. Nesse caso a operação deve ser manual, usando o comando modprobe. Para verificar se os módulos foram carregados, use o comando lsmod:

```
root#  lsmod
```

Se todos os módulos estiverem presentes, conecte o módulo Bluetooth USB. Com o comando dmesg, verifique se o dispositivo foi detectado pelo USB e se o HCI USB utilizado como *device driver* do dispositivo *Bluetooth* USB.

Por fim, volte ao status de usuário normal, e utilize o seguinte comando do hciconfig:

```
user $ hciconfig hci0 up
user $ hciconfig
```

Na chamada "hciconfig hci0 up", nenhuma mensagem deve aparecer no terminal, e o HCI está sendo inicializado de forma manual. Na segunda chamada, se a inteface estiver ativa, o hciconfig deve retornar o BD_ADDR do dispositivo diferente de 00:00:00:00 e apresentar estatísticas de pacotes transmitidos ou recebidos pelo HCI. Exemplo:

```
edson $ hciconfig hci0 up
edson $ hciconfig
hci0:   Type: USB
        BD Address: 00:0C:AC:2B:3C:8E ACL MTU: 192:8  SCO MTU: 64:8
        UP RUNNING PSCAN ISCAN
        RX bytes:69 acl:0 sco:0 events:8 errors:0
        TX bytes:27 acl:0 sco:0 commands:7 errors:0
```

O BD_ADDR deve ser anotado para utilizá-lo posteriormente. O comando “hciconfig hci0 up” pode ser automatizado através do daemon do HCI (hcid). O hcid inicializa automaticamente o dispositivo HCI, utilizando as configurações listadas no arquivo /etc/bluetooth/hcid.conf. Depois de editar o hcid.conf, como será descrito no capítulo 6, basta inicializar o hcid, basta usar o comando:

```
root#  hcid
```

Utilize o comando hciconfig apenas para obter o BD_ADDR e verificar a interface HCI.

Para buscar por novos dispositivos, utilize o comando “hcitool inq” para colocar o dispositivo Bluetooth USB no subestado *Inquiry* (ver seção 2.2.4.5).

```
user$  hcitool inq
```

Se houver resposta, o comando “hcitool inq” retorna o BD_ADDR de dispositivos que encontrou. Esses endereços devem ser anotados, pois são necessários para a programação via sockets.

Por último, se dois computadores pessoais estiverem utilizando o Bluez, cada um com o seu dispositivo Bluetooth e mesma versão do bluez-utils, podemos testar a transmissão de dados via ACL ou SCO, utilizando os programas scotest e l2test. Chamando um computador de A e o outro de B, e PATH_BLUEZ_UTILS como o caminho até o diretório do bluez-utils, podemos fazer um teste em que o computador B envia dados para o A (cujo dispositivo tem BD_ADDR 00:0C:AC:2B:3C:8E), através do L2CAP (e portanto vai utilizar o transporte lógico ACL):

Computador Pessoal A
Diretório: PATH_BLUEZ_UTILS/bluez-utils-2.8/test/
Comando : `root# ./l2test -r`

Computador Pessoal B
Diretório: PATH_BLUEZ_UTILS/bluez-utils-2.8/test/
Comando: `root# ./l2test –s  00:0C:AC:2B:3C:8E:`

Se houver conexão e o computador A imprimir na tela do terminal que dados estão chegando, então ambos os computadores aparentemente tem o Bluez devidamente configurados para conexões ACL. Se houver problemas de comunicação, verifique utilizando o lsmod se todos os módulos necessários foram carregados (ex. bluez, hci_usb e l2cap) e se os dois dispositivos Bluetooth já foram detectados pelo HCI (utilizando o hciconfig).

Falta agora verificar a comunicação via SCO. Faça o mesmo procedimento utilizado no teste da conexão ACL, mas utilizando o scotest ao invés do l2test.

Computador Pessoal A
Diretório: PATH_BLUEZ_UTILS/bluez-utils-2.8/test/
Comando : `root# ./scotest -r`

Computador Pessoal B
Diretório: PATH_BLUEZ_UTILS/bluez-utils-2.8/test/
Comando: `root# ./scotest –s  00:0C:AC:2B:3C:8E:`

Se houver conexão e o computador A imprimir na tela do terminal os dados que chegam, então ambos os computadores tem o Bluez devidamente configurados para conexões SCO. Se houver problemas de comunicação, verifique com o lsmod se todos os módulos necessários foram carregados (ex. bluez.o, hci_usb.o e sco.o) e se os dois dispositivos Bluetooth já foram detectados pelo HCI (utilizando o hciconfig). Se o problema persistir, verifique também se no arquivo .config, utilizado para configurar o Kernel Linux 2.4.20, contém a opção CONFIG_BLUEZ_HCIUSB_SCO = y. Do contrário, se a opção for ‘n’, o HCI USB não conseguirá transportar dados vindos do AUDIO (SCO), e o dispositivo Bluetooth USB não tem como entregar os dados ou receber das camadas de protocolo superiores.

B2 – Programação e criação de aplicações via *Bluetooth*

A criação de aplicativos que utilizem os protocolos *Bluetooth* é em muito facilitada no Linux, com o uso de uma API (*Application Programming Interface*), ou seja, uma interface entre o programa do usuário e a implementação em *software* do protocolo. Uma API conhecida é a interface Berkeley Socket que, com funções genéricas, suporta o uso de diferentes protocolos como o TCP ou UDP. Algumas dessas funções genéricas da interface Berkeley Socket são listadas a seguir:

- socket();
- bind();
- listen();
- connect();
- accept();
- send();
- recv(

A figura B.1 apresenta em amarelo os processos em espaço do Kernel, que são as implementações em software de protocolos Bluetooth, e em azul os processos em espaço de usuário que pertencem ao bluez-utils

(aplicativos para um usuário usar ou testar a comunicação *Bluetooth*). No Bluez existem 3 protocolos diferentes que podem ser acessados pela Interface Berkeley Socket: o HCI, o L2CAP e o AUDIO SCO, e estão destacados com a cor verde na figura B.1.

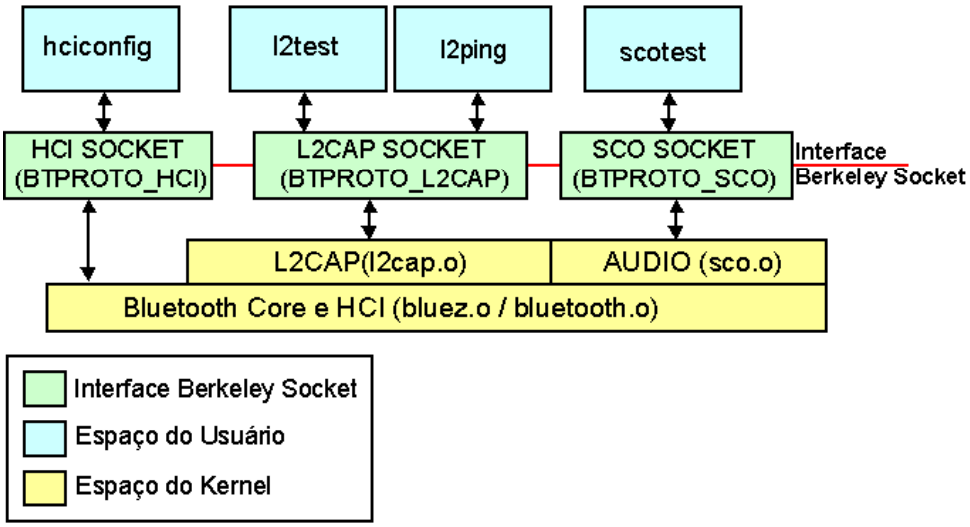


Figura B.1 – Interface Berkeley Socket entre os processos no espaço de usuário e do Kernel

Embora a API de Berkeley Socket seja uma abstração facilmente generalizada para outras linguagens de programação, esta será apenas descrita em linguagem C. Pela figura B.1, observa-se que os protocolos L2CAP e AUDIO e a interface HCI pertencem a mesma família de protocolos: a família Bluetooth. Assim é convencionado no Bluez que todos estes Sockets *Bluetooth* pertencem a PF_Bluetooth (Protocol Family Bluetooth). Para diferenciar cada protocolo da família *Bluetooth*, é utilizado um código (um número inteiro) associado com os seguintes macros: BTPROTO_HCI, BTPROTO_L2CAP e BTPROTO_SCO, como apresentado na figura B.1. A figura B.2 apresenta como se deve chamar a função socket(), para que um protocolo da família *Bluetooth* seja acessado.

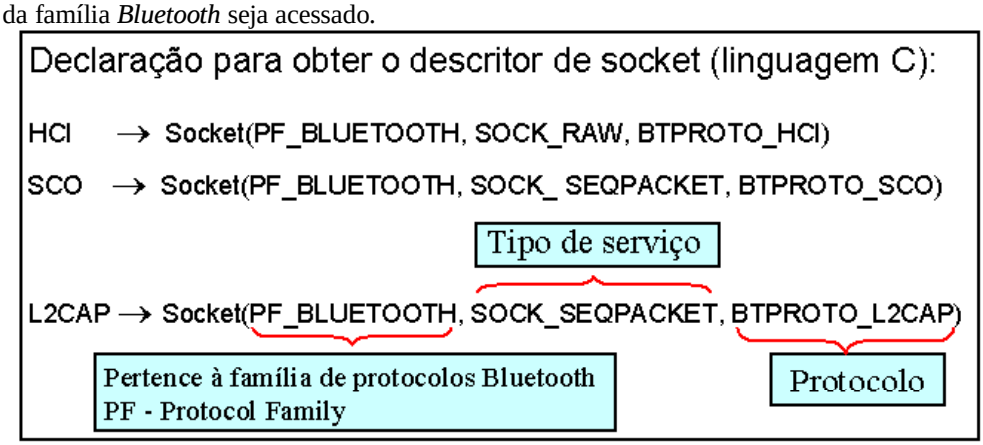


Figura B.2 – Chamada da função socket(), para cada um dos protocolos L2CAP e SCO, e para a interface HCI.

Para exemplificar a programação no Bluez, serão apresentados dois esboços de programas em linguagem C e como compilar esses programas com o GCC (Gnu Compiler Collection). Os programas são baseados no l2test.c, presente no bluez-utils.

O primeiro programa apresenta como um cliente utiliza o protocolo L2CAP para se conectar, enviar e receber dados com um servidor que espera por conexões através do L2CAP, utilizando as funções socket(), bind(), connect(), send() e recv().

O segundo programa apresenta como um servidor utiliza as funções socket(), bind(), listen() e accept(), para escutar por conexões do protocolo L2CAP na PSM igual a 10 (ver seção 2.2.7 acerca do protocolo L2CAP), aceitar a conexão. Nesse caso o cliente tem que utilizar o valor de PSM correta, ou o servidor não vai receber a requisição de conexão. A figura B.3 ilustra as aplicações.

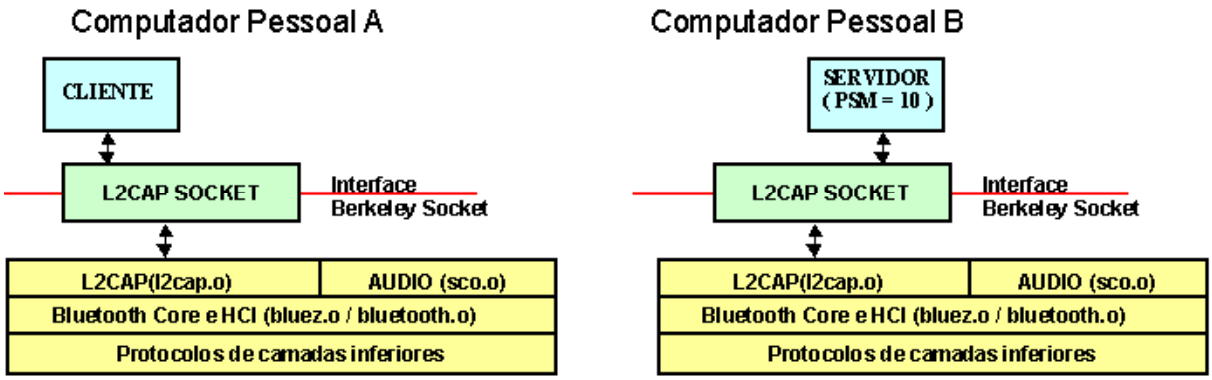


Figura B.3 – Aplicações Cliente e Servidor utilizam a interface socket L2CAP para acessar o protocolo L2CAP.

Obviamente a aplicação Cliente da figura B.3 também tem que ter um valor PSM para o protocolo L2CAP entregar os dados a aplicação correta. No entanto o valor PSM do Cliente pode ser qualquer valor não reservado e não utilizado por outra aplicação no mesmo hospedeiro, portanto apesar de não ser explicitamente apresentada, precisa ser ajustada.

Os códigos dos dois programas foram bastante simplificados e não funcionam da maneira como estão apresentados, sendo apenas um esboço. Os códigos completos estão no Apêndice A e é recomendado que o l2test seja utilizado para verificar a conectividade antes de qualquer implementação. Mais detalhes sobre a aplicação de programação utilizando o Bluez serão apresentados no capítulo 6, onde são descritos os programas utilizados nesse projeto.

Para a compilação do código e geração de executáveis, no Apêndice A pode-se encontrar os arquivos chamados de Makefile que detalham como o compilador GCC deve ser configurado, para cada um dos programas. O mais importante ao configurar o GCC é garantir que a biblioteca dinâmica libbluetooth.so seja carregada pelo executável, utilizando a opção "-lbluetooth". De maneira geral, para gerar um executável chamado de bluez_cliente a partir do arquivo bluez_cliente.c, temos que usar o seguinte comando no terminal:

```
User$ gcc bluez_cliente.c –o bluez_cliente –Wall –lbluetooth
```

- Esboço de Código do Cliente (utiliza o protocolo L2CAP)

```
/* l2cap_Cliente.c */
/*Alguns headers essenciais */
#include <bluetooth/bluetooth.h>
#include <bluetooth/l2cap.h>
#include <sys/socket.h>
/*Mais alguns headers devem ser adicionados/
void main()
{
        char end_host[20];
        sprintf(end_host, "00:0B:BC:20:21:9A");
        char end_servidor[20];
        sprintf(end_servidor,"00:0B:AA:24:31:7A");
        /*armazena os endereços BD_ADDR do host e do servidor L2CAP nas strings
        end_cliente e end_servidor */

        unsigned char buf1[20],buf2[20];
        sprintf(buf1,"teste 1 2 3");
        /*buf1 será o conteúdo transmitido, e buf2 deve armazenar um dado recebido */

        struct sockaddr_l2 local_addr;
        struct sockaddr_l2 remote_addr;
        /*Estrutura de dados que descreve o tipo de endereço (AF_Bluetooth), o número
        PSM e o endereço BD_ADDR do dispositivo Bluetooth local ou remoto)*/

        int PSM;
        /* inteiro que representa o valor PSM no protocolo L2CAP. Análogo ao conceito
        de porta, no protocolo TCP/UDP */

        int s;
        /* inteiro que vai representar o descritor de Socket */
```

```
        s = socket(PF_BLUETOOTH, SOCK_SEQPACKET, BTPROTO_L2CAP);
        /*inicia o socket para acessar o L2CAP */

        memset(&local_addr, 0, sizeof(local_addr));
        /*inicializa a estrutura */

        local_addr.l2_family = AF_BLUETOOTH;
        baswap(&local_addr.l2_bdaddr,strtoba(end_host));
        /*Escreve o tipo de endereço (AF_BLUETOOTH)
        Converte a string com o BD_ADDR do dispositivo para o formato correto com o
        strtoba()
        Copia o BD_ADDR na ordem de bytes correta, com o baswap() */

        bind(s, (struct sockaddr *) &local_addr, sizeof(local_addr));
        /*associa o socket com o endereço BD_ADDR */

        PSM = 10;
        /*Lembrando que o L2CAP faz a multiplexação entre vários protocolos
        superiores, indexados pelo valor da PSM. */

        memset(&remote_addr, 0, sizeof(remote_addr));
        remote_addr.l2_family = AF_BLUETOOTH;
        baswap(&remote_addr.l2_bdaddr, strtoba(end_servidor));
        remote_addr.l2_psm = htobs(PSM);
        /*escreve o tipo de endereço na estrutura de dados de destino, o endereço
        BD_ADDR do destino (servidor) e a sua PSM */

        connect(s, (struct sockaddr *)&remote_addr, sizeof(remote_addr));
        /*Tenta se conectar com o servidor e aguarda por resposta */

         /*Assumindo que houve sucesso na comunicação com o servidor de BD_ADDR
        00:0B:AA:24:31:7A */
        for(int i=0 ;i < 10;i++) /* Recebe e Transmite 10 vezes */
        {
              num = recv(s,buf2,sizeof(buf2),0));
              /* Recebe num bytes e armazena em buf2 */

              fprintf( stderr, "\n#byte = %d ,recebido = %s\n ",num,buf2);
              /* Imprime o número de bytes recebidos e o conteúdo */

              send(s,buf1,sizeof(buf1),0));
              /* Transmite a string armazenada em buf1 */
        }

        close(s);
        /* termina a comunicação via socket */
}
/*Fim do esboço de código de Cliente */
```

- Esboço de Código do Servidor (utiliza o protocolo L2CAP)

```
/* l2cap_Servidor.c */
/*Alguns headers essenciais */
#include <bluetooth/bluetooth.h>
#include <bluetooth/l2cap.h>
#include <sys/socket.h>
/*Mais alguns headers devem ser adicionados/
void main()
{
        char end_servidor[20];
        sprintf(end_servidor,"00:0B:AA:24:31:7A");
        /*armazena o endereço BD_ADDR do servidor L2CAP na string end_servidor */

        int s;
        /* inteiro que vai representar o descritor de socket */

        int PSM;
        PSM  =  10;
        /* inteiro que representa o valor PSM no protocolo L2CAP. Análogo ao conceito
        de porta, no protocolo TCP/UDP */
```

```
        struct sockaddr_l2 host;
        struct sockaddr_l2 target;
        /*Estrutura de dados que descreve o tipo de endereço (AF_Bluetooth), o número
        PSM e o endereço BD_ADDR do dispositivo Bluetooth local ou remoto)*/

        unsigned char buf1[20],buf2[20];
        sprintf(buf2,"testando 1 2 3");
        /*buf2 será o conteúdo transmitido, e buf1 deve armazenar um dado recebido */

        int s1;
        /* inteiro que vai representar o descritor de socket*/

        s= socket(PF_BLUETOOTH,SOCK_SEQPACKET,BTPROTO_L2CAP));
        /*inicia o socket para acessar o L2CAP*/

        memset(&host,0,sizeof(host));
        host.l2_family = AF_BLUETOOTH;
        baswap(&host.l2_bdaddr, strtoba(end_servidor));
        host.l2_psm = htobs(PSM);
        /*Escreve o tipo de endereço (AF_BLUETOOTH),
        Converte a string com o BD_ADDR do dispositivo para o formato correto com o
        strtoba(),
        Copia o BD_ADDR na ordem de bytes correta, com o baswap() */

        bind(s, (struct sockaddr *) &host, sizeof(host)));
        /*associa o socket com o endereço BD_ADDR */

        listen(s,10))
        /* Escuta a PSM 10 por pedidos de conexão */

        int len = sizeof(host);
        s1 = accept(s, (struct sockaddr *) &target, &len) );
        /* Se receber pedido por conexão, o servidor aceita e cria novo descritor,
        indexado por s1 */

        int num_bytes;
        for(int i=0 ;i < 10;i++) /* Transmite e Recebe 10 vezes */
        {
              num_bytes = send(s1, buf2, sizeof(buf2),0));
              /* Transmite num bytes de buf2 */

              num_bytes = recv(s1, buf1, sizeof(buf1),0))
              /* Recebe num bytes e armazena em buf1 */

              fprintf( stderr, "msg = %s, # bytes =%d \n ", buf1, num_bytes);
              /* imprime o conteúdo de buf1 */
        }

          close(s1);
          close(s);
          /*Finaliza os descritores de socket */
}
/* Fim do esboço de Servidor L2CAP */
```

- Gerando os executáveis:

  Gerando o executável a partir do l2cap_Cliente.c:

  User$ gcc l2cap_Cliente.c –o l2cap_cliente –lbluetooth

  Gerando o executável a partir do l2cap_Servidor.c:

  User$ gcc l2cap_Servidor.c –o l2cap_servidor –lbluetooth

Apêndice C

C.1 – Instalando uma versão do Real Time Linux/Free

O RTLinux/Free está disponível em [35]. Entretanto, antes do RTLinux ser instalado em um computador que já tenha o sistema operacional Linux, é necessário aplicar um conserto (patch), de maneira similar a descrita na seção 2.4.2. Durante o desenvolvimento deste projeto, apenas as versões de Kernel do Linux 2.4.20 e 2.4.19 eram suportados. Leia a seção 2.4.2 primeira, para depois ler esta. Muitos procedimentos em comum da seção 2.4.2 serão utilizados para instalar o RTLinux/Free, mas também se deve atualizar o Bluez do Kernel 2.4.20. Em relação ao Bluez, a única diferença é que o Kernel do Linux 2.4.20 não mais será colocado em /usr/src/linux-2.4.20/, mas em /usr/src/rtlinux/linux-2.4.20/. Siga os passos a seguir, lembrando de realizar também o que foi descrito na seção 2.4.2.

Passo 1: Obtendo os arquivos necessários.

Na página da FSMLabs, em [34], o arquivo rtlinux-3.2-pre2.tar.gz deve ser obtido e colocado no diretório /usr/src/. Faça:

```
Diretório : /usr/src/
root#   tar –zxvf rtlinux-3.2-pre2.tar.gz
Diretório criado: /usr/src/rtlinux-3.2-pre2/
```

Na página da Internet [29], o Kernel do Linux versão 2.4.20 pode ser obtido. Um arquivo pode ser o linux-2.4.20.tar.gz. Coloque o Kernel 2.4.20 no seguinte caminho:

```
Diretório : /usr/src/rtlinux-3.2-pre2/
Faça:
root#   tar –zxvf linux-2.4.20.tar.gz
Diretório criado:/usr/src/rtlinux-3.2-pre2/linux-2.4.20/
```

Passo 2: Aplicando o patch, configurando a instalação do Kernel

Vá ao diretório /usr/src/ rtlinux-3.2-pre2/patches/, para descompactar as correções:

```
Diretório : /usr/src/rtlinux-3.2-pre2/patches/
root# bzip2 –d kernel_patch-2.4.20-rtl3.2-pre2.bz2
```

Vá ao diretório /usr/src/ rtlinux-3.2-pre2/linux-2.4.20/, e aplique o *patch* do RTLinux para o kernel 2.4.20:

```
Diretório : /usr/src/rtlinux-3.2-pre2/linux-2.4.20/
root# patch –p1 < /usr/src/rtlinux-pre2/patches/kernel_patch-2.4.20
root# make mrproper
```

Esse concerto é feito para possibilitar que o Linux e o RTLinux trabalhem juntos. Deve-se também aplicar o *patch* (conserto) para o Bluez, como descrito na seção 2.4.2.

No diretório /usr/src/ Diretório criado: /usr/src/ rtlinux-3.2-pre2/linux-2.4.20/ vamos configurar o Kernel, utilizando os comandos make menuconfig ou make xconfig. Conforme descrito na seção 2.4.2, facilita-se o trabalho de configuração reutilizando-se a configuração atual, disponível em /boot/, lembrando que a distribuição Linux utilizada é o Red Hat 9.0:

```
Diretório : /boot/
root# cp config-2.4.20-8 /usr/src/rtlinux-3.2-pre2/linux-2.4.20/

Diretório: /usr/src/rtlinux-3.2-pre2/linux-2.4.20/
root# cp config-2.4.20-8 .config
```

Agora, quando o comando make menuconfig ou make xconfig forem utilizados, não haverá tantos detalhes para se configurar. Faça então:

```
Diretório:/usr/src/linux-2.4.20/
```

```
root# make menuconfig
          ou
root# make xconfig
```

Uma interface gráfica será aberta. Procure e configure os itens *Bluetooth Support* e Bluetoo*th Device Drivers*, como descrito na seção 2.4.2. Verifique também nos itens:

```
# Code maturity level options
CONFIG_EXPERIMENTAL=y

# Loadable module support
CONFIG_MODULES=y
CONFIG_MODVERSIONS=y

# Kernel hacking
CONFIG_DEBUG_KERNEL=y
CONFIG_MAGIC_SYSRQ=y
```

Passo 3: Compilando o Kernel do Linux 2.4.20

Depois da configuração, basta fazer:

```
Diretório : /usr/src/rtlinux-3.2-pre2/linux-2.4.20/
root# make dep
root# make bzImage
root# make modules_install

Faça :
root# make install
        ou
root# make bzImage
root# cp arch/i386/boot/bzImage /boot/rtzImage
```

Verifique a configuração do Boot Loader (LILO ou GRUB), para adicionar o novo Kernel do Linux, que deve ter suporte ao Bluetooth e vai funcionar em conjunto com o Real Time Linux/Free.

Agora devemos reiniciar o computador e verificar se não houve algum problema na configuração do Boot Loader (Lilo ou Grub) ou na configuração do Kernel. O passo 2, no qual se configura o Kernel, é a parte da instalação do RTLinux mais crítica. Se houver algum problema, deve-se testar outras configurações do Kernel ou procurar ajuda nos documentos disponíveis no diretório /usr/src/rtlinux-3.2-pre2/doc/ ou nas listas de email da FSMLABs, em http://www2.fsmlabs.com/pipermail/rtl/.

Passo 4: Compilando e instalando o RTLinux/Free

Mais duas tarefas devem ser realizadas nesse passo: instalar as ferramentas de usuário do *Bluetooth*, como descrito no passo 4 da seção 2.4.2, e compilar e instalar o RTLinux/Free. Para essa última tarefa, vá ao diretório /usr/src/rtlinux-3.2-pre2/ e faça:

```
Diretório: /usr/src/rtlinux-3.2-pre2/
root# make xconfig
```

Aceite a configuração padrão e faça:

```
root# make devices
root# make install
```

Reinicie o computador e selecione novamente a versão do Linux no qual foram aplicadas as correções para o RTLinux/Free. Vamos fazer um teste para verificar se o RTLinux/Free está funcionando.Utilize o comando rtlinux start:

```
root# rtlinux start
```

A seguinte tela deve aparecer, onde ‘(+)’ significa que os módulos do RTLinux foram carregados com sucesso:

```
Scheme: (-) not loaded, (+) loaded
  (+) mbuff
  (+) rtl_fifo
  (+) rtl
  (+) rtl_posixio
  (+) rtl_sched
  (+) rtl_time
  (+) rtsock
```

Em seguida, vá até o diretório /usr/src/rtlinux-3.2-pre2/examples/hello/, crie o objeto hello.o com o comando make, e carregue o hello.o :

```
Diretório: /usr/src/rtlinux-3.2-pre2/examples/hello
root# make
root# rtlinux start hello
            ou
root# insmod hello.o
```

Utilize os comando “lsmod” ou “rtlinux status”, para verificar se o módulo hello.o foi carregado. Em seguida faça:

```
root# dmesg
```

Utilizando o comando dmesg, devem aparecer mensagens que o módulo hello imprime a cada 0.5 segundo. Para terminar a execução do hello.o, utilize qualquer um dos dois comandos:

```
root# rmmod hello.o
ou
root# rtlinux stop hello
```

C.2 – Programação e Comunicação entre processos

Para esse projeto, é necessário apenas criar um programa para o RTLinux que possibilite 3 coisas: a execução de uma tarefa periódica, operações com números com ponto flutuante e a comunicação com processos do Linux através das RT-Fifos.

Também é importante criar um arquivo Makefile para ser utilizado pelo comando “make” e assim gerar o módulo (arquivo do tipo objeto “.o”) que será carregado no RTLinux. Para isso, exemplos de arquivos Makefile estão disponíveis no apêndice A.

Exemplos de programação para RTLinux podem ser encontrados junto com a distribuição do RTLinux rtlinux-3.2-pre2.tar.gz, no diretório ../rtlinux-3.2-pre2/examples/, e embora não sejam documentados, servem de ponto de partida para aprender a programar.

Todo módulo do RTLinux deve declarar as seguintes funções: init_module( ), cleanup_module() e rotinas que serão executadas como threads.

A função init_module( ) deve ser utilizada para criar RT-Fifos, com o comando rtf_create(,), e para criar threads a partir do comando pthread_create(, , ,).

A função cleanup_module() é chamada para destruir as RT-Fifos, com o comando rtf_destroy( ), e para liberar os recursos reservados para as threads criadas no init_module( ) através do comando pthread_delete_np( ).

A seguir serão apresentados 3 exemplos de programação para RTLinux, cada um apresentando um aspecto necessário para implementar:

- uma tarefa periódica (seção 3.4.1);
- uma tarefa que utilize ponto flutuante (seção 3.4.2);
- uma tarefa que utilize RT-Fifo (seção 3.4.3);

C.3 – Primeiro Exemplo: Executando tarefas periódicas

A seguir é apresentado um código que executa periodicamente, com período de 5 ms, imprimindo na tela o tempo entre duas execuções consecutivas. São feitos comentários acerca de cada uma das funções e dados utilizados. Para compilar esse programa é necessário adaptar um dos Makefiles específicos para o RTLinux, apresentados no apêndice A. Se esse módulo for carregado, basta usar o comando “dmesg” para visualizar a saída do programa.

```
/* Tarefa_periodica.c */
/*cabeçalhos necessários*/
```

```
#include <time.h>
#include <rtl.h>
#include <rtl_time.h>
#include <pthread.h>
#include <rtl_sched.h>
#include <rtl_debug.h>

pthread_t thread1; /*estrutura de dados que deve ser associada ao código da função
tarefa_periodica, que descreve a tarefa executada por uma thread*/

void * tarefa_periodica(void); /*declaração do protótipo da função que será
executada por uma thread*/

void * tarefa_periodica(void)
{

       hrtime_t elapsed_time,now,last; /*armazena o tempo em nanosegundos*/

       struct sched_param p; /*struct necessário para ajustar parâmetros da
       thread como a prioridade (0-10000)*/

       p . sched_priority = 999;/*ajustando a prioridade*/

       pthread_setschedparam (pthread_self(), SCHED_FIFO, &p);
       /*ajustando  parâmetros, como a política do agendador (FIFO), e a
       prioridade*/

       pthread_make_periodic_np(pthread_self(),gethrtime(),5000000);
       /*fazendo com que a thread seja executada periodicamente, com período    de
5 ms*/

       last = clock_gethrtime(CLOCK_REALTIME);
       int iterador=0;

       /*cada iteração do loop é executada a cada 5 ms*/
       for(;iterador < 9999999;iterador++ )
       {
              pthread_wait_np ();/* a thread é posta para dormir e vai
       acordar dentro de 5 ms*/

              now = clock_gethrtime(CLOCK_REALTIME);
              elapsed_time = now - last;
              last =now;
              rtl_printf("%Ld \n",(long long)elapsed_time);
              /*apresenta o tempo entre duas execuções consecutivas, em
       nanosegundos */
       }
       return 0;
}

/*funções obrigatórias init_module( ) e cleanup_module( )*/
int init_module(void) {
return pthread_create(&thread1,NULL,(void*)tarefa_periodica,(void *) NULL);
/*cria a thread*/
}

void cleanup_module(void) {
       pthread_delete_np (thread1);
       /*libera os recursos reservados a thread*/
}
/*Fim de Tarefa_periodica.c */
```

C.4 – Segundo Exemplo: Operações com Ponto Flutuante

O RTLinux, assim como o Linux, não permite o uso de números em ponto flutuante de maneira implícita, sendo necessário ativar essa opção através de comando específicos no caso do RTLinux. Isso ocorre para não incentivar o uso de números em ponto flutuante, cujas operações são consideradas muito lentas para serem utilizadas em módulos do Kernel (tanto do Linux quanto do

RTLinux). Vamos utilizar o programa anterior, da seção 3.4.1, para exemplificar uma tarefa periódica que realiza operações como o cálculo de senos e co-senos (mas não os imprime, pois o rtl_printf não tem suporte a números em ponto flutuante). Nesse exemplo a biblioteca matemática (de cabeçalho math.h) é utilizada pelo módulo para calcularmos sin() e cos().

```
/* Tarefa_periodica_pt_flutuante.c */
/*cabeçalhos necessários*/
#include <time.h>
#include <rtl.h>
#include <rtl_time.h>
#include <pthread.h>
#include <rtl_sched.h>
#include <rtl_debug.h>

/*cabeçalho para calcular um seno ou co-seno. É necessário carregar a biblioteca
matemática com –lm, no momento de compilar o código */
#include <math.h>


pthread_t thread1; /*estrutura de dados que deve ser associada ao código da função
tarefa_periodica_pt_flutuante, que descreve a tarefa executada por uma thread*/

void * tarefa_periodica_ptf(void); /*declaração do protótipo da função que será
executada por uma thread*/

void * tarefa_periodica_ptf(void)
{

       pthread_self()->uses_fp = 0;
       pthread_setfp_np (pthread_self(), 1);
       /*essas duas linhas são necessárias para garantir suporte a números em ponto
       flutuante*/

       struct sched_param p; /*struct necessário para ajustar parâmetros da
       thread como a prioridade (0-10000)*/

       p . sched_priority = 999;/*ajustando a prioridade*/

       pthread_setschedparam (pthread_self(), SCHED_FIFO, &p);
       /*ajustando  parâmetros, como a política do agendador (FIFO), e a
       prioridade*/

       pthread_make_periodic_np(pthread_self(),gethrtime(),5000000);
       /*fazendo com que a thread seja executada periodicamente, com período    de
5 ms*/

       last = clock_gethrtime(CLOCK_REALTIME);
       int iterador=0;

       double x,y,t;
       x = y = t = 0.0;

       /*cada iteração do loop é executada a cada 5 ms*/
       for(;iterador < 9999999;iterador++ )
       {
              pthread_wait_np ();/* a thread é posta para dormir e vai
       acordar dentro de 5 ms*/

              x = sin(t);
              y = cos(t);
              t = t + 0.005;
       }
       return 0;
}

/*funções obrigatórias init_module( ) e cleanup_module( )*/
int init_module(void) {
return pthread_create(&thread1,NULL,(void*)tarefa_periodica_ptf,(void *) NULL);
```

```
/*cria a thread*/
}

void cleanup_module(void) {
       pthread_delete_np (thread1);
       /*libera os recursos reservados a thread*/
}
/*Fim de Tarefa_periodica_pt_flutuante.c */
```

C.5 – Terceiro Exemplo: Filas Fifo

Nesse exemplo, vamos fazer uma tarefa periódica que escreve numa fila RT-Fifo e lê outra. Quando um processo do Linux escreve numa RT-Fifo, o RTLinux envia um sinal para o processo do RTLinux que criou a RT-Fifo, logo é necessário cria uma sub-rotinas de tratamento desse sinal, para atender a esse evento.

```
/* Tarefa_periodica_rtfifo.c */
/*cabeçalhos necessários*/
#include <time.h>
#include <rtl.h>
#include <rtl_time.h>
#include <pthread.h>
#include <rtl_sched.h>
#include <rtl_debug.h>


pthread_t thread1; /*estrutura de dados que deve ser associada ao código da função
tarefa_periodica_rtfifo, que descreve a tarefa executada por uma thread*/

void * tarefa_periodica_rtfifo(void); /*declaração do protótipo da função que será
executada por uma thread*/

void * tarefa_periodica_rtfifo(void)
{

       struct sched_param p; /*struct necessário para ajustar parâmetros da
       thread como a prioridade (0-10000)*/

       p . sched_priority = 999;/*ajustando a prioridade*/

       pthread_setschedparam (pthread_self(), SCHED_FIFO, &p);
       /*ajustando  parâmetros, como a política do agendador (FIFO), e a
       prioridade*/

       pthread_make_periodic_np(pthread_self(),gethrtime(),5000000);
       /*fazendo com que a thread seja executada periodicamente, com período    de
5 ms*/

       last = clock_gethrtime(CLOCK_REALTIME);
       int iterador=0;
       int tx,rx;
       tx = rx = 0;

       /*cada iteração do loop é executada a cada 5 ms*/
       for(;iterador < 9999999;iterador++ )
       {
              pthread_wait_np ();/* a thread é posta para dormir e vai
       acordar dentro de 5 ms*/

              tx = tx + 1;
              rtf_put(12,&tx,sizeof(tx));
              /* escreve tx na fila rt-fifo 12, e deve ser lida por um processo do
              Linux em /dev/rtf12 */

              if( (rtf_get(13,&rx,sizeof(rx))) > 0) /*se não houver nada na rtfifo
              13, rtf_get retorna um número menor que 0. Se algum dado foi escrito,
```

```
            rtf_get é maior que 0, e o dado recebido vai ser impresso (ver com o
            comando 'dmesg'). */
            {
                  rtl_printf("numero recebido %d\n", rx);
            }

      }
      return 0;
}

/*cria sub-rotina de tratamento de sinais associados a uma rt-fifo*/
int fifo_handler1(unsigned int fifo{

      int r, msg;

      r = rtf_get(fifo, &msg, sizeof(msg));
      r = rtf_put(fifo, &msg, sizeof(msg));
      /*essa rotina espera que o processo do Linux escreva apenas um número do tipo
      inteiro, e coloca esse número de volta na fila. Essa rotina serve apenas para
      tratar o sinal enviado pelo RTLinux, os dados serão tratados pela thread
      periódica*/

      return 0;
}


/*funções obrigatórias init_module( ) e cleanup_module( )*/
int init_module(void) {

      rtf_destroy(12);
      rtf_destroy(13);
      /*por garantia, certifica-se que estas rt-fifos vão ser inicializadas,
      destruindo elas.*/


      rtf_create(12, 4000);/*vamos utilizar a rt-fifo 12 para escrever, e vamos
      reservar 4000 bytes de espaço de memória*/

      rtf_create(13, 4000);/*vamos utilizar a rt-fifo 13 para ler dados enviados
      pelo Linux*/

      /*Para o Linux, um programa deve abrir e ler o arquivo /dev/rtf12 e abrir e
      escrever no arquivo /dev/rtf13 */

      rtf_create_handler(13, fifo_handler1);/*a sub-rotina fifo_handler1 está
      associada a rt-fifo 13, esperando por um sinal que indica que um processo do
      Linux escreveu em /dev/rtf13 */


return pthread_create(&thread1,NULL,(void*)tarefa_periodica_rtfifo,(void *) NULL);
/*cria a thread*/
}

void cleanup_module(void) {

      printk("%d\n", rtf_destroy(12));
      printk("%d\n", rtf_destroy(13));
      /*destrói as rt-fifos criadas pelo init_module()*/

      pthread_delete_np (thread1);
      /*libera os recursos reservados a thread*/
}
/*Fim de Tarefa_periodica_rtfifo.c */
```

C.6 – Depurando o programa e evitando que o PC trave

Um importante aviso para a programação em RTLinux é que como um módulo é carregado no Kernel, qualquer problema pode afetar todo o sistema operacional, causando o travamento da máquina. Para evitar ou minimizar esse problema, o RTLinux/Free dispõe de um módulo chamado de rtl_debug.o que impede que o sistema trave sempre que ocorrer algum problema, o que não é difícil de ocorrer já que a programação é feita em linguagem C e certos problemas não são muito aparentes, como por exemplo tentar usar um número em ponto flutuante, sem ativar o suporte ao mesmo.

Seguindo os procedimentos apresentados na seção 3.3, depois de utilizar o comando "rtlinux start", vá ao diretório /usr/src/rtlinux-3.2-pre2/debugger/ e carregue o módulo rtl_debug antes de carregar qualquer outro módulo:

```
Diretório : /usr/src/rtlinux-3.2-pre2/debugger/
root# insmod rtl_debug.o
```

Em seguida utilize o comando lsmod para verificar se o módulo foi carregado. Com esse recurso, e os descritos em /usr/src/rtlinux-3.2-pre2/debug/README, pode-se evitar não apenas o travamento da máquina, como realizar a depuração do programa através de programas de depuração como o programa gdb.