RONALDO APARECIDO SILVA

PROPOSTA DE ARQUITETURA DE COMUNICAÇÃO PARA SISTEMAS EMBARCADOS BASEADA NO PROTOCOLO PUBLISHER/SUBSCRIBER

FLORIANÓPOLIS
2007

UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO
EM ENGENHARIA ELÉTRICA

PROPOSTA DE ARQUITETURA DE COMUNICAÇÃO PARA SISTEMAS EMBARCADOS BASEADA NO PROTOCOLO PUBLISHER/SUBSCRIBER

Dissertação submetida à
Universidade Federal de Santa Catarina
como parte dos requisitos para a
obtenção do grau de Mestre em Engenharia Elétrica.

RONALDO APARECIDO SILVA

Florianópolis, Novembro de 2007.

PROPOSTA DE ARQUITETURA DE COMUNICAÇÃO PARA SISTEMAS EMBARCADOS BASEADA NO PROTOCOLO PUBLISHER/SUBSCRIBER

RONALDO APARECIDO SILVA

'Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Engenharia Elétrica, Área de Concentração em *Controle, Automação e Informática Industrial*, e aprovada em sua forma final pelo Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Santa Catarina.'

Prof. Leandro Buss Becker, Dr.
Orientador

Kátia Campos de Almeida, Ph.D.
Coordenadora do Programa de Pós-Graduação em Engenharia Elétrica

Banca Examinadora:

Leandro Buss Becker, Dr.
Presidente

Joni da Silva Fraga, Dr.

Mario Dantas, PhD.

Aos meus Pais e a minha namorada Juliana.

AGRADECIMENTOS

Agradeço aos meus pais pela educação que me proporcionaram, concedendo-me muito além do que tiveram. Agradeço em especial à minha mãe, Dona Nice, por sempre estar disposta a contribuir com palavras de apoio e incentivo durante as fases mais difíceis da concepção deste trabalho. Aos meus irmãos Silvio e Andreia o mais profundo e sincero muito obrigado, por sempre estarem do meu lado me auxiliando nos momentos mais difícies e por me darem maravilhosos sobrinhos, João Victor, Pedro Henrique e Manuela.

Agradeço ao meu Orientador, Prof. Leandro Buss Becker, pelo apoio e dedicação, por sempre estar disposto a colaborar, indicando caminhos para solucionar os obstáculos que surgiram nesse período, demonstrando ser uma pessoa serena que realmente me orientou no desenvolvimento deste trabalho.

Agradeço aos amigos de longa data, Alex (Band), Rodrigo (Bruxo), Juliano (Gersting, Juba), Alexandre (Sapão), Gustavo Xavier (Custela), Gustavo Venerato, Juliano (Juca Bala, Juquinha), Juliano Roberto (Buda), Bruno (Jabá) pelo apoio que sempre me deram e por me aturarem por tanto tempo.

Agradeço também ao amigo Emerson Mello, pela amizade, e ao professor Joni da Silva Fraga, pelo apoio. A todos os professores do Departamento de Automação e Sistemas (DAS), o meu muito obrigado pelos ensinamentos. Aos membros dessa banca examinadora que contribuíram na revisão deste trabalho, colaborando com sugestões, o meu agradecimento.

Agradeço, também ao pessoal do Laboratório de Sistemas de Potência (LABSPOT), no qual trabalhei durante esse tempo, em especial aos professores Antônio Simões, Katia Almeida, Hans Zürn, Roberto Salgado e Jacqueline Rolim pela compreensão em determinados momentos em que precisei ausentar-me das obrigações do trabalho.

Por fim, agradeço a todas as pessoas que contribuíram com este trabalho de forma direta, como Rafael Cancian, pelos passos iniciais na rede CAN; Marcelo Sobral, com todo tipo de ajuda e esclarecimentos; Filipe Sá, pelas contribuições com a plataforma Coldfire; Tiago Semprebom, pela contribuição gráfica nas figuras; ou, de forma indireta, pelos momentos de descontração e muita bebida, os amigos André Piazza, Eduardo Alchieri, Rodrigo Machado, Marciano Bortoli, Lauro, Benito, Luciano, Kleber (Cambara), Renato (Sem braço) e Silvio (Binho).

E para concluir, agradeço também ao Programa de Pós-Graduação em Engenharia Elétrica, pela oportunidade de desenvolver minha pesquisa na UFSC.

Resumo da Dissertação apresentada à UFSC como parte dos requisitos necessários para obtenção do grau de Mestre em Engenharia Elétrica.

PROPOSTA DE ARQUITETURA DE COMUNICAÇÃO PARA SISTEMAS EMBARCADOS BASEADA NO PROTOCOLO PUBLISHER/SUBSCRIBER

RONALDO APARECIDO SILVA

Novembro/2007

Orientador: Leandro Buss Becker, Dr
Área de Concentração: Controle, Automação e Informática Industrial
Palavras-chave: Sistemas Embarcados, *Publisher/Subscriber*, Middleware, Controle Distribuído
Número de Páginas: xiv + 105

Esta dissertação apresenta uma proposta de arquitetura denominada *Distributed Object-based Architecture for Controlling Autonomous Vehicles* - DOCAS. DOCAS é baseada no protocolo *Publisher/Subscriber*, e tem sua utilização voltada para o controle de veículos autônomos. Um sistema distribuído baseado no protocolo *Publisher/Subscriber* tem como características o desacoplamento entre seus componentes, comunicações anônimas com mensagens curtas e identificadas por conteúdo e também comunicações muitos-para-muitos. Com a utilização da arquitetura proposta, torna-se possível realizar comunicação através de diferentes protocolos de forma anônima, ou seja, pode-se constituir um sistema que integra vários protocolos de rede para interligar diferentes plataformas de *hardware* (nodos da rede) interagindo através de eventos. Além disso, a arquitetura proposta permite programar, de uma forma simplificada, a comunicação entre os elementos envolvidos na aplicação. Esta proposta também discute alternativas para representar o problema num nível de abstração mais elevado. Investigam-se as características de um sistema de controle dessa natureza, levantando-se questões de modelagem que contemplem tais características. Para validar e exemplificar a arquitetura proposta é mostrado um estudo de caso voltado para o controle de um veículo autônomo.

Abstract of Dissertation presented to UFSC as a partial fulfillment of the requirements for the degree of Master in Electrical Engineering.

ARCHITECTURE FOR EMBEDDED SYSTEMS CONTROL BASED ON THE PUBLISHER/SUBSCRIBER PROTOCOL

RONALDO APARECIDO SILVA

November/2007

Advisor: Leandro Buss Becker, Dr
Area of Concentration: Control, Automation and Industrial Computing
Key words: Embedded System, Publisher/Subscriber, Middleware, Distributed Control
Number of Pages: xiv + 105

This work presents the architecture named Distributed Object-based Architecture for Controlling Autonomous Vehicles - DOCAS. DOCAS is based on the Publisher/Subscriber protocol and its utilization is aimed for autonomous vehicles control. A distributed embedded system based on a Publisher/Subscriber protocol is characterized by the uncoupling components between its components, short and content-based messages with anonymous communications, and many-to-many communications. The proposed architecture allows anonymous communication through different protocols. Thereby, a system that integrates several network protocols can be used to interconnect different hardware platforms (network nodes), allowing event-based interaction. Moreover, the proposed architecture allows programs to be written in a simplified way, making it easier to describe the communications among the elements involved in the application. This work also considers alternatives to represent the problem in a higher abstraction level. The characteristics of a system with such features is investigated in a case study in order to validate and exemplify the proposed architecture.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Códigos

Lista de Abreviaturas

ABS	*Anti-lock Braking System*
ACC	*Adaptive Cruise Control*
CAN	*Controller Area Network*
CISC	*Complex Instruction Set Computer*
COSMIC	*COoperating SMart devICes*
CPBM	*Controle Preditivo Baseado em Modelo*
CRC	*Cyclic Redundant Check*
CSMA/DCR	*Carrier Sense Multi-Access with Deterministic Collision Resolution*
DLC	*Data Length Code*
DOCAS	*Distributed Object-based architeture for Controlling Autonomous vehicleS*
ECB	*Event Channel Broker*
ECH	*Event Channel Handler*
ELDK	*Embedded Linux Development Kit*
EOF	*End Of Frame*
ESML	*Embedded Systems Modelling Language*
ESP	*Electronic Stability Program*
ECU	*Electronic Control Unit*
FPU	*Float Point Unit*
IFS	*Intermission Frame Space*
LIN	*Local Interconnect Network*
LSB	*Least Significant Bit*
MDD	*Model Driven Development*
MSB	*More Significant Bit*
MBPC	*Model Based Predictive Control*
NFS	*Network File System*
PCAN	*Peak CAN*
POWERPC	*Power Optimization With Enhanced RISC - Performance Computing*
PLC	*Programmable Logic Controllers*
RISC	*Reduce Instruction Set Computer*
RMI	*Remote Method Invocation*
RPC	*Remote Procedure Call*

RSI	*Request Short Node ID Message*
RTR	*Remote Transmission Request*
SAE	*Society of Automotive Engineers*
SSI	*Supply Short Node ID Message*
SOC	*System-On-Chip*
SOF	*Start Of Frame*
TFTP	*Trivial FileTransfer Protocol*
UART/SCI	*Universal Asynchronous Receiver/Transmitter/Serial Communications Interface*
U-boot	*Universal Boot Loader*
UML	*Unified Modeling Language*

Capítulo 1

Introdução

1.1 Motivação

Com o avanço tecnológico na área de microeletrônica, torna-se mais comum encontrar processadores embarcados em algum dispositivo ou máquina. Esses componentes ganharam poder de processamento e diminuíram expressivamente o tamanho e o custo, tornando-se mais acessíveis. Hoje em dia, há muito mais processadores dedicados a uma função específica do que fazendo parte de computadores de propósito geral (WOLF, 2001), (TENNENHOUSE, 2000), (PARET, 2007).

Geralmente os sistemas embarcados são compostos de múltiplos dispositivos interligados através de uma rede, de forma que tenham certo grau de autonomia e cooperem para a realização de alguma tarefa. Como exemplo, citam-se os sistemas embarcados em veículos automotivos (POP; ELES; PENG, 2004), aeronaves e equipamentos de controle (LIU, 2000). Tais sistemas geralmente possuem requisitos de tempo real, sendo relativa a interação entre eles. Além disso, na maioria das vezes, esses sistemas possuem limitações de *hardware* as quais devem ser observadas durante o projeto do software. A comunicação é feita muitas vezes em redes heterogêneas, dificultando a interação entre os dispositivos conectados.

A distribuição dos componentes implica a adoção de um modelo de interação entre eles. Dentre os modelos descritos em (BECKER et al., 2001), o modelo *Publisher/Subscriber* apresenta propriedades interessantes, como o desacoplamento entre os objetos distribuídos, comunicações anônimas, mensagens diferenciadas por conteúdo (denominadas eventos) e comunicação muitos-para-muitos (possivelmente explorando capacidades de difusão da estrutura da rede). Portanto, torna-se desejável conceber um sistema embarcado com dispositivos distribuídos e interligados por uma rede, e interagindo por eventos segundo um protocolo *Publisher/Subscriber*.

Muitos aspectos relacionados à comunicação podem ser abstraídos utilizando um *middleware* como suporte, tais como tipos de protocolos e programação de baixo nível. Assim, o programador pode focar seu trabalho em componentes que usam o suporte oferecido pelo *middleware*. Portanto, a adoção de um *middleware* que dê suporte ao modelo de comunicação *Publisher/Subscriber* se torna essencial. O *middleware* COSMIC (CASIMIRO; KAISER; VERISSIMO, 2004) foi desenvolvido para dar esse tipo de suporte.

Esta dissertação visa propor uma arquitetura de comunicação, denominada *Distributed Object-based architeture for Controlling Autonomous vehicleS* (DOCAS), que seja adequada para o projeto de sistemas embarcados distribuídos, cooperativos e tempo real. É objetivo, também, o desenvolvimento de um protótipo que comprove a eficiência da proposta. Como conseqüência, deverá ser possível agilizar o processo de desenvolvimento de tais sistemas, aumentando o nível de abstração dos modelos.

1.2 Objetivo

O presente trabalho tem por objetivo propor uma arquitetura de comunicação voltada para sistemas embarcados distribuídos. Conforme sugerido, a arquitetura em questão deverá focar aspectos relacionados com o modelo de comunicação *Publisher/Subscriber*. Os elementos definidos na arquitetura proposta deverão ser validados através de simulação e de estudos de caso na área de veículos autônomos. Além disso, deve-se apresentar alternativas para facilitar a modelagem de sistemas desta natureza, elevando ao máximo o nível de abstração para o projetista, focando seu trabalho nos comportamentos dos componentes do sistema, isto é, abstraindo aspectos da comunicação baixo nível.

1.3 Estrutura da Dissertação

A organização deste trabalho dá-se da seguinte forma:

- O segundo capítulo descreve questões pertinentes de outras tecnologias que estão associadas a este, apresentado discussões presentes na literatura, além da revisão de tecnologias relacionadas ao tema.

- O terceiro capítulo apresenta a arquitetura proposta, iniciando pela sua descrição e, em seguida, apresentando aspectos sobre como modelar os componentes da arquitetura.

- O quarto capítulo apresenta os detalhes de implementação da arquitetura proposta em diferentes plataformas destinadas a sistemas embarcados distribuídos. Também mostra como foi realizada a sua implementação, detalhando aspectos da comunicação baixo nível.

- O quinto capítulo apresenta o estudo de caso desenvolvido para validar a arquitetura proposta. Para esse estudo de caso foi utilizado um veículo autônomo que percorre trajetórias com base em referências, tomando-se como base o algoritmo de controle preditivo baseado em modelo desenvolvido por (GOMES, 2006). Também são feitas comparações entre o modelo anteriormente projetado (de forma centralizada) e o atual (de forma distribuída).

- O sexto capítulo apresenta as conclusões finais sobre o trabalho realizado e as perspectivas futuras de pesquisa e aprimoramento.

Capítulo 2

Redes, Sistemas Distribuídos e Middleware

Neste capítulo apresenta-se algumas das diferentes tecnologias de redes utilizadas pela indústria automobilística em um veículo, os modelos de comunicação em sistemas distribuídos e o conceito de middleware. Na parte referente às tecnologias de rede automotiva mostra-se os protocolos CAN, LIN e FlexRay. Na parte que se refere às formas de comunicação entre componentes distribuídos ilustra-se os modelos Send/Receive, RPC, RMI e Publisher/Subscriber. No final, discute-se o conceito de middleware e apresenta-se o middleware COSMIC, o qual é objeto de estudo desta dissertação.

2.1 Redes Usadas em Automóveis

Os sistemas eletrônicos são utilizados pela indústria automobilística há muito tempo para controlar as várias funções existentes nos automóveis. Os primeiros sistemas de controle foram desenvolvidos de forma que cada um operasse isoladamente e fosse responsável por um determinado tipo de função no automóvel. Com o aumento dos dispositivos eletrônicos, a necessidade da interligação desses vários sistemas passou a ser cada vez maior, dando origem a barramentos de comunicação distintos. Essa diversidade de barramentos deu-se principalmente pela subdivisão das partes do automóvel.

A Sociedade de Engenharia Automotiva (*Society of Automotive Engineers* - SAE (SAE, 2007)) descreve uma classificação de redes de comunicação em função dos requisitos das aplicações automotivas.

- Classe A: A-BUS (SAE, 1993), LIN-BUS (LIN, 1999), TTP/A (SAE, 1993);
- Classe B: CAN (BOSCH, 1991), VAN e J1850 (SAE, 1993);
- Classe C: TTP/C (COMPUTERTECHNIK, 1999), TT-CAN (FüHRER et al., 2000), ByteFlight (PELLER; BERWANGER; GRIESSBACH, 1999) e FlexRay (FLEXRAY, 2005).

A seguir, nas próximas seções, apresentam-se os protocolos CAN, LIN e FlexRay.

2.1.1 CAN

O modelo *Controller Area Network* (CAN) (BOSCH, 1991) foi proposto por Robert Bosch, em 1980, para interconexão dos sistemas de controle em veículos, buscando simplificar os complexos sistemas de fios. Em 1986, a empresa BOSCH anunciou oficialmente o desenvolvimento de um sistema de comunicação entre Unidades Eletrônicas de Controle (ECU´s) em um automóvel, atendendo a um pedido feito pela empresa Mercedes. Já em 1987 surgiram os primeiros circuitos integrados para CAN, fabricados pela Intel e pela Philips, e em 1994 tornou-se um padrão internacional (ISO 11898).

Visal Geral do Protocolo CAN

CAN é um protocolo de comunicação serial síncrono utilizado para conectar dispositivos distribuídos. O sincronismo entre os módulos conectados à rede é feito em relação ao início de cada mensagem lançada ao barramento (evento que ocorre em intervalos de tempo conhecidos e regulares).

A rede é multimestre, isto é, todos os nodos podem tornar-se mestre em determinado momento, e escravo em outro; além disso é possível que mais de um nodo seja controlador, o que facilita a criação de um sistema redundante. Cada nodo tem livre acesso ao meio de transmissão podendo, enviar uma mensagem sempre que necessário ou em resposta a um evento.

CAN usa o protocolo de acesso ao meio *Carrier Sense Multi-Access with Deterministic Collision Resolution* (CSMA/DCR) (KIM; SERRO, 1995), ou seja, cada nodo, antes de enviar informação para a rede, “escuta” o que se passa nela e, se eventualmente estiver livre, envia a mensagem.

Para evitar colisão, o algoritmo verifica o barramento analisando as prioridades das mensagens. A mensagem que tiver menor prioridade cessára sua transmissão, e a de maior prioridade continuará enviando sua mensagem sem ter que reiniciá-la.

A prioridade da mensagem é especificada por identificadores únicos, que podem ser de 11 ou 29 *bits*, e que também determinam a prioridade intrínseca da mensagem ao competir com outras pelo acesso ao barramento.

Esse identificador também serve para que os receptores decidam se devem ou não processar a mensagem, já que esta, quando transmitida no barramento, não possui endereço de receptor. Quanto menor o valor binário associado, maior a prioridade.

A taxa de transmissão dos dados é inversamente proporcional ao comprimento do barramento. A taxa máxima de transmissão é 1Mbit/s que utiliza um protocolo de arbitragem e mecanismos de detecção e sinalização de erros que proporcionam a integridade das informações transmitidas.

No CAN, os dados não são representados por *bits* em nível “0” ou nível “1”; são representados por *bits* Dominantes e bits Recessivos, criados em função da condição presente nos *bits* CAN_H e CAN_L. A Figura 2.1 ilustra os níveis de tensão em uma rede CAN, assim como os *bits* Dominantes e Recessivos (GUIMARãES, 2006).

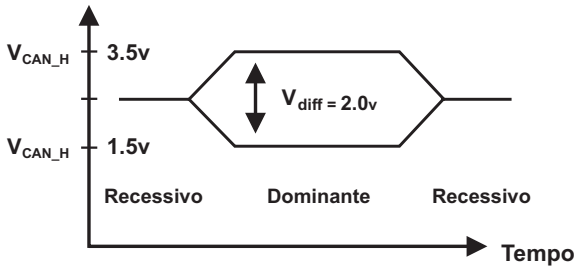

Figura 2.1: Níveis de tensão e *bits* dominantes e recessivos (GUIMARãES, 2006).

Formato dos *Frames*

O protocolo CAN suporta dois formatos para mensagens; uma diferença entre eles é quanto ao número de *bits* do identificador. No formato padrão (2.0A), o identificador possui 11 *bits*, e no formato estendido (2.0B), o identificador possui 29 *bits*.

O formato padrão, Figura 2.2, começa com um campo chamado *Start Of Frame* (SOF) composto por um único *bit* dominante; este é seguido por um campo arbitrário que consiste em um identificador de 11 *bits* (*Identifier*) e o *Remote Transmission Request* (RTR) *bit* dominante, quando requisita dados de outro(s) nodo(s) especificado(s) pelo campo identificador. Seguindo a seqüência encontra-se o campo de controle que contem *Identifier Extension* (IDE) *bit* que distingue entre o *frame* padrão e o *frame* estendido, e o *Data Length Code* (DLC) 4 *bits* indicando o total de *bytes* usados no campo de

dados (*Data Field*), que é o próximo campo o qual pode conter até 64 *bits* de dados.

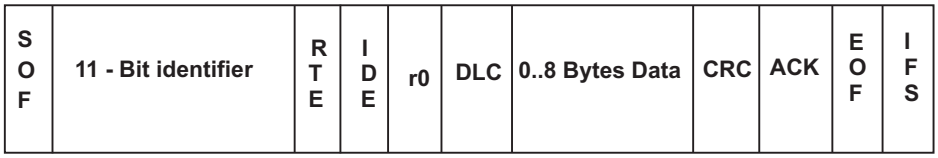

Figura 2.2: CAN 2.0A.

A integridade do *frame* é garantida pelo campo *Cyclic Redundant Check* (CRC), composto pela seqüência numérica gerada, e utilizado para verificação de erros no quadro.

O campo *ACKnowledge* (ACK) possui 2 *bits* que permitem a todo e qualquer nodo validar a mensagem, 1 *bit* para escrita e outro como delimitador. Na seqüência, há o campo *End Of Frame* (EOF) indicando o final da mensagem. O campo *Intermission Frame Space* (IFS) indica o tempo para o controlador disponibilizar o dado para a aplicação.

Já o padrão estendido, Figura 2.3, difere do formato padrão; a diferença está no identificador da mensagem; o de 11 *bits* é chamado de identificador base; o de 18 *bits* chamado de identificador de extensão. O campo IDE indica a extensão do identificador. O campo SRR substitui o campo RTR do formato padrão, que continua um *bit* dominante; agora o campo RTR contém um *bit* recessivo indicando um *frame* estendido.

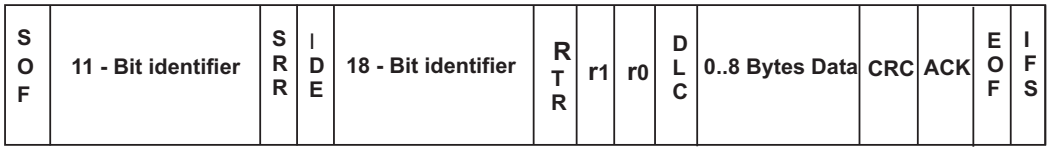

Figura 2.3: CAN 2.0B.

Para configurar um nodo ou quando um nodo necessita de algum dado é utilizado o quadro de requisição remota. O quadro remoto é idêntico ao quadro de dado exceto por não conter o campo de dados.

2.1.2 LIN

A *LOCAL INTERCONNECT NETWORK* (LIN) (LIN, 1999) é uma sub-rede de comunicação que foi projetada pelo Consórcio LIN, composto pelas empresas automotivas Volkswagen, Audi, BMW, DaimlerChrysler e Volvo, mais a empresa Motorola, fabricante de semicondutores e com especialistas

em arquitetura distribuída Volcano. Seu objetivo é especificar um padrão aberto de baixo custo para a rede automotiva, onde a largura de banda e a versatilidade do protocolo CAN não são necessários. Esse consórcio iniciou suas atividades como um grupo de trabalho em 1998 e lançou a primeira especificação em 1999.

O padrão LIN não especifica somente a transmissão de dados, mas também fornece ferramentas automatizadas, endereçando as complexas e crescentes necessidades de implementação e de manutenção de software em sistemas distribuídos. Por esta mesma razão, a especificação do LIN cobre, além da definição do protocolo, as interfaces para as ferramentas de desenvolvimento e para uma aplicação de rede independente (SPECKS; RAJNáK, 2000).

Essa rede foi desenvolvida como um complemento ao barramento CAN, sendo usada em aplicações onde o custo é crítico e as taxas de transmissão de dado são baixas. As aplicações automotivas que utilizam a tecnologia de rede LIN são: vidro elétrico, trava elétrica, iluminação, controle de clima, sensor de chuva, regulagem de assentos, entre outras que requerem baixo poder de processamento e de recursos de comunicação. Nesses sistemas o uso do LIN é mais adequado do que o uso do CAN no que diz respeito ao custo.

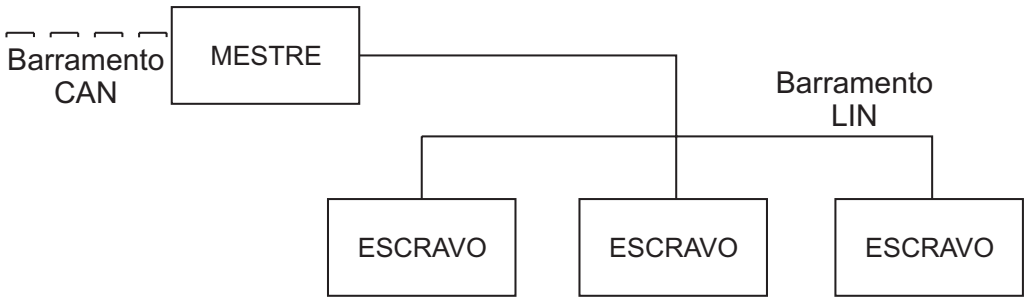

Figura 2.4: Visão geral da rede LIN (APPLICATIONS, 2007).

As principais motivações para se utilizar LIN, entre outras, devem-se ao fato de que ele possui um simples fio para transmissão/recepção de dados (baixo custo); a arquitetura do barramento conta com um único nodo mestre e vários nodos escravos, no máximo de 16 nós por rede (APPLICATIONS, 2007); implementação de baixo custo baseada no *hardware* de interface *Universal Asynchronous Receiver/Transmitter*/*Serial Communications Interface* (UART/SCI), em um software equivalente, ou como a máquina pura de estados; auto-sincronização sem cristal nos nós escravos, ou seja, apenas o mestre possui o *clock* de sincronização.

Na Figura 2.4 pode-se visualizar a integração da sub-rede LIN, através do nodo mestre, com

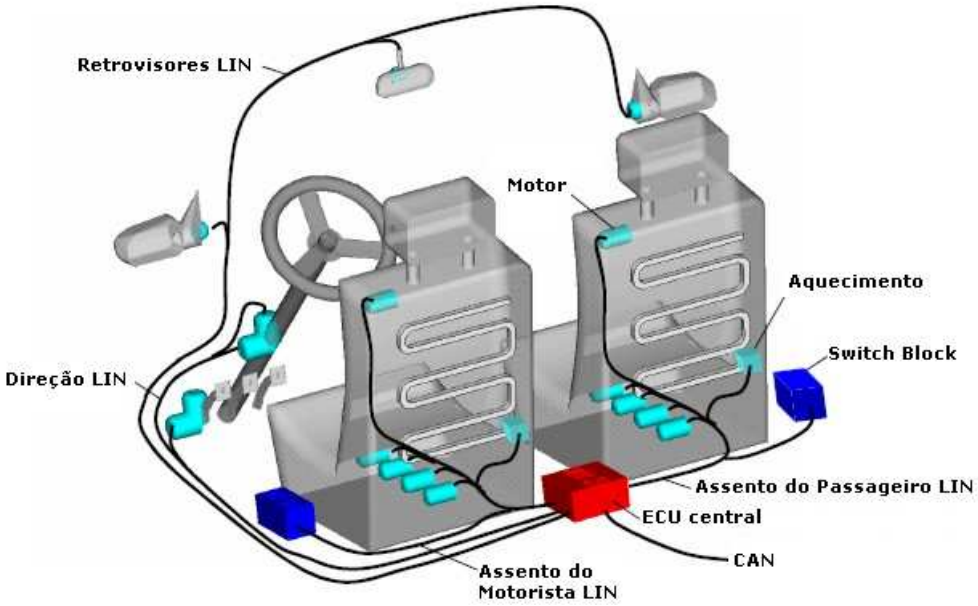

Figura 2.5: Estudo de caso - LIN (SPECKS; RAJNáK, 2000).

redes de nível superior, como do tipo CAN, estendendo os benefícios de trabalhar em rede por todos os componentes do veículo. Já na Figura 2.5 pode-se visualizar quatro redes LIN com uma Unidade Eletrônica de Controle (ECU) que também atua como um *gateway* entre as diferentes redes. O nodo mestre é também conectado a rede CAN.

Uma rede LIN é formada por um nodo mestre e um ou mais nodos escravos. Todos os nodos incluem uma tarefa escrava (tarefa esc.), usada para que eles se comuniquem com os outros nodos, que é dividida em duas funções: transmitir e receber mensagens, enquanto o nodo mestre inclui um tarefa adicional também chamada mestre (tarefa mestre). A interação entre os nodos em uma rede LIN é sempre inicializada pela tarefa mestre, como ilustrado na Figura 2.6. O cabeçalho de uma mensagem mestre inclui o identificador da mensagem (campo identificador), o *byte* de sincronização (campo sincronização) e a quebra de sincronização (campo quebra sinc.) usada para identificar o inicio do *frame*, a quebra fornece oportunidades regulares para os nodos escravos sincronizarem-se ao barramento LIN, ou entrarem na comunicação LIN em qualquer momento, por exemplo depois de "resetar"o controlador (SPECKS; RAJNáK, 2000).

O nodo mestre envia o cabeçalho da mensagem para o nodo escravo, e somente uma tarefa escrava é ativada através do identificador da mensagem. O nodo que recebeu a mensagem inicia a transmissão da mensagem de resposta com os devidos dados. A resposta inclui dois, quatro, ou oito *bytes* de dados

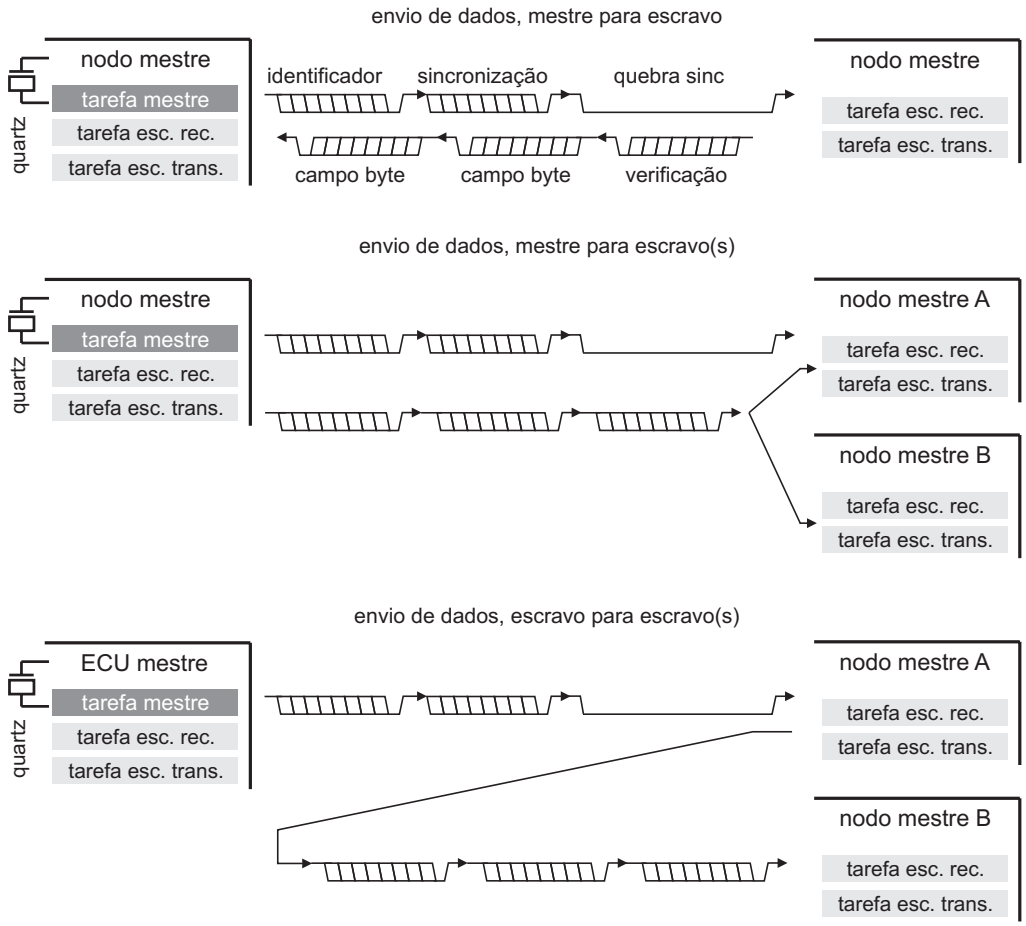

Figura 2.6: Algumas formas de comunicação de dados com LIN (SPECKS; RAJNáK, 2000).

e um *byte* de verificação (SPECKS; RAJNáK, 2000).

O identificador de uma mensagem identifica o conteúdo da mensagem, mas não o destino. Esse conceito de comunicação habilita a troca de dados de várias maneiras: do nodo mestre (usando sua tarefa esc.) para um ou mais nodos escravos, e de um nodo escravo para dois nodos mestres e/ou outros nodos escravos. É possível comunicar-se através de sinais, diretamente de um nodo escravo para outro nodo escravo sem a necessidade de roteamento através do nodo mestre, ou através da difusão da mensagem do nodo mestre para todos os nodos na rede.

Formato das Mensagens

A comunicação LIN está baseada em mensagens de formato fixo e com tamanho selecionável. Cada mensagem é construída em oito-bits (campo *byte*) com configuração 8N1, conhecida como SCI ou UART, formato de dados serial. Cada *byte* tem dez *bits*, primeiro iniciando com um *bit*

dominante (*bit* inicial), seguido por oito *bits* de dados, com o *bit* menos significativo (LSB) sendo enviado primeiro e, no final, com o *bit* recessivo (Figura 2.7).

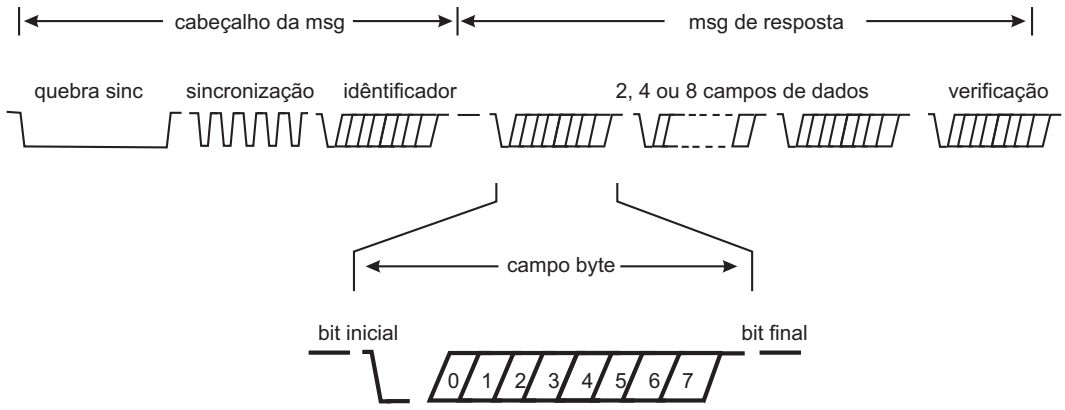

Figura 2.7: Formato de uma mensagem LIN (SPECKS; RAJNáK, 2000)

A mensagem é composta de um cabeçalho que é enviado pela tarefa mestre e por uma resposta que é enviada pela tarefa escravo. O cabeçalho da mensagem é formado por quebra da sincronização (campo quebra sinc), sincronização (campo sincronização), e por um identificador (campo identificador), enquanto que a mensagem de resposta é composta por dois, quatro, ou oito campos de dados e o campo de verificação.

2.1.3 FlexRay

A rede FlexRay (FLEXRAY, 2005) é apontada como a próxima geração de redes em aplicações automotivas, as quais exigem barramentos de dados de alta velocidade que são determinantes para um sistema distribuído de controle. Na Figura 2.8 pode-se ver a evolução dos requisitos para aplicações automotivas. Com o aumento dos dispositivos eletrônicos presentes em um automóvel e o aumento da quantidade de mensagens, o protocolo tem que ser capaz de ter banda suficiente para atender a todos os nodos.

O FlexRay tornou-se o protocolo favorito das montadoras para o desenvolvimento de redes automotivas mais sofisticados, por vários motivos: por possuir a largura da banda 20 vezes maior do que o barramento CAN, por ter um comportamento determinístico e tolerância a falhas (dois canais de comunicação com redundância). Ou seja, por transmitir as mensagens no tempo certo e de forma confiável para as aplicações críticas de segurança, além de ser flexível e de permitir uma maior liberdade de topologias de rede, variando topologias ponto-a-ponto, como também topologias de estrela ativa e de barramento passivo (BRANCO, 2006).

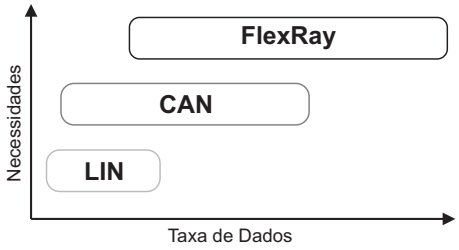

Figura 2.8: Comparação LIN, CAN e FlexRay (INSTRUMENTS, 2006).

Nesse protocolo, a idéia é controlar os dispositivos mecânicos por aplicações do tipo *x-by-wire*, por exemplo, *steer-by-wire*, *brake-by-wire*, etc, permitindo que o mesmo aparelho seja usado para todas as conexões de rede, reduzindo assim os esforços de qualificação e simplificando a administração dos componentes.

O sistema FlexRay é mais do que um protocolo de comunicação, inclui também especificações para transmissões de alta velocidade e definições de interfaces de *hardware* e software, entre os componentes de um nodo FlexRay.

Uma das características do FlexRay são os dois canais de comunicação, o canal A e o canal B, como pode ser visto na Figura 2.9. É importante observar que não se trata de linhas de sinais como CAN_H e CAN_L, pois cada canal tem um par trançado, e cada nó pode ser conectado apenas a um barramento ou aos dois, e, neste último caso, não é permitido que se use o mesmo controlador de comunicação para acessar os dois nodos (PAPAIOANNOU, 2005).

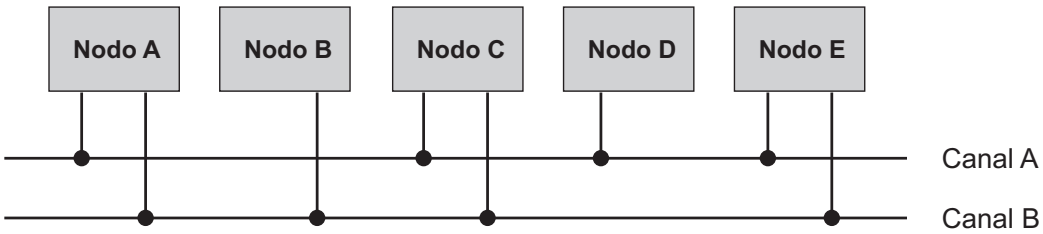

Figura 2.9: Configuração de barramento FlexRay com dois canais (FLEXRAY, 2005).

O desenvolvimento desse novo sistema teve início em 2000, com a participação imediata das empresas DaimlerChrysler, BMW, Philips e Motorola; posteriormente, uniram-se a este grupo também a Volkswagen, a General Motors e a Bosch, e atualmente o consórcio reúne cada vez mais empresas. Ao admitir o maior número possível de empresas, o objetivo das idealizadoras do projeto foi tornar comum o conhecimento da tecnologia, de modo a conseguir a adoção do sistema FlexRay como padrão da indústria, mais rapidamente.

O veículo X5, modelo 2007 da BMW, é o primeiro do mundo a utilizar a tecnologia FlexRay. A rápida transmissão de dados do sistema FlexRay é usada pela primeira vez nas funções rápidas e seguras de coordenação da Condução Adaptável (interação cuidadosamente coordenada das barras antiderrapantes e dos amortecedores feita pela Condução Adaptável). Usando sensores, a Condução Adaptável monitora e calcula permanentemente os dados sobre a velocidade que o veículo tem na estrada, o seu ângulo de condução, a aceleração dianteira e lateral, a aceleração da estrutura e das rodas, assim como os níveis de altura. Em seguida, a partir dessas informações, o sistema faz a padronização tanto dos motores de rotação nas barras antiderrapantes quanto das válvulas eletromagnéticas dos amortecedores, controlando o fluxo da estrutura e a necessidade de amortecimento na medida exata em todos os momentos (DIAS, 2006), (BERWANGER; SCHEDL; TEMPLE, 2006).

2.1.4 Resumo

Nesta seção, apresentaram-se três tipos de barramentos comumente usados em aplicações automotivas: o protocolo CAN, atualmente responsável pelo funcionamento dos sistemas críticos de um automóvel, como os sistemas de segurança ou os sistemas de controle do motor; o LIN, usado onde a largura de banda e a versatilidade do CAN não são necessárias, como vidro elétrico, trava elétrica etc; e por fim, o FlexRay elencado como o futuro da comunicação automotiva.

Deu-se uma visão de como se pode construir um sistema embarcado composto por diferentes protocolos. A Tabela 2.1 ilustra as principais diferenças entre os protocolos.

Tabela 2.1: Comparação entre CAN, LIN e FlexRay.

	CAN	**LIN**	**FlexRay**
Baud rate	1 Mbps	20 Kbps	20 Mbps
Acesso ao barramento	Aleatório e com controle de prioridade	Controlado por escalonamento	Controlado por escalonamento
Endereçamento	Mensagens	Mensagens	Mensagens
Topologia de rede	Linha, estrela	Linha	Linha e/ou estrela
Bytes por *frame*	8	8	254
Comportamento temporal	Determinístico	Determinístico	Determinístico
Custo relativo por conexão	1	0.5	3

2.2 Modelos de Comunicação em Sistemas Distribuídos

Ao projetar mecanismos de interação para sistemas de controle distribuídos, nos quais os componentes distribuídos são considerados como autônomos e concorrentes, é importante e faz-se necessário elencar algumas características (BECKER et al., 2001) que deverão ser suportadas pela infra-estrutura de comunicação subjacente como (COULOURIS; DOLLIMORE; KINDBERG, 2001) (TANENBAUM; STEEN, 2002) (BIRMAN, 1996) (LINCHY, 1996):

- Comunicação muitos-para-muitos particularmente importante para sistemas de controle compostos de sensores inteligentes e atuadores, porque a saída de um sensor pode ser usada por muitas entidades;
- Geração espontânea de mensagens ativadas por um temporizador ou por um evento externo;
- Controle da autonomia;
- Projeto independente e de fácil extensão.

Existem vários modelos de programação para construir aplicações distribuídas, i.e. aplicações compostos por programas que cooperam e rodam em diferentes processos. Os programas necessitam invocar operações de outros processos que podem estar em outra máquina. Para possibilitar isto, alguns modelos tradicionais de programação foram estendidos:

- Programação Procedural: chamada de funções => chamada remota de funções;
- Programação Orientada a Objetos: chamada local de métodos => chamada remota de métodos;
- Programação baseada em Eventos => programação baseada em eventos remotos.

As próximas subseções descrevem possíveis formas de comunicação entre componentes distribuídos.

2.2.1 Send/Receive

O modelo mais utilizado para comunicação entre processos é baseado no modelo cliente-servidor. Através dele são definidos processos que oferecem serviços que podem ser requeridos por processos clientes. Assim, pode-se resumir a comunicação da forma requisição/resposta e, por conseguinte não

ter a necessidade de incluir todas as camadas do protocolo *Open Systems Interconnection* (OSI) e todo o processamento dos cabeçalhos pelas camadas.

Processos cooperantes, componentes de uma aplicação, em inúmeras situações necessitam trocar informações. Com memória compartilhada, os processo compartilham variáveis e trocam informações através do uso de variáveis compartilhadas. Sem memória compartilhada, os processos compartilham informações através de troca de mensagens (*passing messages*). Neste modelo, o Sistema Operacional é responsável pelo mecanismo de comunicação entre os processos.

A comunicação de processos por troca de mensagens permite a troca de informações entre processos que não compartilham memória. A forma mais simples de implementar o modelo-cliente servidor é com a utilização das primitivas *send* e *receive*. Quando um processo que oferece um serviço, chamado de processo servidor, pode receber requisições de serviços este executa a primitiva receive e aguarda (normalmente bloqueado) que uma mensagem chegue para ele. Quando um processo necessita que um serviço seja executado (processo cliente) envia uma requisição usando para isso a primitiva send, colocando como destinatário o processo servidor. Geralmente, o processo cliente usa a primitiva receive (ficando bloqueado) até que o serviço seja executado e uma reposta seja enviada pelo servidor.

Existem várias maneiras de se implementar as primitivas *send* e *receive*:

- Bloqueante ou não-bloqueante;
- Com ou sem bufferização;
- Confiável ou não-confiável.

O grande problema do uso direto na programação de aplicações baseadas no modelo cliente-servidor é que ele é pouco transparente impondo dificuldades para os programadores. Considerando que é amplamente utilizado o conceito de modularização de programas surgiu uma forma mais próxima da programação utilizada atualmente que seria a chamada de procedimento remoto *Remote Procedure Call* (RPC).

2.2.2 RPC

O RPC estende o conceito tradicional de chamada de procedimentos (BIRRELL; NELSON, 1984). Ele permite que chamadas a procedimentos ou funções sejam feitas via rede, isto é, permite chamar

um procedimento de um programa que está sendo executado em outro computador. Dessa maneira, transforma a chamada de procedimento para outra máquina, de forma transparente ao processo invocador. Com a transparência de acesso, toda a complexidade relacionada às tarefas de serialização/deserialização de parâmetros também são ocultados do programador e encapsulados dentro dos *stubs*.

Os *stubs* são procedimentos que contêm o código de chamadas de rede. Com eles o RPC protege as aplicação (cliente e servidor) de preocupações com detalhes baixo nível, por exemplo, *sockets*. Visando à interoperabilidade, há um formato padrão dos dados, e é nos *stubs* que acontece a conversão dos dados. Os *stubs* são gerados automaticamente durante a compilação dos programas que compõem o sistema por algum compilador.

Resumindo, uma chamada de procedimento remoto pode ser assim definida:

- A aplicação cliente faz a chamada ao procedimento remoto, como se fosse um procedimento local;
- O *stub* do cliente obtém os dados dessa chamada (identificação e respectivos argumentos) e faz o empacotamento desses dados;
- O módulo responsável pela comunicação é acionado de forma a fazer o envio de uma mensagem contendo a chamada de procedimento para o servidor apropriado;
- Do outro lado, o módulo de comunicação do servidor recebe a mensagem vinda do cliente e a repassa para o *stub* do servidor (também chamado de *skeleton*);
- O *stub* do servidor desempacota a mensagem, executando uma chamada de procedimento ao servidor, e torna possível a invocação do procedimento adequado no servidor;
- O procedimento desejado é executado;
- O *stub* do servidor empacota os resultados da execução do procedimento desejado (código de retorno e argumentos de saída) e encaminha para o módulo de comunicação;
- A mensagem é então enviada de volta para o cliente;
- A mensagem é recebida e repassada ao *stub* do cliente;

- O *stub* do cliente recebe a mensagem de retorno, desempacota a resposta e a retorna para a aplicação cliente.

Do lado do processo cliente, todas as etapas listadas anteriormente ocorrem de forma transparente. A aplicação cliente simplesmente faz uma chamada de procedimento e fica esperando o retorno.

Esta tecnologia passou a apresentar dois problemas (DEITEL; DEITEL, 2002). O primeiro está relacionado ao tipo de dados: com o advento das linguagens orientadas a objetos surgiu a necessidade de trabalhar com tipos de dados mais complexos (objetos, com atributos e métodos). Entretanto, o RPC foi projetado para trabalhar apenas com determinadas estruturas (compostas apenas por um conjunto de dados simples). O segundo problema está relacionado ao fato de que o programador deve compreender a Linguagem de Definição de Interface (Interface Definition Language - IDL), que é utilizada para descrever as possíveis funções a serem invocadas remotamente e gerar automaticamente os *stubs* do cliente e do servidor. Neste contexto, surge a tecnologia de Invocação de Método Remoto (*Remote Method Invocation* - RMI), com a finalidade de resolver os problemas anteriormente citados.

2.2.3 RMI

O RMI é uma tecnologia de programação de objetos distribuídos comumente utilizada com linguagens orientadas a objetos como C++ e Java. O RMI possibilita a comunicação entre objetos que rodam em máquinas virtuais diferentes, independentemente dessas máquinas virtuais estarem no mesmo hardware ou não. Neste modelo de objetos distribuídos, cada processo contém uma coleção de objetos. Alguns desses objetos podem ter seus métodos chamados local ou remotamente:

- Chamada Remota é denominada RMI;
- Objetos Remotos são aqueles cujos métodos podem ser invocados a distância.

Interface

Linguagens de programação modernas organizam um programa como um conjunto de módulos que comunicam uns com os outros. A comunicação é por chamada de procedimento ou acesso direto a variáveis. Para controlar todas as possíveis interações, uma interface é definida para cada módulo. A interface de um módulo (um módulo denota uma procedure ou um método de um objeto) permite esconder os detalhes da implementação, tornando visível somente o que interessa ao invocador. A

implementação do módulo pode ser modificada sem implicar mudanças nos invocadores (COULOURIS; DOLLIMORE; KINDBERG, 2001).

Para que um objeto A invoque um método de um objeto B é preciso:

- Que A tenha uma referência ao objeto B;
- Que A saiba quais métodos de B podem ser chamados e como deve chamá-los (argumentos).

Referência a objetos remotos

A referência remota é um identificador único para o objeto remoto dentro do sistema distribuído. Sua representação geralmente difere de referências de objetos locais, porém referências de objetos remotos são análogas as locais nisso:

- O objeto remoto para receber uma invocação de método remoto é especificado como uma referência de objeto remota;
- Podem ser passadas referências de objeto remotas como argumentos e resultados de invocação de método remotos.

Interface Remota

Todo objeto remoto possui uma interface remota que especifica quais dos seus métodos podem ser invocados remotamente. Interfaces remotas são classes interface, ou seja, não podem ser instanciadas, somente realizadas.

As Linguagens de Definição de Interface permitem que objetos escritos em diferentes linguagens invoquem os métodos uns dos outros. A semântica de passagem de parâmetros deve ser ajustada para se adequar ao esquema de invocação remota. Em particular, os parâmetros devem ser definidos como sendo de entrada ou saída e ponteiros não são válidos em processos diferentes, portanto ponteiros não são passados como argumentos ou retornados (COULOURIS; DOLLIMORE; KINDBERG, 2001).

Limites de uma Interface Remota

Os modelos de invocação remota de módulos são muito parecidos com os de invocação local, porém há restrições. A interface de um módulo (procedure ou objeto) especifica as operações e as variáveis que podem ser acessadas por outros módulos. Em sistemas distribuídos módulos de um

mesmo sistema podem rodar em máquinas diferentes, logo, não é possível para um módulo acessar diretamente as variáveis de outro módulo. Outra diferença é que não é possível fazer passagem de argumentos por referência. Na programação local, quando passamos um argumento por referência (i.e. passamos seu endereço de memória), estamos alterando-o na origem, ou seja, não trabalhamos sobre uma cópia do argumento e sim, sobre o argumento original. Ponteiros não podem ser utilizados porque não fazem referência a uma área da memória local, logo perdem o sentido em um ambiente distribuído.

Semântica de uma Invocação Remota

Invocações locais são executadas exatamente uma vez, este pode não ser o caso para invocações remotas. Basicamente, há três opções de execução do método em caso de falha na sua invocação:

- Talvez = zero ou uma: não há tratamento de falhas, uma invocação pode ser executada com sucesso na primeira solicitação ou com insucesso. O problema é que no caso de insucesso, o programador não sabe:
  - Se o método não foi executado: a solicitação não chegou ao servidor; ou
  - Se o método foi executado: a resposta não chegou ao cliente, mas o servidor executou o método.
- Pelo menos uma vez;
  - O cliente recebe um resultado (e o método foi executado pelo menos uma vez) ou uma exceção (sem resultado);
  - Falhas arbitrárias: se a mensagem de invocação é retransmitida, o objeto remoto deve executar o método ao menos uma vez, possivelmente causando valores falhos a serem armazenados ou retornados;
  - se operações idempotentes são usadas, arbitrariamente falhas não irão ocorrer.
- No máximo uma vez.
  - O cliente recebe um resultado (e o método foi executado no máximo uma vez); ou
  - Uma exceção (ao invés de um resultado, no caso, o método foi executado uma vez ou não foi).

Implementação de RMI

- Módulo de Comunicação: implementa protocolo *REQUEST-REPLY* sobre a pilha de protocolos. Utiliza os três primeiros campos da mensagem *request-reply*, notadamente a referência ao objeto remoto para destinar a mensagem ao *dispatcher* correto. É o módulo de comunicação que implementa a semântica de invocação do protocolo *request-reply*. Existe um *dispatcher* por classe de objeto remoto.

- Módulo de Referência: traduz as referências remotas à objetos que chegam nas mensagens de *request* em referências locais, para isto utiliza uma tabela Referência Remota x Referência Local. Cria referências à objetos remotos: uma referência remota é criada pelo módulo de referência nas seguintes situações:

  – Quando um objeto remoto é passado como argumento ou resultado de um método pela primeira vez, o módulo de referência cria uma referência remota e a coloca na tabela (associada à referência local).

  – Quando uma referência a um objeto remoto vem num *request* ou numa resposta, o módulo de comunicação pergunta ao de referência pela referência local do objeto (dada a referência remota). A referência local pode identificar um objeto local ou um *proxy*. No caso onde uma referência remota não é encontrada na tabela, o software RMI cria um novo *proxy* e solicita ao módulo de referência para incluir esta nova referência.

- *Proxy*: é o representante do objeto remoto. Torna a invocação do método transparente. O *proxy* realiza a interface remota transmitindo *request* para o objeto servidor e recebendo a resposta. Funciona como um objeto local para o invocador. Encapsula os detalhes da referência ao objeto remoto, do *marshalling* dos argumentos do método invocado e do *unmarshalling* da resposta. Há um *proxy* para cada objeto remoto para o qual um processo tem uma referência. No RMI objetos remotos podem ser passados como argumentos no momento da chamada de um método remoto (na verdade passa-se o *proxy* do objeto).

- *Marshalling*: transforma a mensagem num formato neutro e plano (em relação às estruturas de dados), isto é, independente da plataforma (sistema operacional e *hardware*) e das estruturas de dados e tipos de dados da linguagem de programação de origem.

- *Unmarshalling*: transforma a mensagem do formato neutro para o formato das estruturas de dados que foram “achatadas”;

- *Dispatcher* e *Skeleton*: cada objeto remoto possui uma dupla *Dispatcher* e *Skeleton*. O *Dispatcher* recebe o *request* do módulo de comunicação e invoca o método no *Skeleton* usando o *methodId* que vem na mensagem. O *Skeleton* realiza efetivamente a invocação (chamada). Ele faz o *unmarshall* do *request*, invoca o método e pega o resultado. Em seguida, faz o *marshall* do resultado e monta uma mensagem de *reply* passando ao módulo de comunicação.

- Formato da mensagem *request-reply*:

  – Request ID: identificador da mensagem. Quando um cliente invoca um método de um objeto remoto um *requestID* é gerado. O servidor coloca este mesmo *requestID* é colocado na mensagem de resposta. Este número permite checar se uma resposta é do último *request* ou se é de um *request* antigo cuja resposta chegou atrasada.

  – Id do método: pode ser um número seqüencial que identifica cada método da interface. Se o cliente e o servidor utilizam uma mesma linguagem que suporta reflexão (ex. Java RMI) então uma representação do método pode ser colocada.

  – Referência ao objeto remoto: mesmo depois que um objeto remoto é suprimido, é interessante não utilizar sua referência, pois outros objetos podem guardar a referência obsoleta. Qualquer tentativa de invocar um objeto inexistente produz um erro, o que é melhor que permitir acesso a um objeto diferente.

  – Interface: algumas implementações de RMI contêm informação sobre a interface do objeto remoto, por exemplo, o nome da mesma. Esta informação é relevante para outro processo que recebe a referência como argumento ou resultado de uma invocação remota para saber sobre os métodos oferecidos pelo objeto remoto. Para permitir realocação (migração) de um objeto de uma máquina a outra, a referência do mesmo não deve ser composta pelo endereço IP+porta.

2.2.4 Publisher/Subscriber

O padrão de comunicação Publiher/Subscriber oferece facilidade para manter a cooperação entre os componentes envolvidos no sistema. Esse padrão é também conhecido como *Observer* (GAMMA et

al., 1995).

Em alguns sistemas, os dados mudam em certo lugar, e vários objetos em outras partes do sistema dependem daqueles dados; é necessário então utilizar algum mecanismo para informar os dependentes das mudanças. Pode-se resolver esse problema com chamadas explícitas do objeto cujo estado mudou para os seus dependentes, mas esta solução não é flexível nem reutilizável. De uma forma mais genérica, é preciso um mecanismo de propagação de mudanças que possa ser aplicado em vários contextos. A solução tem que equilibrar as seguintes forças:

- Um ou mais objetos devem ser notificados sobre mudanças em outro objeto;
- O número e a identidade de dependentes não são conhecidos *a priori* e podem até mudar dinamicamente;
- Fazer com que os dependentes peguem o estado, de tempos em tempos (*polling*) é muito caro;
- O objeto que provê o dado e aqueles que o consomem não devem ser fortemente acoplados.

A solução proposta em (SCHMIDT et al., 2001) diz que o objeto que contém o dado que interessa é chamado de *Publisher* (*subject* em (GAMMA et al., 1995)). Todos os objetos dependentes são chamados de *Subscribers* (*observers* no (GAMMA et al., 1995)). O *Publisher* mantém uma lista dos objetos registrados que são aqueles interessados em receber notificações sobre as mudanças e ainda oferece uma interface para manifestação de interesse (interface *subscribe*). Quando o estado do *Publisher* muda, há o envio de uma notificação para todos os objetos que manifestaram interesse. Ao receber a notificação, o *Subscriber* decide se solicita a informação ao *Publisher* ou não.

Há os seguintes graus de liberdade para se implementar o padrão:

- Pode-se utilizar classes abstratas para implementar o comportamento básico do padrão;
- O *Publisher* pode decidir quais partes do seu estado interno serão notificadas;
- O *Publisher* pode decidir enfileirar as mudanças de estado antes de enviar as notificações;
- Um objeto pode se inscrever em vários *Publishers*;
- Um objeto pode ser simultaneamente *Publisher* e *Subscriber*;
- Inscrição e notificação podem ser filtradas de acordo com o tipo de evento gerado, ou tipo de dado que foi alterado;

- O *Publisher* pode enviar, junto com a notificação, informações sobre o que mudou ou não.

Em termos gerais, há diferenças entre o modelo *push* e o modelo *pull*. No modelo *push*, o *Publisher* envia todos os dados para os *Subscribers* quando alguma alteração acontece. Os *Subscribers* não têm escolha sobre o recebimento dos dados e nem quando os receberão. No modelo *pull*, o *Publisher* somente envia o conteúdo mínimo de informação para os *Subscribers* quando alguma alteração acontece; os Subscribers são responsáveis pelo recebimento dos dados necessários.

Muitas variações são possíveis entre esses dois modelos. O modelo *push*, por exemplo, tem um comportamento mais rígido, enquanto que o modelo *pull* oferece mais flexibilidade ao custo de um número maior de mensagens.

Para mudanças de dados complexas, o modelo *push* pode não ser a melhor escolha, especialmente quando o *Publisher* envia grande quantidade de pacotes para o *Subscriber* e este pode não estar interessado. O envio de grande quantidade de dados pode ocasionar *overhead*. Nesses casos, usa-se o modelo *pull* e faz-se os *Subscribers* descobrirem que tipo de mudança aconteceu.

Logo percebe-se que há dois extremos:

- No modelo *push* há pouca flexibilidade, mas possivelmente melhor uso da banda;
- No modelo *pull* há mais flexibilidade, mas o número de mensagens é maior.

No padrão *Publisher/Subscriber* pode haver algumas variações como *Gatekeeper*, *Event Channel* e *Producer/Consumer*.

Esse padrão pode também ser aplicado em sistemas distribuídos, passando a ser chamado de *Gatekeeper*. Nessa variante a instância *Publisher* notifica *Subscribers* remotos. Pode-se dividir o *Publisher* em dois processos: um processo gera os eventos e o outro faz a demultiplexação definindo quais *Subscribers* os receberão e enviando as mensagens usando o padrão *Reactor* do POSA II (KON, 2007).

Outro variante do modelo é o *Event Channel*, proposto pela OMG no CORBA *Event Service* (OMG, 2004a). Nesse tipo de modelo os *Publishers* e os *Subscribers* estão fortemente desacoplados e ligados por um canal (*channel*). Neste caso, os consumidores não precisam saber a origem dos eventos nem quem é o *Publisher*; o interessante é o que foi publicado. Enquanto para os *Publishers*, não interessa quem são os *Subscribers*.

Neste caso, um ou mais canais são introduzidos no meio de um ou mais *Publishers* e de um ou mais *Subscribers*. Para os *Publishers*, o *Event Channel* se parece com um *Subscriber*, enquanto para os *Subscribers*, o *Event Channel* se parece com um *Publisher*.

Publishers, *Channels* e *Subscribers* podem residir em máquinas distintas e, além disso, podem utilizar vários canais. Esses canais podem oferecer serviços adicionais: "bufferização", filtragem e priorização. Esse tipo de variante fornece múltiplos *Publishers* e vários tipos de eventos.

Outra variação do padrão *Publisher/Subscriber* é o uso do estilo de cooperação *Producer/Consumer*. Aqui, duas *threads*, em um processo ou dois processos em uma máquina que compartilham um *buffer*, interagem; um deles produz dados, enquanto o outro os retira do *buffer* e os consome. Nesse caso não há comunicação direta entre eles, mas há controle do acesso concorrente ao *buffer* compartilhado. Há uma relação de 1:1 que não existe nas outras variantes do padrão (KON, 2007).

A arquitetura Jini (SUN, 1999) (SUN, 2001) (EDWARDS, 2000) é um exemplo de computação distribuída que fornece através de uma API a construção de aplicações distribuídas baseadas na idéia de eventos. O mesmo mecanismo de eventos usado na construção de JavaBeans e presente nas Java Foundation Classes é estendido para o ambiente distribuído. Eventos podem ser usados, por exemplo, para permitir que um Servidor de *Lookup* notifique um dado cliente quando o servidor que aquele cliente estava esperando finalmente executou a operação "*join*" e ingressou na federação (OLIVIERA; FRAGA; MONTEZ, 2002).

Outras referências sobre o padrão *Publisher/Subscriber* podem ser vistas em (SCHMIDT et al., 2001) e (GAMMA et al., 1995) no qual o modelo é chamado de Observer, assim como em (EUGSTER et al., 2003).

2.2.5 Considerações Adicionais

Discutiram-se os modelos de comunicação que melhor se adaptam para aplicações em sistemas embarcados distribuídos. Com o modelo RMI é possível ativar diretamente um método de outro objeto, tanto local quanto remotamente. No modelo *Publisher/Subscriber*, os objetos publicam mensagens com eventos que podem ser assinados por outros objetos.

O modelo *Publisher/Subscriber* mostrou-se mais adequado à proposta aqui defendida, i.e., de comunicação anônima e assíncrona, havendo a ligação entre os objetos em tempo de execução. O desenvolvedor identifica e decide quais eventos devem ser publicados e de quais eventos os assinantes

precisam. Sendo assim, quando mais de um componente deseja receber o mesmo evento é necessário somente modificar os componentes assinantes; o que não acontece quando se usa RMI.

2.3 *Middleware* para Sistemas Embarcados

Sistemas embarcados apresentam, em sua estrutura, uma multiplicidade de dispositivos e uma variedade tecnológica, tanto em *hardware* quanto em software, além de estarem conectados com diferentes tecnologias e protocolos de redes que resultam em uma arquitetura heterogênea e distribuída. Particularmente, os sistemas embarcados robóticos são constituídos de múltiplos dispositivos de diferentes naturezas. Invariavelmente, essas aplicações possuem três conjuntos de elementos básicos: sensores (de deslocamento, posicionamento etc.), para a percepção de seu ambiente de execução; atuadores, representados tipicamente por motores; e uma ou mais unidades de processamentos das informações do sistema (do mais simples microprocessador até os mais sofisticados microcomputadores e microcontroladores).

Para auxiliar no enfrentamento dessa complexidade inerente aos sistemas embarcados, pode-se utilizar o conceito de *middleware*. Um *middleware* consiste basicamente em uma camada de integração de software, localizada acima do sistema operacional e do substrato de comunicação, o qual oferece abstrações de alto nível e um conjunto de serviços, tendo como objetivo facilitar a programação distribuída.

Segundo a literatura especializada, um dos principais argumentos contra a utilização de *middleware* em aplicações embarcadas deve-se ao acréscimo de código no sistema, prejudicando seu desempenho. O aumento de código no sistema deve-se a *middlewares* de uso geral, porém quando o *middleware* assegura somente as características necessárias para a aplicação alvo, oferece muitos benefícios sem prejudicar o desempenho do sistema.

O *middleware* COoperating SMart devICes (COSMIC) (KAISER; BRUDNA; MITIDIERI, 2005) é um exemplo de *middleware* para sistemas embarcados que melhora e agiliza o processo de desenvolvimento de software sem prejudicar o desempenho do sistema. O COSMIC possibilita ao desenvolvedor abstrair aspectos de comunicação focando seu trabalho nos conceitos da aplicação, deixando de lado aspectos de baixo nível.

Basicamente, a implementação do *middleware* é composta por uma parte CAN-Bus e uma parte TCP/IP. Para permitir o uso de ambas as redes, TCP/IP e CAN-Bus, o conceito de um *gateway* é

introduzido na arquitetura de comunicação. O papel do *gateway* no middleware é garantir que toda aplicação que funcione em cada nodo receba as mensagens publicadas em toda a rede; a Figura 2.10 ilustra as partes do *middleware*. Então, do ponto de vista da aplicação, a infra-estrutura de comunicação fornece os mesmos serviços para ambos os objetos, rodando sobre nodos TCP/IP e CAN. Há um *Event Channel Handler* (ECH) em cada nodo que fornece todo o suporte para os objetos locais e um *Event Channel Broker* (ECB) rodando sobre um nodo TCP/IP que faz a comunicação para os nodos TCP/IP. Há também um ECB para a rede CAN, que é implementado no *gateway*. Então, o gateway é um elemento especial que pode ser visto como um ECH e um ECB para a rede CAN e como um ECH para rede TCP/IP.

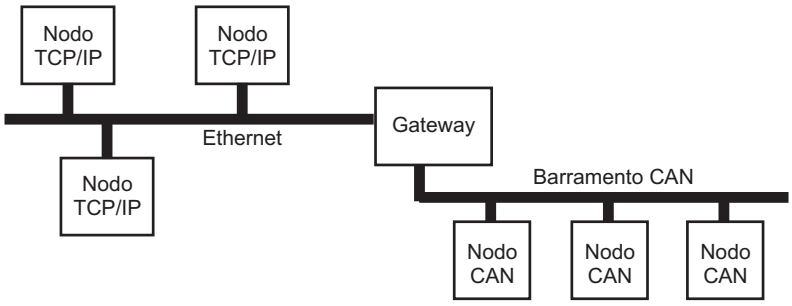

Figura 2.10: Visão do *middleware* COSMIC

Na parte CAN-Bus, para suportar o modelo de comunicação *Publisher/Subscriber* e para explorar esquema de prioridade dinâmica, o identificador de 29 *bits* CAN-Bus (CAN 2.0 B) de um evento é dividido em três campos. Um dos campos identifica a qual tipo de canal o evento pertence. Um segundo campo, o qual contém a identificação de um nodo, é adicionado para assegurar um identificador CAN único. Além destes, há mais um campo que fornece prioridade explícita que pode ser usada para escalonar mensagens. Este esquema faz uso do *broadcast* natural do protocolo CAN, evitando assim métodos tradicionais de endereçamento direto, e , além disso, permite a configuração dinâmica dos nodos CAN e dos identificadores de canais.

Sempre que um novo nodo CAN é inicializado, deve ser feita a troca do identificador (de 64 *bits*) por um menor. Isto é feito pelo ECH em uma configuração específica no qual o identificador é enviado em oito fragmentos. Depois de receber o último fragmento, o CAN-ECB fornece um identificador com 14 *bits* (chamado de TxNode) para o nodo CAN em uma mensagem dedicada. Esta etapa da configuração não produz nenhum efeito no lado da rede TCP/IP.

Uma vez feita a troca do identificador, o nodo CAN começa outras operações de comunicação,

tais como *Subscriber* ou *Publisher*, para um canal. Para fazer *Subscriber* no canal, um objeto usa o identificador de 64 *bits* do canal como parâmetro. Então, o ECH faz *update* de sua lista interna de assinantes de modo que a aplicação seja notificada sempre que um novo evento for publicado no canal. O que acontece em seguida depende somente se o ECH tem uma *tag event* para o canal; se não tiver será necessário primeiro pedir uma *tag event* ao CAN-ECB usando uma mensagem especial, em que o identificador de 64 *bits* é usado, identificando o canal. Então, um identificador de 14 *bits* é emitido pelo ECB e pode ser usado pelo ECH para reconhecer toda mensagem transmitida pela rede CAN-Bus relacionada a determinado canal.

Para publicar um evento em um determinado canal, a aplicação terá que emitir um pedido ao ECH local fornecendo os 64 *bits* do identificador do canal relacionado ao evento. Então, o ECH deverá checar se algum outro objeto local fez *Subscriber* para o mesmo canal e, neste caso, emitir uma cópia do evento ao objeto. Além disso, o ECH fará *broadcast* na rede CAN-Bus. Se uma *etag* ainda não estiver disponível, o ECH deverá primeiro pedir um identificador de 14 *bits* para o ECB-CAN, e somente depois de recebê-la a etag é que será feito o *broadcast* do evento. Finalmente, outros nodos CAN receberão a mensagem e poderão fazer assinaturas para o determinado canal.

Da mesma maneira que na parte CAN-Bus, a comunicação na parte TCP/IP é realizada por um ECH local e pelo TCP/IP-ECB. Quando a comunicação não usa o protocolo CAN, isto é, só usa rede TCP/IP, não há a necessidade de configuração dos identificadores de nodos. Os nodos são simplesmente identificados pelo seu número IP e pelo identificador de canal (64 *bits*).

Para fazer uma operação de assinatura em determinado canal, uma aplicação fornece o identificador do canal desejado e o ECH local adiciona a aplicação na lista de aplicações interessadas naquele canal. Depois disso, o ECH envia uma mensagem para o TCP/IP-ECB requisitando a assinatura no canal. Então, o TCP/IP-ECB adiciona o número IP do nodo na lista geral de nodos interessados naquele canal. Esta lista, por canal, contém todos os publicadores e assinantes, permitindo que o TCP/IP-ECB envie *updates* da lista de assinantes para todos os IP's publicadores da lista. Quando uma aplicação faz o pedido de operação *Publish*, o ECH local verifica se qualquer outra aplicação local esta interessada nesta informação e envia uma cópia para ela. Após isso, o ECH procura tal informação em sua lista de assinantes do canal e envia uma cópia do evento para cada um da lista.

Da parte TCP/IP do sistema, o CAN-Bus é visto como um nodo (*gateway*) identificado por seu número IP. Assim, quando um nodo CAN publica um evento, o *gateway* faz a ligação, usando a

informação fornecida pelo ECB, entre o publicador do evento e os nodos assinantes TCP/IP. Da mesma maneira, quando o *gateway* recebe um evento da rede TCP/IP, ele tem que publicá-lo na rede CAN-Bus de modo que os nodos CAN o recebam.

Outra tarefa do *gateway* é realizar a configuração dos nodos CAN. Então, sempre que um nodo CAN envia uma mensagem de requisição "*short* ID", o *gateway* terá que receber cada um dos oito fragmentos do identificador único do nodo ID e computar seu valor original. Depois disso, o *gateway* fornecerá um TXNodo ao nodo CAN. Quando o *gateway* recebe uma mensagem "*request etag*" ele fornece uma *etag* e salva o identificador original (64 *bits*) junto com a *etag* em uma lista local do *gateway*.

Sempre que um objeto rodando em nodo CAN faz um pedido de assinatura para um determinado canal ao ECH local, ele também envia uma mensagem para o *gateway*. O *gateway* envia uma mensagem para o TCP/IP-ECB pedindo assinatura naquele canal. Como resposta, o TCP/IP-ECB envia a lista atual dos assinantes do respectivo canal. Esta lista pode também ser requisitada ao ECB durante a primeira operação, quando vai ser publicado um evento. Então, todas as próximas mudanças na lista serão emitidas automaticamente pelo ECB.

O middleware COSMIC quando comparado com outros middlewares, mostrou-se mais adequado a proposta aqui defendida.

(HARRISON; LEVINE; SCHMIDT, 1997) introduziu um sistema de eventos tempo real para CORBA. Os eventos são roteados através de um servidor de eventos central que provê suporte para requerimentos de tempo real. Ou seja, um componente central é encarregado de garantir os requerimentos de tempo real.

Esse componente central não está incluído no middleware COSMIC, ou seja, no COSMIC não há sobrecarga ou vulnerabilidade de um componente especifico.

Para explorar a camada subjacente do sistema CORBA, a disseminação e a notificação de eventos são baseadas em mecanismos de invocação remota do CORBA. Os problemas discutidos anteriormente na seção Modelos de Comunicação em Sistemas Distribuídos e em (BECKER et al., 2001), mostra que esse tipo de comunicação acaba introduzindo problemas de controle de dependência.

Outro problema encontrado é a complexidade desse sistema, ou seja, inadequado para sistemas embarcados com pouca memória.

Já (KIM et al., 2000) e (LANKES; JABS; BEMMERL, 2003) implementam o CAN-Bus para CORBA.

Entretanto, não há garantia de confiabilidade como também não há garantia de atendimento dos requisitos de tempo real.

2.4 Considerações Finais

Apresentaram-se, neste capítulo, possíveis formas de comunicação entre componentes distribuídos, como por exemplo Send/Receive, RPC, RMI e *Publisher/Subscriber*. O modelo *Publisher/Subscriber* mostrou-se capaz de atender aos objetivos da proposta aqui defendida, ou seja, comunicação tipicamente anônima, inerentemente assíncrona e *multicast* por natureza e, desse modo, selecionou-se como modelo de comunicação. Além disso, foram apresentados alguns barramentos utilizados em aplicações automotivas. O protocolo CAN foi escolhido por ser atualmente responsável pelo funcionamento dos sistemas críticos de um automóvel.

Por abstrair aspectos de comunicação baixo nível, por oferecer um conjunto de serviços que facilita a programação distribuída, por oferecer suporte ao modelo *Publisher/Subscriber* e por também dar suporte a diferentes protocolos de rede como CAN e TCP/IP, selecionou-se o *middleware* COSMIC como base da arquitetura proposta neste trabalho.

Capítulo 3

Descrição da Arquitetura Proposta

3.1 Introdução

Esta seção apresenta a arquitetura de controle originalmente usada no veículo Mini-baja, utilizado como objeto de estudo nesta dissertação. O veículo Mini-baja foi automatizado e convertido em um veículo *drive-by-wire*. Este é movido através de tração das rodas traseiras por um motor à combustão de 5HP. Em sua condição original pode ser guiado como um carro convencional e possui um deslocamento máximo das rodas de aproximadamente 0,79rad. A arquitetura de controle original foi organizada em diferentes subsistemas como tração, aceleração, freios e controle do ângulo de direção, conforme ilustrado na Figura 3.1.

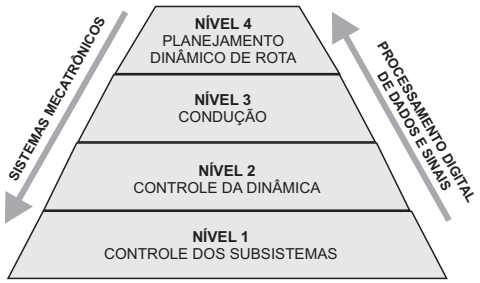

Figura 3.1: Hierarquia dos sistemas de controle de um veículo autônomo (JUNG et al., 2005)

Os sistemas do primeiro nível realizam as tarefas fundamentais do veículo, como frenagem, aceleração e guiamento, tendo como exemplo os sistemas de injeção eletrônica. No segundo nível, encontram-se os sistemas que atuam sobre o comportamento dinâmico do veículo, como sistemas *Anti-lock Braking System* (ABS) e *Electronic Stability Program* (ESP). No terceiro nível, estão presen-

tes os controladores responsáveis por guiar o veículo de acordo com uma trajetória pré-determinada. Esta trajetória é definida no quarto e mais alto nível, onde estão os sistemas que realizam o planejamento de trajetória e, caso necessário, a sua adaptação de forma dinâmica ao longo do trajeto, como, por exemplo, o contorno de obstáculos.

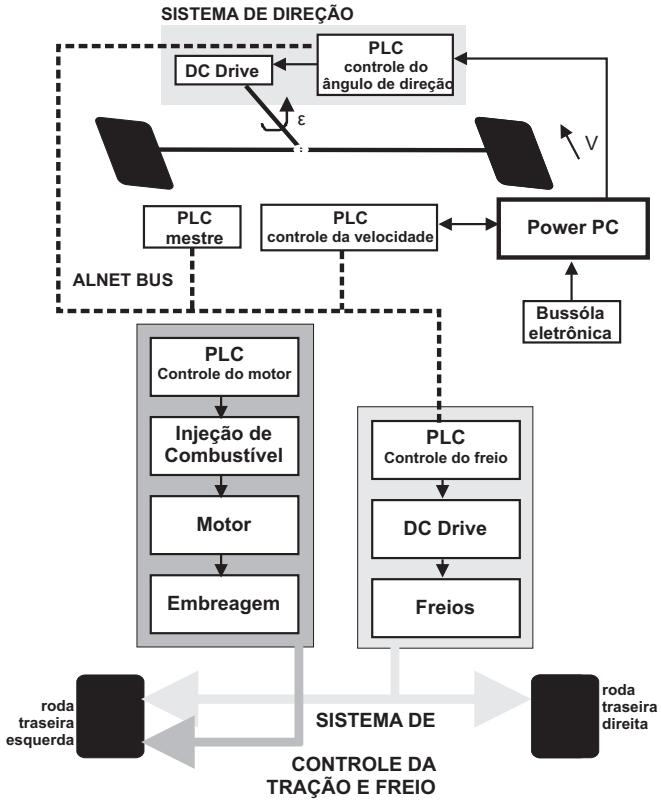

Figura 3.2: Arquitetura do sistema de controle original

A arquitetura de controle original implementada no veículo foi baseada na rede PLC ALTUS, com os PLCs interconectados pelo barramento ALNET1, conforme mostrado na Figura 3.2. Há um PLC responsável por cada subsistema, tal como em uma estrutura de controle distribuída. Outros dois PLCs são usados para controle de velocidade e para controle da rede. A direção, tração e o sistema de freio foram analisados separadamente e para cada um deles foi desenvolvido um controlador específico. Para controle de velocidade um *Adaptive Cruise Control* (ACC) foi implementado. O sistema ACC foi separado em duas partes, uma parte para aceleração e a outra parte para frear o veículo. Cada parte apresenta algumas particularidades, que são integradas em um modelo matemático. O controle de velocidade trata a aceleração, desaceleração e a frenagem do veículo.

Esta arquitetura foi remodelada no escopo deste trabalho conforme discutido na próxima seção.

3.2 Visão Geral da Arquitetura

A nova arquitetura apresentada neste trabalho substitui os PLCs por nodos computacionais de baixo custo. O barramento CAN é usado para interconectar estes nodos, em vez do protocolo proprietário do fabricante de PLC (ALNET). A nova arquitetura proposta é mostrada na Figura 3.3.

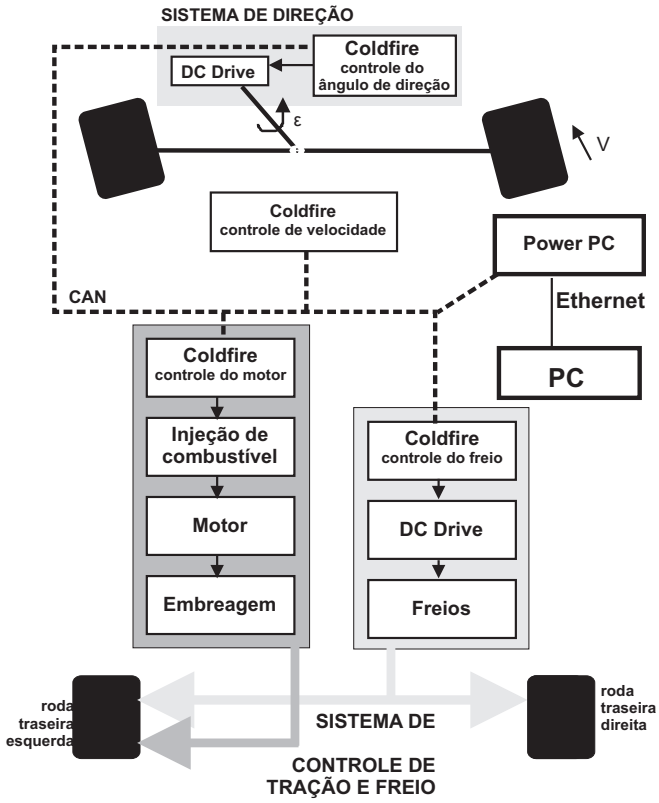

Figura 3.3: Arquitetura do sistema de controle modificada

Uma das primeiras modificações introduzidas na arquitetura original foi trocar os PLCs por nodos micro-processados de baixo custo, baseados na arquitetura Coldfire. Adicionalmente, o PLC mestre foi substituído por um nodo PowerPC com maior capacidade computacional. Além disso, um PC foi usado aqui para emular a situação onde veículos poderiam comunicar entre si usando uma rede sem fio. Para simplificar o problema utilizou-se uma conexão ethernet que conecta o PC ao nodo PowerPC.

A arquitetura proposta neste trabalho denomina-se DOCAS. Ela é baseada no modelo de comunicação *Publisher/Subscriber*, permitindo assim a interação entre componentes distribuídos. Outro componente importante da arquitetura é a parte de software, cuja estrutura geral é mostrada na Figura 3.4. Essa arquitetura oferece suporte à execução de objetos autônomos presentes em diferentes nodos

computacionais, de modo que esses objetos possam interagir a fim de realizar determinada tarefa. Com isto, a arquitetura vem a ser útil em vários tipos de aplicações para sistemas embarcados.

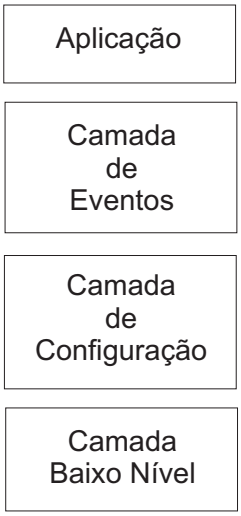

Figura 3.4: Camadas da arquitetura DOCAS

A primeira camada da Figura 3.4 engloba os componentes da aplicação, ou seja, todos os objetos que interagem de alguma forma com o sistema. Os componentes da aplicação são os objetos que publicam e recebem eventos dos canais de eventos.

A segunda camada, denominada Camada de Eventos, vem logo abaixo da camada aplicação, dando todo suporte aos componentes, ou seja, o próprio canal de eventos e o *event channel handler* presente em cada nodo da rede. Além disso, essa camada é responsável por enviar os eventos para a camada subjacente e por receber da camada inferior notificações do protocolo da rede.

A terceira camada, denominada Camada de Configuração, é responsável por implementar os mecanismos de transporte e configuração da rede, estruturando as mensagens, enviando para os respectivos nodos, além de receber mensagens. Essa camada é responsável pela configuração do protocolo, por exemplo, pela troca do identificador de um nodo CAN 64 *bits* por um identificador menor, de 7 *bits*, conforme é detalhado no Capítulo 4.

A quarta camada, denominada Camada Baixo Nível, é responsável pelas configurações de mais baixo nível, ou seja, ela dá suporte à rede CAN, na priorização das mensagens, detecção de erros e interrupções; além disso, trata os *bufferes* de saída e chegada das mensagens CAN.

3.3 Modelagem de Aplicações na Arquitetura DOCAS

Este trabalho preocupa-se também com a modelagem do sistema em alto nível antes da implementação propriamente dita. Para tanto, utilizou-se a linguagem UML 2.0 (OMG, 2004b) capaz de expressar os requisitos de uma aplicação dessa natureza.

A UML 2.0 (OMG, 2004b) não está vinculada a nenhum *framework*; além disso, ela contém alguns aspectos como geração automática de código, customização, padronização dos modelos e precisão semântica cujo potencial e adequação despertam o interesse pelo fato de ser uma linguagem de modelagem padronizada e extensamente usada. A UML 2.0 acompanha a evolução do desenvolvimento de software, dando suporte ao Desenvolvimento Baseado em Componentes (DBC) e ao desenvolvimento baseado em modelos *Model Driven Development* (MDD), além do enfoque em arquitetura de software.

A investigação sobre o uso da UML para modelar uma aplicação distribuída divide o problema em duas partes: (i) descrições da estrutura da aplicação; e (ii) descrição do seu comportamento. A modelagem estrutural consiste em classes e seus relacionamentos, componentes e nodos do sistema, com as respectivas instâncias. A modelagem comportamental descreve como cada componente reage aos estímulos recebidos.

Do ponto de vista da modelagem dos elementos de software da aplicação, parece natural que se apresente o *middleware* como uma interface, neste caso provida pela classe ECH (que oferece uma interface por onde os componentes de uma aplicação publicam e recebem eventos, além de assinarem canais de eventos), e se concentre a funcionalidade comum em uma classe base. Esta classe implementa funções de interação com o *middleware* para publicação e recepção de eventos, e assim está associada à classe ECH. Cada componente da aplicação possui em sua definição uma classe derivada dessa classe base. Neste trabalho, denomina-se esta classe de **BasicObject**.

O diagrama de classes da Figura 3.5 explicita sua definição e demonstra como se poderia especializá-la, com um exemplo da definição da classe derivada Speed, que representaria um sensor de velocidade. O diagrama mostra a associação entre a classe base e a interface ECH, e a definição das mensagens de evento com a classe Event.

O diagrama de classes possibilita definir a visão estática do sistema em termos de classes e seus relacionamentos, porém não revela como as instâncias destas classes estarão distribuídas, o que neste trabalho se denomina a configuração (*deployment*) da aplicação. Esta informação pode ser mostrada

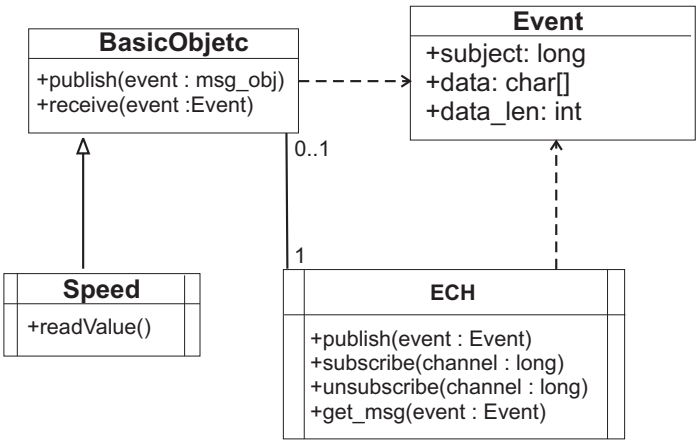

Figura 3.5: Diagrama de classes para uso da classe base.

com um diagrama de *deployment* da UML, conforme a Figura 3.6. Neste diagrama se evidencia o número de nodos que compõem o sistema, assim como quais componentes estão instanciados e em que nodos se localizam; desta forma obtém-se uma representação física da aplicação.

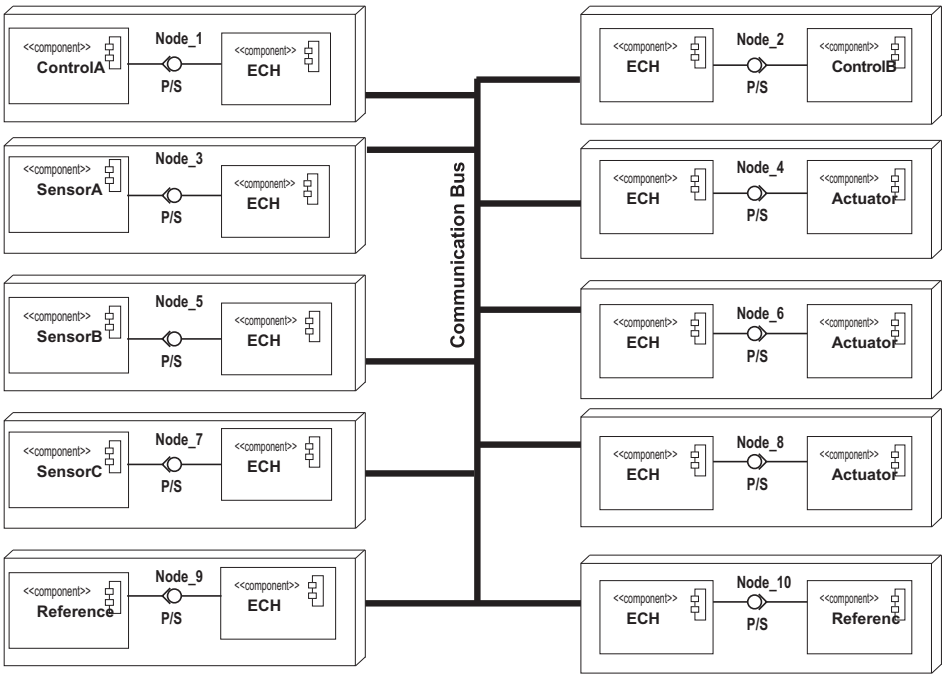

Figura 3.6: Configuração e distribuição dos componentes

Diagramas de classes e de *deployment* representam a estrutura estática da aplicação, porém pouco dizem a respeito de como os componentes se comportam ao longo do tempo. Este tipo de informação pode ser contida em diagramas de seqüência, que mostram as interações entre objetos, com as

sequências de mensagens que podem ser desencadeadas.

Os componentes de uma aplicação distribuída, incluindo o ECH, são intrinsecamente concorrentes. Tipicamente, cada componente reside em um nodo, e há também uma instância do ECH em cada nodo. Portanto, estes componentes surgem como objetos ativos na modelagem, e as mensagens trocadas entre eles são assíncronas.

Cada objeto que publica ou recebe eventos na aplicação tem um comportamento comum para comunicação. A publicação de um evento se faz pelo envio de uma mensagem *publish* para uma instância de ECH, que se encarrega de distribuí-la aos seus assinantes. A recepção de eventos pode ser feita tanto via *polling* do ECH no respectivo nodo, quanto pela notificação automática pelo *middleware* (o Cosmic atualmente implementa a primeira forma de recepção). Estas sequências de comunicação podem ser padronizadas de forma que um projetista não precise incluir seus detalhes durante a modelagem de uma nova aplicação. Assim, uma seqüência de comunicação pode ser entendida conforme ilustrado no diagrama de seqüência na Figura 3.7.

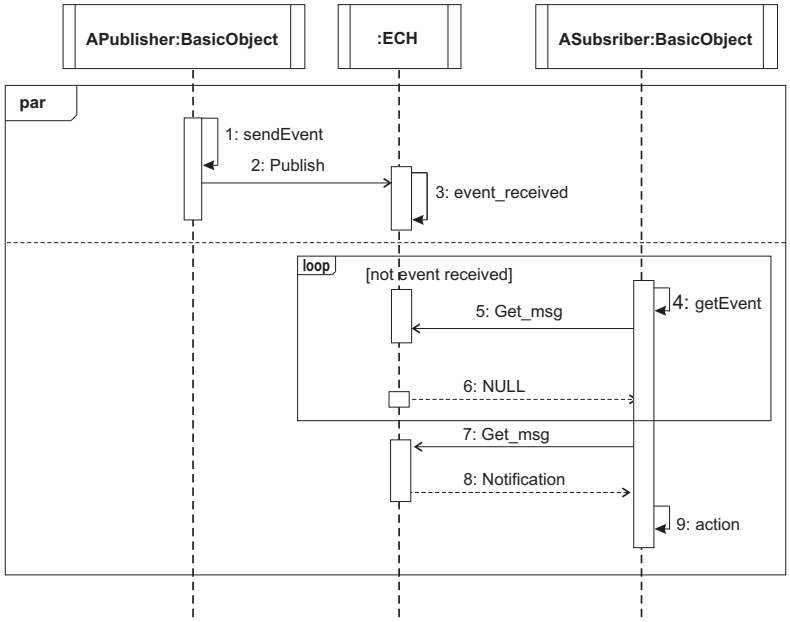

Figura 3.7: Diagrama de seqüência para comunicação.

A seqüência se compõe de duas partes concorrentes: a publicação e a recepção de evento. No diagrama, expressa-se isto com um fragmento *par*, que informa que seqüências em diferentes operandos são concorrentes. A recepção se faz por consulta ao *middleware*, até que um evento esteja disponível. Assim como no diagrama de classes, o *middleware* é abstraído como um objeto ECH, ocultando-se as mensagens trocadas dentro do *middleware* e entre instâncias de ECH nos nodos. Sabendo-se que toda

comunicação se dará como descrita neste diagrama, a modelagem dos comportamentos dos objetos pode se concentrar em suas finalidades. Desta forma, os diagramas de seqüência para os objetos da aplicação seriam semelhantes ao diagrama contido na Figura 3.8.

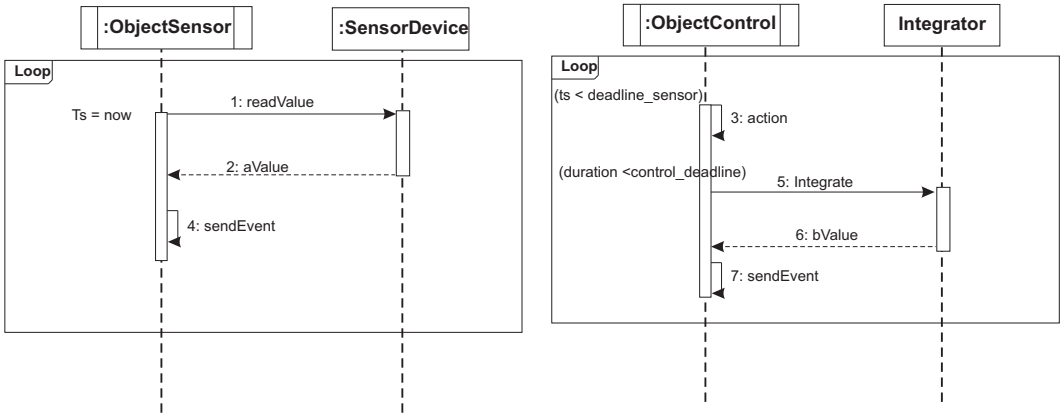

Figura 3.8: Diagrama de seqüência para os objetos da aplicação.

Neste diagrama, as ações desencadeadas pelos objetos estão evidentemente desacopladas. Neste exemplo, o objeto ObjectSensor lê periodicamente o valor de um sensor e o publica com um evento. Já o objeto ObjectControl recebe um evento, realiza algum cálculo e depois publica um evento com o resultado. Nota-se que esta modelagem não mostra quais objetos se comunicam, mas apenas o que cada objeto faz. Desta forma, o nível de abstração obtido permite que se concentre em "quem" precisa se comunicar, colocando em segundo plano "como" se dá a comunicação. Outro ponto importante que surge neste diagrama é a especifição de restrições temporais: há um *deadline* para a recepção de um evento do sensor (expressão $t_s <$ *sensor_deadline)*, e um *deadline* para a duração da tarefa de controle (expressão *duration* $<$ *control_deadline*).

A definição de quais objetos assinam determinados canais de eventos podem ser mostrados também por diagramas de seqüência. A assinatura de um canal de eventos resume-se ao envio de uma mensagem *subscribe* ao ECH, e o cancelamento de assinatura se vale da mensagem *unsubscribe*. Desta forma, um diagrama de seqüência provê grande liberdade de modelar as assinaturas e saídas de canais de eventos, tanto no caso estático quanto dinâmico — quando objetos mudam suas assinaturas de canais de eventos durante a execução da aplicação.

3.4 Conclusões

Neste capítulo, apresentou-se a arquitetura DOCAS para aplicações distribuídas que interagem através do protocolo *Publisher/Subscriber*.

Além disso, abordou-se o uso da linguagem UML para realizar a modelagem da arquitetura adotando-se o *middleware* COSMIC como meio de estudo. Foram identificados problemas comuns neste tipo de aplicação, como a interação com o *middleware*, a configuração estática ou dinâmica dos componentes, a abstração dos detalhes de comunicação e o desacoplamento entre os componentes que se comunicam. Com base nisto, propuseram-se diretrizes de modelagem capazes de contemplar estes aspectos e representar a estrutura e o comportamento de uma aplicação dessa natureza.

Em relação à modelagem, verificou-se que o uso da UML foi benéfico no sentido de organizar uma estrutura bem definida para depois iniciar a implementação da arquitetura baseada nessa estrutura. A modelagem realizada promove também a reutilização de software.

Capítulo 4

Implementação da Arquitetura DOCAS

4.1 Introdução

Neste capítulo apresenta-se a implementação da arquitetura proposta no Capítulo 3. Utilizaram-se as plataformas de *hardware* x86, Coldifre e PowerPC, a fim de validar a proposta aqui defendida.

Nas seções que se seguem são descritos os detalhes de cada implementação realizada, as configurações e alterações no *middleware*, bem como os detalhes de cada plataforma de hardware e também do sistema operacional embarcado utilizado.

4.1.1 x86

A primeira plataforma em que foi implementada a arquitetura proposta baseou-se em processadores da família x86. Essa plataforma de *hardware* tanto pode ser usada como estações de supervisão e monitoração, como também pode ser utilizada em sistemas baseados no barramento CAN. Existem diferentes placas de interface para PC´s via barramentos internos ISA ou PCI e barramento CAN. Assim, usando-se PC´s com placas controladoras PCI ou ISA e dando suporte a redes CAN, pode-se montar uma rede heterogênea valendo-se dos protocolos CAN e TCP/IP. Esse tipo de sistema apresenta custo relativamente baixo, sendo muito utilizado em pesquisas.

O PC utilizado como base de desenvolvimento, testes, configurações e compilações possui processador Pentium 550 Mhz com 256 *megabytes* de memória RAM.

Já na parte relacionada ao sistema operacional, a aplicação do conceito de objetos ativos distribuídos e que podem ser executados de forma concorrente, implica diretamente na utilização de um sistema operacional multitarefa, que permita a interação entre objetos distribuídos presentes em di-

ferentes plataformas de *hardware*, interligadas através de uma rede. Portanto, o sistema operacional deve dar suporte ao *hardware* escolhido, tornando possível também a implementação dos objetos como processos concorrentes.

Há muitos sistemas operacionais que dão esse tipo de suporte, porém paga-se pela utilização de alguns. Uma alternativa aos sistemas comerciais disponíveis é o sistema operacional Linux, geralmente reconhecido como sofisticado, confiável e portável. Esse sistema operacional possui praticamente todas as características e ferramentas de desenvolvimento que os sistemas UNIX possuem, fornece serviços de TCP/IP, interfaces gráficas, possui compiladores para diversas linguagens, tais como C, C++, Java e, além disso, é um sistema aberto.

O porte do COSMIC (versão 1.4) para arquitetura x86 aconteceu de forma natural, ou seja, sem nenhum problema de compilação ou execução. Entretanto, como o *hardware* utilizado não tem suporte para a rede CAN, só foi compilada e executada a parte do *middleware* referente à rede TCP/IP, ou seja, o *gateway* que dá suporte a rede CAN-TCP/IP não foi compilado.

Na próxima seção serão apresentadas as outras plataformas de *hardware* no qual foi implementada a arquitetura proposta, com suporte à rede CAN.

4.1.2 PowerPC

O suporte a uma rede CAN foi alcançado através de um PowerPC. Essa segunda plataforma de *hardware* além de dar suporte a rede CAN, vem sendo bastante utilizada em sistemas embarcados. Desse modo, pode assumir o papel de um *gateway* interligando redes TCP/IP e CAN presentes em diferentes plataformas de *hardware*.

Os processadores PowerPC de 32 *bits* tornaram-se um padrão bastante utilizado no desenvolvimento de processadores embarcados. Como forma de manter baixos os custos e o alto volume de produtos competitivos, o *core* da CPU é normalmente empacotado em um *System-On-Chip* (SOC) de um circuito integrado. Nos SOCs ficam o *core* do processador, a *cachê* e o processador de dados locais; ao longo do *chip* aparecem sincronizadores, temporizadores, memória(SDRAM), periféricos(network, serial I/O) e controladores de barramentos (PCI, PCI-X, ROM/Flash, I2C).

A plataforma de *hardware* PowerPC utilizada neste trabalho faz parte de um *kit* da Freescale. O *kit* LITE5200 *Evaluation Board*, Figura 4.1, possui um processador de 32 *bits* que opera a uma freqüência de *clock* de 400MHz, 16 *MBytes* de memória *flash*, 64 *MBytes* de memória RAM e possui

Figura 4.1: PowerPC

os seguintes periféricos de comunicação:

- Uma porta usb 1.1;
- Uma interface EThernet 10/100Mbps;
- Uma porta *serial* RS232;
- Duas portas CANBUS 2.0 A/B de alta velocidade, *frames* padrão e extendido e taxa de *bits* programável de até 1 Mbps;
- Um barramento PCI;
- Um barramento IDE;
- Uma porta J1850, baseada em protocolo desenvolvido pela Society of Automotive Engineers (SAE);
- Uma porta para protocolo I2C;
- Um controlador serial de periféricos PSC1;
- Uma interface de comunicação serial SP1;
- Uma interface ATA/IDE Interface;
- Uma porta de depuração COP/JTAG.

4.1.3 Coldfire

A terceira plataforma de *hardware* utilizada neste trabalho foi o *kit* de desenvolvimento da Coldfire modelo M5484EVB, Figura 4.2. Essa plataforma possui processador embarcado de alta performance, baseado no *core* Coldfire v4e. O *kit* possui os seguintes periféricos de comunicação:

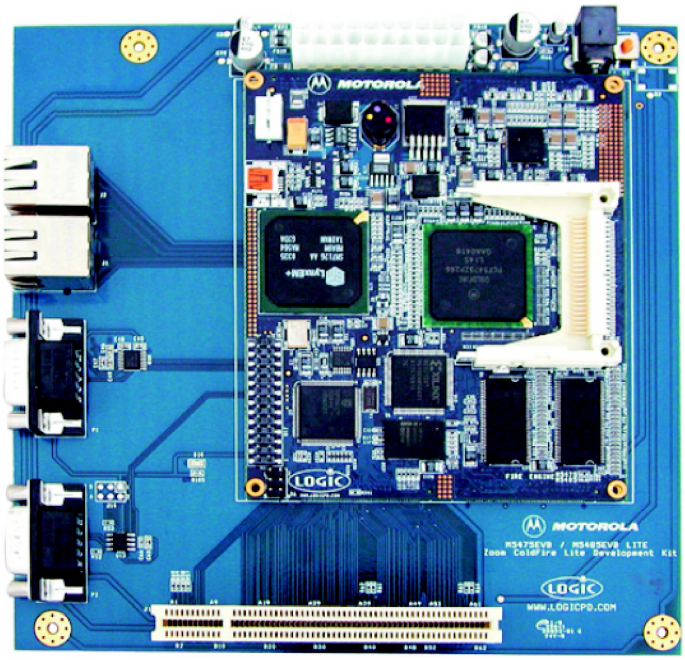

Figura 4.2: Coldfire

- Uma porta CAN;
- Uma interface de comunicação serial;
- Duas portas Ethernet;
- Um conector ATX;
- Um barramento PCI.

Para a plataforma Coldfire e PowerPC, a parte destinada ao sistema operacional e à implementação do *middleware* é um pouco mais complexa do que a destinada à arquitetura x86. Sendo assim, foi dividida em seções e apresentada a seguir.

4.1.4 Sistema Operacional Embarcado

O *Embedded Linux Development Kit* (ELDK) é um *kit* que já vem com diversas configurações para instalar um sistema operacional embarcado em diferentes arquiteturas, inclusive na do PowerPC 5200 Lite. Inclui também ferramentas de desenvolvimento GNU, como compiladores, *binutils*, *gdb*, etc. Com as configurações pré-ajustadas, o sistema operacional já contém algumas funcionalidades (bibliotecas, ferramentas pré-compiladas).

O código fonte é totalmente *free* e está disponível para PowerPC, ARM e MIPS, e para PC/Linux e SPARC/Solaris (denx). Inclui todos os *patches*, extensões, *scripts* e programas. Pode ser encontrado em (ENGINEERING, 2006).

4.2 Instalação do Sistema

Para instalar o ELDK, é necessário um PC com uma entrada *ethernet* e uma entrada *serial* rodando o sistema operacional Linux. Para fazer a comunicação entre o PC e o Kit LITE5200 *Evaluation Board*, utilizou-se um cabo de rede par trançado (RJ-45) no modo *cross-over* e o *Trivial FileTransfer Protocol* (TFTP).

Na fase de desenvolvimento foi utilizado o *Network File System* (NFS), isto é, os arquivos foram acessados do PowerPC para o PC de forma transparente. Com o uso do NFS o tempo de desenvolvimento do projeto diminuiu, ou seja, as alterações puderam ser feitas no PC, ao invés de serem gravadas em uma memória externa para depois serem utilizadas no PowerPC ou mesmo gravadas na memória *flash*.

Para acessar diretamente o PowerPC foi utilizado um cabo *serial* (RS-232) e o aplicativo Minicom[1]. Uma vez configurada a comunicação entre o PowerPC e o PC, ajustou-se o *Boot Loader*, programa que é executado quando a máquina se inicia. Há dois Boot Loaders, o dBUG e o U-boot (*Universal Boot Loader*).

O dBug foi desenvolvido para ser utilizado como *ROM Monitor/Debbuger* e já veio instalado na parte *High Boot* (0xFFF00000) do PowerPC. Como possíveis formas de utilização do dBug, pode-se citar o comando *set*, usado para configurar, por exemplo, *baud*, *mac*, *client*, *server*, *netmask*, etc.

O boot loader U-Boot é considerado mais flexível quando comparado com o dBug (PELGRIMS; VELDE; VONDEL, 2004); por essa razão foi escolhido. Esse *Boot Loader* tem suporte para diferentes

[1] http://linux.die.net/man/1/minicom

arquiteturas (MPC5xx, 82xx, 7xx, 74xx, 4xx; ARM7 9, Strong ARM e Xscale; MIPS 4Kc, 5Kc; x86;) e este disponibilizado sob licença GPL (PELGRIMS; VELDE; VONDEL, 2004).

O U-Boot foi alocado na memória *flash*, em um local diferente do dBug. Dessa maneira, pôde-se preservar o Monitor Motorola localizado em outra parte da memória para que, no caso de erro, se pudesse acessar o outro *Boot Loader*.

O Motorola MPC5200 fornece *jumpers* na placa (J73, J74, J75), Figura 4.3, que permitem mudar a configuração do *boot*. Eles são chamados, de acordo com o endereço, de *High Boot* e *Low Boot*. *High Boot*, neste caso dBUG, começa no endereço 0xFFF00000, e *Low Boot* inicia no endereço 0xFF000000. Além disso, pode-se configurar o U-Boot para rodar na memória RAM ou na memória *flash*.

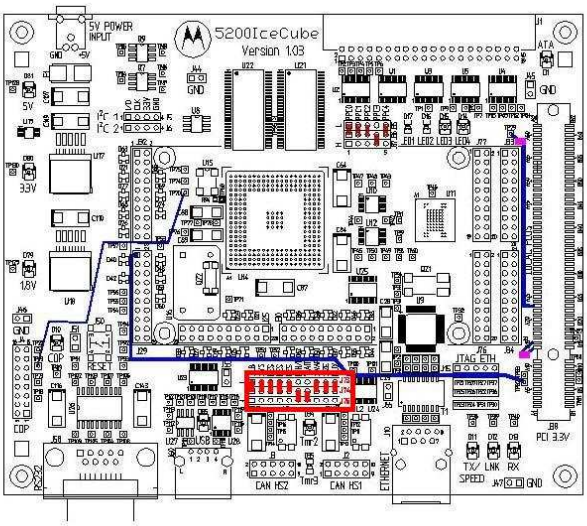

Figura 4.3: Jumpers.

Depois de configurado o *Boot Loader* pôde-se começar a trabalhar com o kernel do linux. O carregamento do sistema operacional pode se dar através de uma imagem na memória *flash* ou estar localizado em outro PC, através do TFTP. Depois de ajustado o *boot* e o *kernel* foi possível *bootar* e carregar o sistema operacional.

Já o processo de instalação para o *kit* Coldfire modelo M5484EVB é detalhado a seguir.

Antes de compilar e executar o *middleware* COSMIC nessa plataforma de *hardware* foi necessário primeiro configurar a plataforma de *hardware*. Seguem abaixo os detalhes para configurar a placa M5484EVB:

- Instalar LTIB baseado no Board Support Package (BSP).
- Rodar LTIB para construir a imagem para o *host* desejado.
- *Bootar* na placa com a imagem gerada pelo LTIB.

O Linux Target Image Builder (LTIB), Figura 4.4, é uma ferramenta que auxilia no desenvolvimento de uma imagem do sistema operacional linux para a plataforma desejada, neste caso MCF5485EVB. Com isso pode-se construir o *colilo* (Coldfire Linux Loader - Boot Loader) e o sistema de arquivos *root*. Essa ferramenta roda em uma arquitetura de *hardware* x86 com o sistema operacional linux.

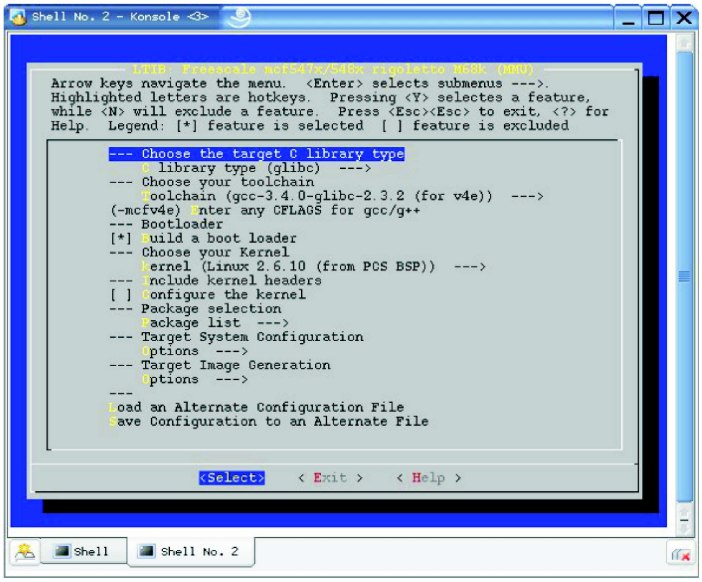

Figura 4.4: LTIB

Com uma imagem para a placa Coldfire pronta, configura-se o nfs e o tftp para auxiliar no carregamento do sistema operacional e na troca de arquivos. Para configurar o boot loader utilizou-se o aplicativo Minicom. No Axeno A são detalhados os passos realizados para a implantação da distribuição linux LTIB no *kit* de desenvolvimento Freescale M5484, utilizando-se como *host* um PC com a distribuição linux Ubuntu.

4.3 Implementações, Alterações e Configurações

No processo de implementação do *middleware* COSMIC ao sistema operacional da DENX (ELDK) compilou-se e executou-se o *middleware* na placa MPC5200. A compilação do *middleware*

COSMIC foi dividida em duas etapas: etapa TCP/IP e etapa CAN.

Na primeira etapa, a compilação não apresentou problemas; porém, na execução, notou-se que a seqüência de processamento não aconteceu da forma esperada. O problema foi encontrado na parte inicial e a solução foi ajustar alguns detalhes para que a execução continuasse da forma esperada.

Já na segunda etapa, inicialmente foi necessário analisar o suporte da placa MPC5200 à rede CAN através do controlador CAN, para depois compilar e executar o *middleware*.

Como o *middleware* COSMIC (versão 1.4) está configurado para trabalhar com o controlador SJA1000 CAN (PHILIPS, 2000) da Philips, o *driver* para esse controlador fornece todo o suporte necessário à rede CAN; e como o PowerPC possui um controlador msCAN (MOTOROLA, 2004) interno (On Chip), que difere do outro controlador, é necessário primeiro ajustar um *driver* CAN para a placa MPC5200, para depois fazer a ligação do *middleware* ao *driver*.

A solução para esse problema foi buscar um *driver* que trabalhasse corretamente com o sistema da DENX, dando suporte ao controlador msCAN. O drive PCAN (TECHNIK, 2006) atendeu às necessidades do projeto, habilitando o funcionamento da rede CAN na placa MPC5200.

Para o *driver* funcionar corretamente, foi preciso recompilar o kernel adicionando o *driver* com isso, habilitou-se a rede CAN, e alguns testes com o protocolo CAN puderam ser realizados, já que a placa MPC5200 conta com duas portas CANBUS.

Tabela 4.1: Pinagem CAN

Pino	**Sinal**	**Descrição**
1	-	-
2	CAN_L	CAN Low
3	CAN_GND	Terra
4	-	-
5	(CAN_SHLD)	Blindagem opcional
6	GND	Terra opcional
7	CAN_H	CAN High
8	-	-
9	CAN_V+	Fonte

A ligação entre as duas portas CANBUS foi feita através de fios; a especificação da pinagem CANBUS é vista na Tabela 4.1 conforme Cia (CIA, 2006). O primeiro fio está conectado no pino 2 CAN_H (CAN High) da porta CAN1 e ao pino 2 da porta CAN2, já o segundo fio esta conectado no pino 7 CAN L (CAN Low) da porta CAN1 e ao pino 7 da porta CAN2.

A estrutura do PCAN, Figura 4.5, mostra como ocorre o acesso ao *driver*, ou seja, este é acessado em um nível mais alto, através de uma biblioteca dinâmica “libpcan.so”. Essa biblioteca fornece um

modo fácil de interagir com as propriedades do *driver* PCAN.

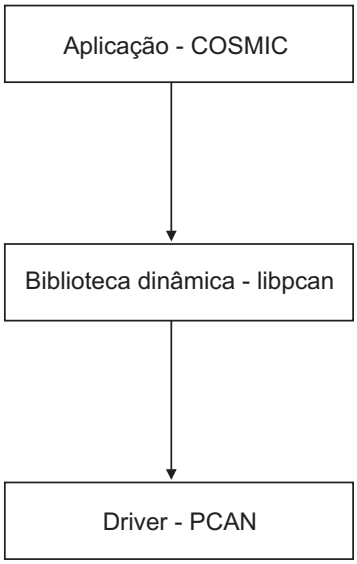

Figura 4.5: Estrutura do PCAN

A biblioteca dinâmica, chamada também de biblioteca compartilhada (*shared library*) é como um executável, exceto pelo fato de que não pode ser executada sozinha; ela consiste de um bloco de código executável que é carregado na memória depois que uma aplicação é iniciada, e pode ser compartilhada por uma ou mais aplicações. Seguem abaixo algumas de suas funções:

- CAN_Open, abre uma porta CAN;
- CAN_Init, inicializa a rede CAN;
- CAN_Close, fecha a conexão;
- CAN_Status, exibe o *status* da rede CAN;
- CAN_Write, envia uma mensagem para o barramento CAN;
- CAN_Read, lê uma mensagem da rede CAN.

Essas funções são usadas tanto no Linux como no Windows, entretanto no Linux há algumas chamadas especificas, como:

- LINUX_CAN_Open, abre uma porta CAN;
- LINUX_CAN_Statistics, mostra as estatísticas.

Quando uma conexão é inicializada, o *frame* estendido e o *baud rate* igual a 500 KBits/s são ajustados por *default*. Essas configurações podem ser alteradas através da função CAN_Init, dependendo dos requisitos do sistema.

O uso dessas funções ajuda a ocultar detalhes baixo nível de comunicação com o *driver*, porém o *middleware* acessa o *device driver* através de outras funções. Dessa maneira, para não mudar a semântica do COSMIC, a interação entre o *middleware* e o *driver* deu-se através das funções open, read, select, ioctl.

O acesso ao *driver* é provido por uma entrada no diretório de dispositivos do ElDK (/dev), que deve ser aberto inicialmente por um comando *open()* para possibilitar as configurações subseqüentes. Como são dois dispositivos, há duas portas: CAN pcan1 e pcan2. Uma vez aberta a taxa de transmissão pode ser configurada através do comando *ioctl()*, e os valores BTR0BTR1 *baud-rate* para o MPC5200 estão definidos no arquivo "pcan_main.h". São eles: 1 MBits/s, 500 KBits/s, 250 KBits/s, 125 KBits/s, 100 KBits/s e 50 KBits/s. Além disso, a função *ioctl()*, pode ser usada para configurar máscaras de filtragem.

A operação de leitura é executada através da função *read()*; já a função *write()*, por sua vez, faz com que o *device driver* armazene a mensagem no *buffer* de transmissão e tente transmití-la. Para fechar a conexão CAN utiliza-se a função *close()*.

O componente *gateway* (Event Channel Handler) da rede CAN sofreu algumas modificações por causa das alterações citadas anteriormente. Sendo assim, foram definidas duas constantes no arquivo "main.c" para identificar os dispositivos CAN. São elas: CAN_DEVICE1 ("/dev/pcan1") e CAN_DEVICE2 ("/dev/pcan2"). No arquivo "can_conn.c", onde são tratadas as conexões CAN, estas são abertas através da função *open()* em modo de não-bloqueio, de forma que se não existirem mensagens no *buffer* do respectivo dispositivo, o Event Channel Handler não fica bloqueado esperando mensagens. Além disso, no arquivo "can_conn.c" são tratadas as leituras no barramento CAN através da função *read()*.

Com isso a estrutura do COSMIC foi preservada, porém outro problema surgiu. O mapeamento da estrutura de dados (dicionário de dados), utilizados pelo *middleware* é diferente da estrutura de dados disponibilizada pelo PCAN; desse modo, foi montada uma estrutura que mapeia os dados da estrutura PCAN para a estrutura do COSMIC, para possibilitar o envio de mensagens através de funções alto nível do PCAN, como também através de funções baixo nível.

Tabela 4.2: Mensagens CAN

etag	**D[0]**	**D[1]**	**D[2]**	**D[3]**	**Descrição**
0	*stage*	*fragmentos*	*TxNode*	-	SSI
1	*stage*	*fragmentos*	*TxNode*	-	RSI
2	canal	...	...	...	requesição *etag*
3	*TxNode*	1	etag(MSB)	etag(LSB)	*subscribe*
3	*TxNode*	2	etag(MSB)	etag(LSB)	unsubscribe
3	*TxNode*	3	etag(MSB)	etag(LSB)	resposta *etag*
$>= 5$	-	-	-	-	*event*

Código 4.1: Estrutura da mensagem CAN no COSMIC

```
1 typedef struct {
2         CanId id;
3         int type; /* standard or extended frame */
4         int rtr; /* remote transmission */
5         int len; /* data length 0..8 */
6         unsigned char d[8]; /* data */
7 } canmsg;
```

Então os campos id, type, rtr, len, d[8] (estrutura do COSMIC) recebem os dados, os quais são do mesmo tipo que deveriam receber, oriundos da rede CAN. Essa conversão é realizada no arquivo "can_conn.c" quando a mensagem é recebida. Desse modo, com a inclusão do PCAN, não foi necessário mudar a estrutura do *middleware* mas apenas ajustá-la e, além disso, não foi necessário montar outra estrutura de dados para o PCAN, ou seja, foi utilizada a estrutura presente no PCAN, o que facilitou o desenvolvimento.

Sendo assim, quando uma mensagem é lançada ao barramento CAN por um nodo, a parte do COSMIC que trata as mensagens CAN localmente (ECH) a captura. Nesta etapa, a mensagem é estruturada para que os outros nodos possam recebê-la a mesma conforme a estrutura do COSMIC. O ECH local pode enviar a mensagem para o *gateway*, e este pode enviá-la para a rede TCP/IP, dependendo dos assinantes do evento.

Mostra-se, no Código 4.1, a estrutura de uma mensagem CAN no COSMIC. Já a Tabela 4.2 mostra os tipos das mensagens CAN.

O suporte a rede CAN para plataforma M5484 foi possível através do *driver* can4linux[2]; com isso, iniciou-se a configuração do *middleware* COSMIC parte CAN para a plataforma Coldfire. As altera-

[2]http://www.canexperts.de/software/can4linux/index.htmlm

ções no *middleware* foram realizadas conforme o suporte da biblioteca do *driver*, ou seja, aproveitou-se o suporte oferecido pelo *driver* e apenas adaptou-se o *middleware* para utilizar esses serviços. Com isso pôde-se ligar o *kit* LITE5200 com o *kit* M5484EVB através de uma rede CAN.

4.3.1 Mensagens CAN no COSMIC

A Figura 4.6 ilustra as etapas de configuração entre o nodo e o *gateway*, neste caso mostrando como é realizada a identificação de um nodo CAN e o procedimento realizado para obter uma *etag*.

O *Request Short Node ID Message* (RSI) é uma mensagem usada pelo nodo (ECH) para obter o identificador único. Sempre que um novo nodo CAN é conectado, uma configuração de protocolo entre o nodo (ECH) e o ECB é realizada. Durante a configuração do protocolo, o nodo (ECH) envia o seu identificador de 64 *bits* em oito partes. Essa etapa é realizada para trocar o identificador do nodo de 64 *bits* por um identificador menor de 7 *bits*.

No envio das partes, aquela destinada aos dados deve estar vazia para não gerar inconsistência na camada física CAN-Bus. Então, toda a informação tem que ser transmitida em pacotes de 1-Byte, encapsulado em mensagens CAN-Bus (BRUDNA et al., 2003).

Desse modo, para fazer o pedido de um identificador CAN, o nodo envia um etag (*event tag*) = 1, d[0] identificando qual parte, d[1] contém o *bits* que estão sendo transmitidos e por meio do campo TxNode, quem está enviando os dados.

Com essa configuração, resolve-se o problema de haver mensagens sem identificadores únicos em uma rede CAN-Bus (BOSCH, 1991).

A Supply Short Node ID Message (SSI) é uma mensagem usada pelo CAN-ECB (*gateway*) para responder à mensagem RSI. Depois de receber a última mensagem (última parte dos 64 *bits*) o CAN-ECB envia uma mensagem de resposta SSI para o ECH com 7 *bits* no campo d[2] identificado como TxNode um nodo CAN, o qual sempre terá que participar de uma comunicação desse tipo para poder iniciar uma assinatura ou publicação de algum evento no respectivo canal.

Quando o *etag* é igual a 2, significa que será usado pelo ECH para receber um ID do canal. O campo d[0] identifica o canal; em resposta, o CAN-EB (*gateway*) envia uma mensagem de resposta. Quando o *etag* é igual a 3, TxNode no campo d[0], o campo d[1] = 3 e d[3] = etag(LSB - *Least Significant Bit*), o que implica que o campo d[3] contém o *byte* de dados menos significativos, enquanto o campo d[2] contém correspondentemente o *byte* de dados com maior valor (MSB *More Significant*

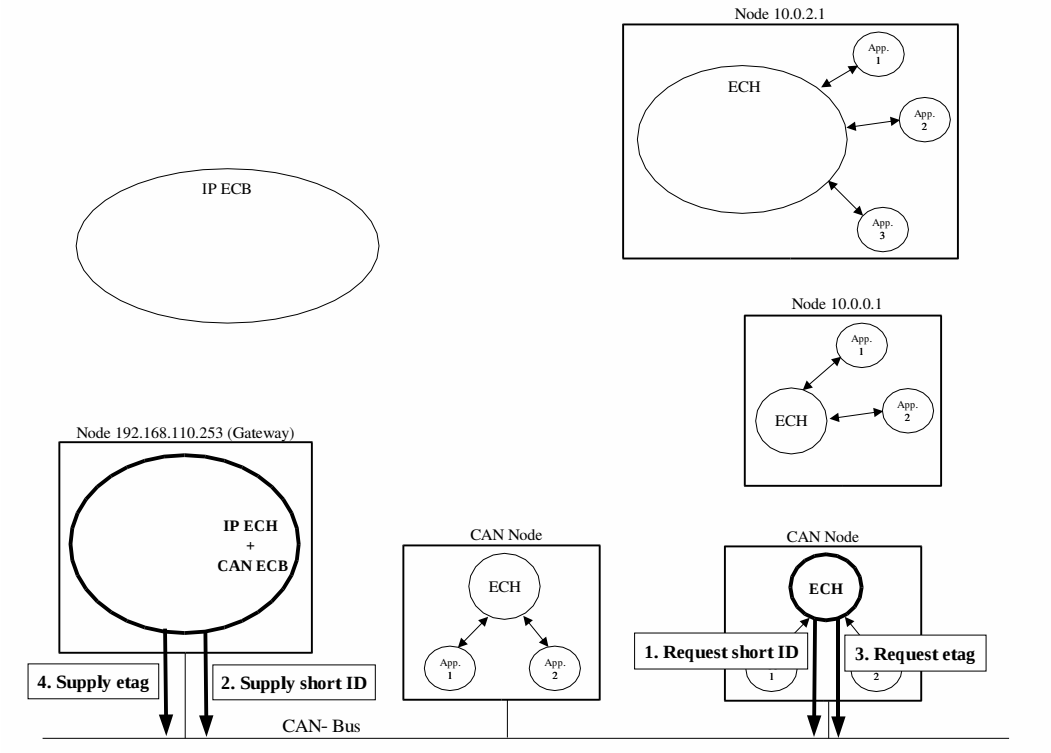

Figura 4.6: CAN-Bus (BRUDNA, 2003)

Bit). Toda essa configuração é realizada entre o *gateway* e ECH.

Para realizar uma requisição *subscribe*, mensagem enviada pelo ECH para o *gateway* a fim de permitir que os nodos CAN recebam eventos publicados fora da rede CAN (neste caso rede TCP/IP), os parâmetros são: etag=3, TxNode em d[0], d[1]=1. Para realizar uma requisição *unsubscribe* o processo é o mesmo, mudando apenas o conteúdo do campo d[1] de 1 para 2. Já o tipo de mensagem usada para publicar um evento na rede CAN utiliza $etag >= 5$ e os campos d[0] até d[7] contêm dados do evento.

4.4 Conclusões

Neste capítulo apresentaram-se as plataformas de *hardware* utilizadas no projeto, as configurações, a implementação, e o sistema operacional. Além disso, indicaram-se as alterações no *middleware* para que ele trabalhasse corretamente com outros tipos de controladores CAN, como por exemplo o msCAN, utilizando a mesma estrutura de dados presente no PCAN e no can4linux.

Sendo assim, as alterações e configurações apresentadas mostram como se pode portar a arquite-

tura para um sistema operacional embarcado utilizando outras plataformas de *hardware* de modo que o esforço de implementação da arquitetura, por parte do programador, seja o mínimo possível.

Capítulo 5

Estudo de Caso

Neste capítulo apresenta-se um estudo de caso referente a um veículo autônomo que utiliza controle preditivo baseado em modelo (CPBM), conforme descrito em (GOMES, 2006). Nesse estudo o sistema proposto por esse autor é alterado para representar conceitos de comunicação baseados no modelo *Publisher/Subscriber*, com o objetivo de distribuir os dispositivos do veículo (sensores, atuadores e controladores) e interligá-los através de uma rede CAN, comumente usada nos automóveis.

5.1 Descrição do Sistema Original

Um veículo autônomo tem como objetivo percorrer um caminho preestabelecido, o qual é definido por um conjunto de pontos chamados aqui de pontos de referência, formando, desse modo, um circuito virtual, em que cada ponto contém a orientação necessária para que o veículo não desvie de seu objetivo. (GOMES, 2006) desenvolveu um sistema embarcado para tal tarefa, incluindo as seguintes funcionalidades:

- Controle de trajetória;
- Controle de velocidade;
- Sistema anti-colisão;
- Seleção de trajetórias;
- Ajuste de modo de operação;
- Sistema de posicionamento real;

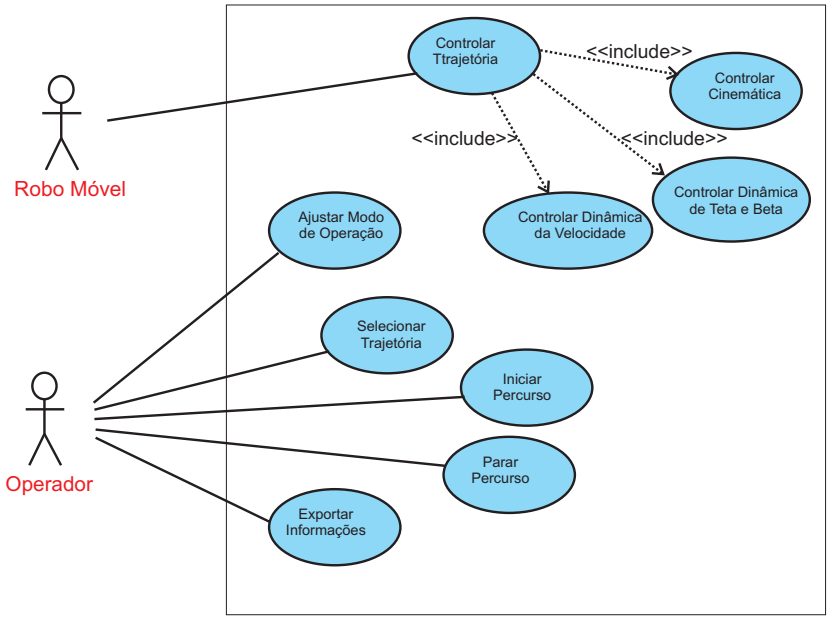

Figura 5.1: Diagrama Caso de Uso (GOMES, 2006)

- Sistema de comunicação;
- Interface com o operador.

Na parte de modelagem do sistema foram apresentados por (GOMES, 2006) alguns diagramas utilizando a linguagem UML. Um diagrama de caso de uso é uma técnica de modelagem usada para descrever o que um novo sistema deve fazer, moldando as interações entre os clientes externos e os casos de uso do sistema. Um caso de uso é a descrição de um conjunto de ações, incluindo variantes, que o sistema realiza para chegar a um resultado de valor observável para um ator (BOOCH; RUMBAUGH; JACOBSON, 1998). No diagrama de casos de uso, quaisquer entidades externas que interagem com o sistema, tais como pessoas, robôs ou mesmo outros sistemas, são chamadas de atores e são representados por bonecos. Os casos de uso são modelados por elipses dentro de um contorno retangular, ou caixa do sistema, que representa o sistema propriamente dito.

A Figura 5.1 ilustra o diagrama de caso de uso do sistema original. Os casos de uso *Iniciar Percurso* e *Parar Percurso* têm a função de controlar a operação do sistema, permitindo comandar o início e o fim do deslocamento do robô móvel. O caso de uso *Visualizar Informações* transmite a algum recurso de *hardware* as informações referentes ao percurso. *Selecionar Trajetória* é o caso de uso que permite, através do banco de dados, definir a trajetória a ser percorrida. Já *Ajustar Modo de Operação* é o caso de uso que redefine a lei de controle de modo a atender diferentes critérios de

desempenho. Por fim, *Controlar Trajetória* é o caso de uso responsável pela condução do veículo ou robô ao longo da trajetória predeterminada, sendo dividido em casos de uso específicos para o controle do modelo cinemático, do modelo dinâmico de β e θ e do modelo dinâmico da velocidade v no centro de massa. O caso de uso *Controlar Trajetória* inclui ainda outros casos de uso relacionados a outras entidades externas além do ator Robô Móvel.

A etapa seguinte consiste na definição das classes do sistema e no relacionamento entre elas. As classes e seu relacionamentos são modeladas em UML através de diagramas de classes, nos quais são representadas por um retângulo dividido em três partes. A parte superior contém o nome da classe, a parte do meio contém os atributos e a parte inferior contém as operações da classe. Atributos de uma classe são dados de tipos primitivos ou tipos criados pelo usuário, como outras classes. Já as operações são funções membros da classe, que têm acesso a todos os atributos desta. As classes se relacionam umas com as outras através de associações representadas por uma linha cheia que conecta duas classes. As associações unidirecionais apresentam um seta em uma das extremidades, indicando a sua direção. Já as associações bidirecionais não apresentam setas. Os números junto às linhas de associação expressam valores de multiplicidade e indicam quantos objetos de uma classe participam das associações. Para modelar o conceito de herança, segundo o qual as classes novas são criadas a partir de classes existentes, pela absorção de seus atributos e operações, é utilizado o símbolo de um triângulo em uma extremidade de associação. A classe com o triângulo na sua extremidade da linha é denominada classe base, e as classes da outra extremidade são denominadas classes derivadas, pois estas últimas herdam as propriedades da classe base.

O diagrama de classes devenvolvido é mostrado na Figura 5.2. Pode-se observar neste diagrama a presença de uma classe central, classe MPC, que centraliza o controle das funcionalidades descritas. Esta classe agrega outras classes que auxiliam nas tarefas de controle. São elas: classe *ControlLaw-Processor* e a classe *CommandProcessor*.

A classe *PurePursuit* é usada pelo MPC em controle de trajetória de veículos que utiliza sistema de coordenadas locais. A trajetória de aproximação é calculada ao longo do horizonte de predição, de forma a proporcionar uma aproximação suave até o ponto de referência atual, evitando variações elevadas no ângulo de orientação. Trajetórias de aproximação são necessárias em situações em que o veículo encontra-se muito distante do ponto de referência desejado. Nestas situações, o erro elevado faria com que a ação de controle provocasse variações elevadas no ângulo de orientação, acentuando

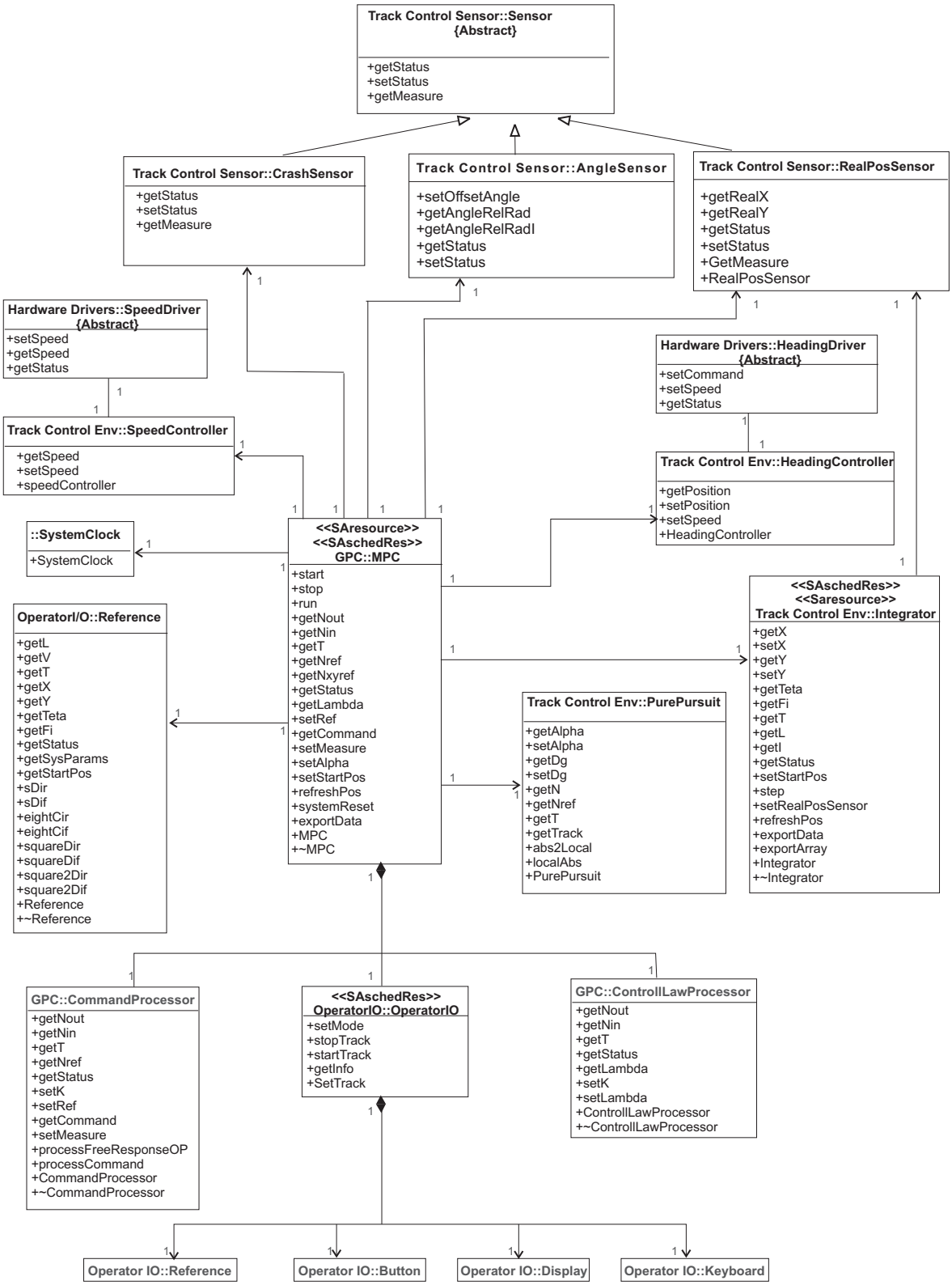

Figura 5.2: Diagrama de Classes do Sistema Original (GOMES, 2006)

as não-linearidades do sistema e tornando impróprio o modelo linear utilizado (GOMES, 2006). A classe *Pure Pursuit* opera ao longo de todo o percurso da trajetória, de modo que a trajetória de aproximação gerada a cada período de amostragem é a referência que efetivamente é utilizada pelo controle. A classe *Integrator* realiza a integração das coordenadas x e y e do ângulo de orientação do veículo, com base na leitura do ângulo de orientação teta.

Sendo assim, a classe *MPC* atua de forma centralizada, recebe mensagens das classes *OperatorIO* e *SystemClock*, e envia mensagens às demais classes do sistema; ou seja, todas as classes precisam conhecer ou ter uma referência para a classe *MPC*.

Portanto, esse sistema requer a interação entre os diversos dispositivos do veículo e também dos dados externos do ambiente. Porém o sistema está fortemente acoplado, interagindo através de chamadas de métodos e tendo a ligação entre objetos feita em tempo de projeto. Como a comunicação entre os dispositivos do sistema embarcado distribuído deve ser anônima e assíncrona, significa que o modelo atual não é adequado.

O modelo *Publisher/Subscriber* é muito utilizado nesses casos, por causa do desacoplamento dos componentes e pelo fato de que a comunicação é anônima. Sendo assim, uma nova estruturação dos componentes de software do sistema proposto por (GOMES, 2006) se faz necessária para tornar distribuído o sistema centralizado.

5.2 Reestruturação do Sistema

Pretende-se com a reestruturação do sistema representar conceitos de comunicação baseados no modelo *Publisher/Subscriber*, com o objetivo de distribuir os dispositivos do veículo. A interligação desses dispositivos é feita através dos protocolos CAN e TCP/IP, ou seja, é necessário representar como é realizada a comunicação entre os diferentes protocolos de rede. Além disso, nessa nova estrutura há mais de uma plataforma de *hardware*; sendo assim, leva-se em conta a comunicação entre elas e como estão dispostos os componentes em cada plataforma de *hardware*.

Nesse novo modelo, mostrado na Figura 5.3, os componentes estão dispostos em nodos diferentes e não há ligação direta entre os componentes, mesmo que se tenha dois deles na mesma plataforma de *hardware*.

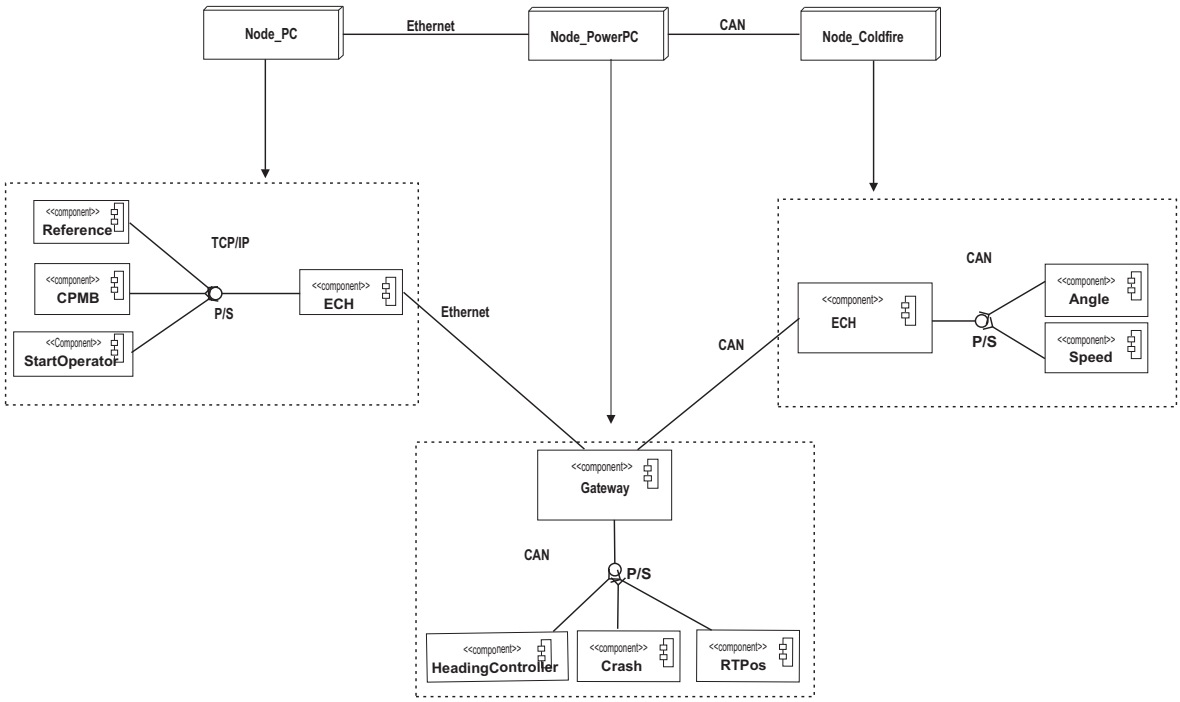

Figura 5.3: Diagrama de Componentes e Deployment

Desta forma, foi necessário realizarem-se modificações na modelagem original do sistema, pois antes os componentes estavam fortemente acoplados, havendo operações entre eles, através de chamadas de métodos.

Nessa nova estrutura, a comunicação é realizada através de eventos. Esses eventos são gerados pelos produtores de informação ou publicadores de eventos, de modo assíncrono. Além disto, no modelo apresentado por (GOMES, 2006) havia um único fluxo de execução que perfazia uma seqüência de invocações de métodos durante o laço de controle. Já no novo modelo, os componentes executam seus laços concorrentemente.

Como ilustrado na Figura 5.3, o Nodo_PC e o Nodo_PowerPC comunicam-se através de uma rede *Ethernet*, e a comunicação entre o Nodo_PowerPC e o nodo Nodo_Coldfire é realizada através de uma rede CAN. Já a comunicação entre os componentes presentes em um mesmo nodo depende do tipo de suporte oferecido pelo *hardware*. No caso do Nodo_PowerPC e do Nodo_Coldfire, há suporte à rede CAN, enquanto no Nodo_PC a rede é TCP/IP.

O *gateway* faz a ligação entre os componentes da rede CAN, Nodo_PowerPC e Nodo_Coldfire, com o protocolo TCP/IP disponibilizando a interface *Publisher/Subscriber* para que os componentes desse nodo possam publicar ou receber eventos. Além de disponibilizar eventos da rede CAN para

TCP/IP e receber eventos da rede TCP/IP, o *gateway* atua como um *Event Channel Handler* para rede CAN no Nodo_PowerPC.

Já no Nodo_PC o ECH disponibiliza a interface *Publisher/Subscriber* para que os componentes internos, rede TCP/IP, se comuniquem com os componentes do próprio nodo e com os componentes do Nodo_PowerPC. Da mesma forma, o ECH do Nodo_Coldfire disponibiliza a interface *Publisher/-Subscriber* para que os componentes internos, rede CAN, se comuniquem com os componentes do próprio nodo e com os componentes do Nodo_PowerPC e do nodo Nodo_PC, através do *gateway*.

Na Figura 5.4 apresenta-se o diagrama de classes do sistema modificado, o qual é baseado no modelo de comunicação *Publisher/Subscriber*. O diagrama se valeu de estereótipos apresentados anteriormente no Capítulo 3 (Figura 3.5). Os sensores, os atuadores e o controlador comunicam-se através do esteriótipo **BasicObject**. Segue a descrição das classes presentes neste diagrama:

- CPBM: contém o algoritmo de Controle Preditivo Baseado em Modelo;
- Reference: classe para geração de trajetórias de referência;
- PurePursuit: contém o algoritmo *pure pursuit* de geração de trajetórias de aproximação;
- Integrator: classe com método de integração numérica para determinar, em tempo real, a posição do robô em coordenadas cartesianas no plano de deslocamento;
- AngleSensor: representa a bússola eletrônica;
- RealPosSensor: representa o sensor de posicionamento real;
- SpeedController: representa o controle de velocidade;
- CrashSensor: representa o sensor anticolisão;
- SystemClock: representa o relógio de tempo real;
- HeadingController: representa ações de controle para o sistema de direção;
- OperatorIO: representa a interface com o operador;
- Driver: classes que representam as características gerais dos *drivers* de *hardware*.

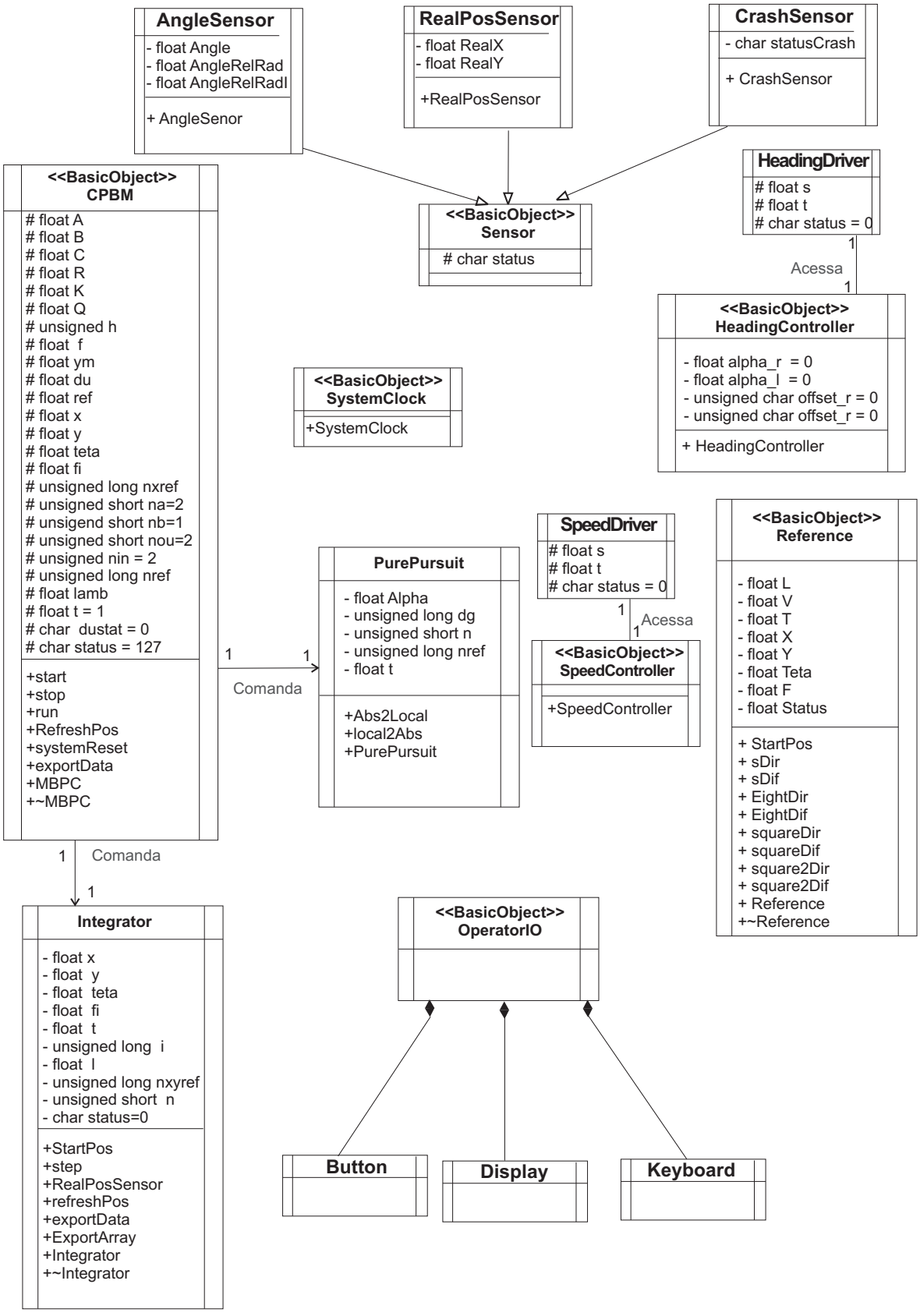

Figura 5.4: Diagrama de Classes do Sistema Modificado

Cada componente da aplicação herda as propriedades da classe base *BasicObject*; por exemplo, o componente OperatorIO pode publicar ou receber eventos através das propriedades herdadas, *publish(event : msg_obj)* e *receive(event: msg_obj)*.

No diagrama da Figura 5.4, as classes CommandProcessor, ControlLawProcessor e MPC, ilustradas no diagrama da Figura 5.2, estão representadas por uma única classe: a classe CPBM, que engloba as três classes de controle. Optou-se por essa representação pelo fato de que as três classes juntas compõem o algoritmo de controle e, desse modo, para não dividi-lo, simplesmente agrupou-se a três classes em uma única, a fim de simplificar a visualização da estrutura de controle.

Outro fator importante que difere os dois diagramas é a forma como é executada a leitura dos sensores do veículo: antes a classe MPC fazia a leitura nos sensores de forma direta, ou seja, acessava as classes Angle, RealPos e Crash. Porém, como o acesso direto não é mais possível por causa do desacoplamento dos objetos e por ser uma comunicação anômima, os sensores Angle e RealPos publicam eventos periodicamente a cada instante de tempo *t*, sendo esses eventos captados pelo componente CPBM. O sensor Crash publica eventos de modo aperiódico quando detecta algum objeto que possa causar uma colisão; esse evento também é captado pelo componente CPBM.

Já a classe Reference, representada no diagrama da Figura 5.2 como parte da classe OperatorIO, modelou-se separadamente, representando um componente externo que informa, através de eventos, o total de pontos da trajetória e os seus respectivos pontos.

As classes Integrator e PurePursuit interagem com o controle da mesma forma como apresentado anteriormente, ou seja, a classe CPBM comanda as ações de controle diretamente, invocando métodos dessas classes. O desacoplamento dessas classes não foi aconselhado por (GOMES, 2006), por se tratar de uma operação crítica do sistema; sendo assim, essa interação foi mantida da mesma forma que estava antes.

Neste novo sistema, a modelagem da interação entre os objetos é realizada através de eventos. As interações entre os objetos são modeladas em UML, através de diagramas de seqüência e diagramas de colaboração. Como não há interação direta entre os componentes do sistema, ou seja, um componente envia um evento mas esse evento não está direcionado a um componente especifico, então a modelagem através de diagramas de seqüência é realizada separamente para cada componente do sistema.

Na Figura 5.5 apresentam-se dois dos diagramas de seqüência do novo sistema. O diagrama

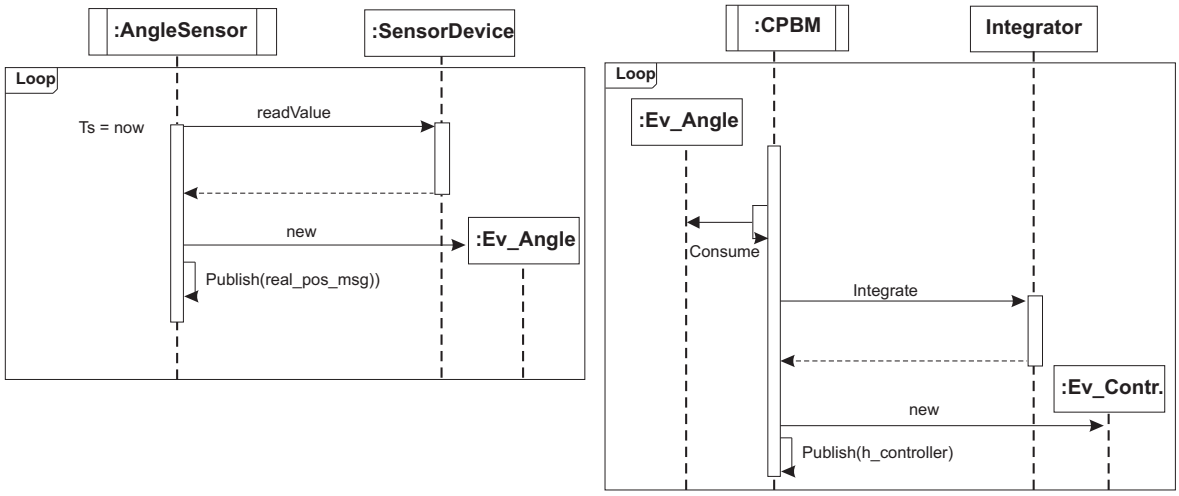

Figura 5.5: Diagramas de Seqüência

se valeu de conceitos apresentados na Figura 3.7, que mostram que cada objeto, quando publica ou recebe eventos na aplicação, tem um comportamento comum para comunicação. Isso inclui os objetos que participam da interação e o *Event Channel Handler*, que se encarrega de distribuir os eventos aos assinantes. Como a comunicação é tratada de forma igual, a modelagem dos comportamentos dos objetos pode se concentrar em suas finalidades.

A Figura 5.5 ilustra dois diagramas: o primeiro mostra o objeto ativo AngleSensor e o objeto SensorDevice, que representa o dispositivo fisico. Primeiro, a partir de um instante de tempo *t*, o objeto AngleSensor lê periodicamente o valor do ângulo de orientação disponibilizado pelo sensor (SensorDevice), para depois enviar um evento com as informações. Nota-se a presença do operador *loop*, o qual informa que as ações nele contidas se repetem até que uma condição seja satisfeita (ou indefinidamente, caso contrário). Neste diagrama as operações se repetem. Já no segundo diagrama, o objeto ativo CPBM recebe um evento em determinado momento, realiza o cálculo da lei de controle, em seguida integra com o objeto Integrator, para somente depois enviar um evento como forma de ação.

5.3 Detalhamento do novo modelo

Uma vez apresentadas as alterações na parte da modelagem do sistema, parte-se para a discussão sobre como é realizada a interação entre os componentes do novo modelo.

A partir da Figura 5.4, pôde-se representar como é o relacionamento entre as classes, utilizando-se para isso, o esteriótipo **BaseObject**; porém o diagrama de classes não ilustra detalhes do modelo P/S.

Os canais, e os respectivos eventos contidos nele, não são detalhados nessa representação.

A Figura 5.6, mostra os componentes do novo sistema e os canais de eventos. Como o sistema troca muitas mensagens e o poder de processamento é um fator limitante, ele foi dividido de forma que ficasse eficiente tornando-se melhor distribuído com relação aos canais de eventos.

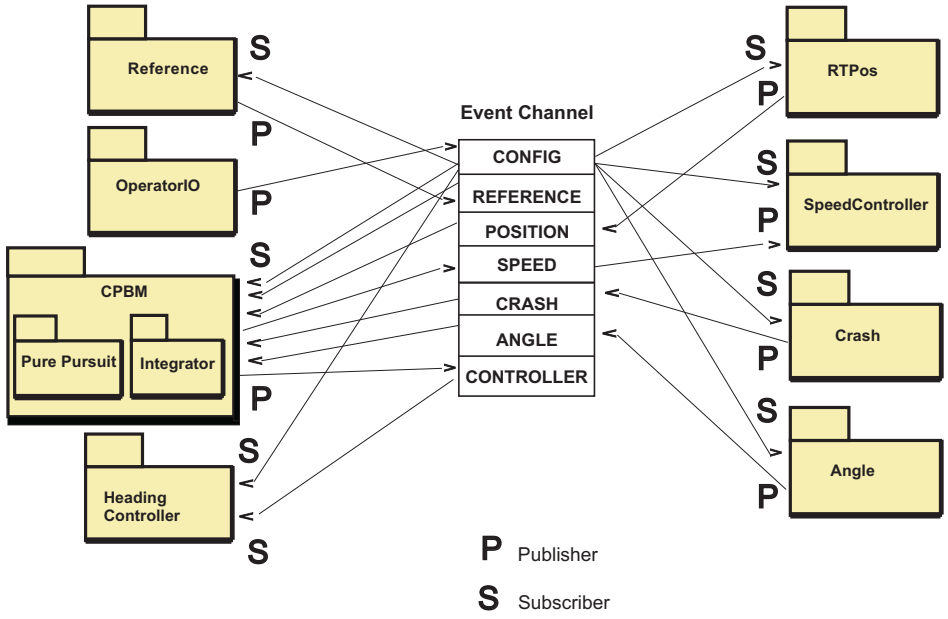

Figura 5.6: *Event Channel*

Procura-se através desse modelo viabilizar a utilização de dispositivos com pouca capacidade de processamento. Quando se aumenta o número de canais de eventos, pode-se estruturar melhor os conteúdos das mensagens, de forma que os consumidores não precisem perder tempo com análise daquelas que não lhes interessa. Isso é feito definindo-se para cada canal um número limitado de eventos, estabelecendo, assim, que todos os canais só disponibilizem determinado conteúdo.

Desse modo, pode-se dizer que o endereçamento fica baseado em eventos e não em canais de eventos; por exemplo, para eventos sobre velocidade, direciona-se para o canal de velocidade, e assim procede-se para os demais canais. A divisão em sete canais de eventos deve-se ao fato de que o componente *Event Channel Handler* endereça as mensagens por canais de eventos. Sendo assim, a divisão facilita a compreensão e torna-se mais eficiente.

5.3.1 Interação entre os Componentes

A descrição desses canais de eventos é mostrada a seguir:

- Canal CONFIG: eventos de configuração do sistema;

- Canal REFERENCE: responsável pelos eventos que contêm os dados da trajetória;
- Canal POSITION: contém dados da posição do veículo, eventos periódicos;
- Canal SPEED: contém dados de controle da velocidade do veículo, eventos periódicos;
- Canal CRASH: contém eventos de um possível obstáculo na pista, eventos aperiódicos;
- Canal ANGLE: contém eventos da bússola eletrônica, eventos periódicos;
- Canal CONTROLLER: contém eventos para o controle de direção, eventos periódicos.

Primeiramente um componente chamado OperatorIO publica no canal CONFIG enviando eventos que serão utilizados por todos os componentes. Os eventos desse canal contêm atributos necessários para inicialização de todos os componentes do veículo. Dessa forma, todos os componentes inscrevem-se no canal CONFIG.

Concluída essa parte do sistema em que todos os componentes receberam os dados necessários de configuração, é a vez do componente Reference publicar no canal REFERENCE, o qual contém eventos referentes aos dados prontos da trajetória, informando o total de pontos e os respectivos pontos da trajetória escolhida. O veículo necessita desses dados para iniciar o percurso da trajetória.

Já o componente CPBM inscreve-se no canal REFERENCE a fim de receber os dados da trajetória e efetuar os cálculos da ação de controle. Uma vez efetuados os cálculos de controle, serão posteriormente disponibilizadas as devidas ações nos canais de controle de direção e controle de velocidade.

Além de inscrever-se no canal REFERENCE, o componente CPBM precisa assinar o canal CRASH para obter dados sobre objetos na pista. O responsável por gerar esses eventos é o componente Crash, que envia um evento no canal CRASH sempre que detecta um obstáculo que possa estar na respectiva trajetória do veículo, evitando uma possível colisão deste com o obstáculo.

Outra assinatura necessária que o componente CPBM faz é no canal POSITION. O componente RTPos disponibiliza nesse canal eventos pertinentes ao controlador; assim, o componente CPBM receberá eventos desse canal, já que este trata de questões relacionadas ao controle.

O canal ANGLE é alimentado pelo componente Angle que disponibiliza eventos referentes à bússola eletrônica. O controlador necessita dos dados da bússola, e por isso torna-se um assinante do canal ANGLE.

Com relação aos canais SPEED e CONTROLLER, nota-se que seus assinantes são respectivamente SpeedController e HeadingController. Entretando, percebe-se também que esses componentes

Tabela 5.1: Lista de Eventos

Eventos	Descrição	Publisher	Subscriber	Processo
e_detect_ obstacle	Detecta obstáculo	Crash	CPBM	ProcessControlLaw
e_speed	Controla velocidade	CPBM	SpeedController	SetSpeed
e_error	Detecta erro nas coordenadas x e y	RTPos	CPBM	ProcessControlLaw
e_angle	Detecta ângulo	Angle	CPBM	ProcessControlLaw
e_position	Ajustar direção	CPBM	HeadingController	setPostion
e_begin	Início do percurso	Operator	CPBM, Crash, Speed, RTPos, Reference e HeadingController	set e run
e_stop	Parar	Operator	CPBM	stop
e_adjust_ StopTrack	Ajusta velocidade	Operator	CPBM	ProcessControlLaw
e_number_ points	Número de Pontos da trajetória	Reference	CPBM	setRef
e_points_ reference	Pontos da trajetória	Reference	CPBM	setRef
e_adjust_ lambda	Ajusta lambda	Operator	CPBM	setLamda

não são publicadores de nenhum evento em qualquer um dos sete canais presentes no sistema, ou seja, são apenas assinantes ou consumidores de informação. O componente SpeedController é responsável pelo controle de velocidade do veículo. Os eventos que informam ou alimentam o canal SPEED são publicados pelo componente MBPC, que também é publicador de eventos do canal CONTROLLER, o qual contém como assinante o componente HeadingController, responsável pelo controle de direção.

O componente CPBM, começa a efetuar o algoritmo de controle depois de receber eventos dos sensores do veículo e também do componente Reference.

Depois que o veículo sai da posição inicial, o CPBM fica em *loop* capturando eventos publicados pelos sensores nos canais em que ele se subscreveu. Além disso, gera eventos nos canais em que é o publicador de eventos, para que os atuadores possam efetuar as ações de controle. Esse laço só termina quando o veículo chega ao ponto final da trajetória.

A Tabela 5.1 descreve os eventos que circulam no sistema, juntamente com seus respectivos *Publishers* e *Subscribers*.

5.4 Implementação

Neste estudo de caso os componentes são divididos em quatro grupos:

- Os Sensores, que geram eventos informando as condições atuais do trajeto como posição, ân-

gulo e obstáculo. Esses eventos são usados pelo CPBM para tomar as devidas ações, como acelerar ou até mesmo parar ou desviar o veículo da trajetória por causa de algum obstáculo.

- A Referência, que contém a base de dados utilizada pelo CPBM e que projeta os pontos da trajetória, caminho real que o veículo deverá percorrer.

- O Controlador, que é baseado no algoritmo de controle preditivo para controlar a trajetória de deslocamento de veículos, seguindo uma trajetória de referência. Necessita dos eventos gerados pelos sensores (RTPos, Crash, Angle), sistema de geração de trajetórias de aproximção e integrador das coordenadas do deslocamento.

- Os Atuadores. Quando o controle recebe eventos dos sensores informando dados como *Crash* e Ângulo de Orientação, deverá realizar o cálculo do algoritmo de controle preditivo utilizando trajetórias de aproximação e integrador das coordenadas do deslocamento, gerando, dessa forma, eventos de controle que deverão ser capturados pelos atuadores (posicionamento das rodas dianteiras, ângulo, freio e velocidade). Esse *loop* é executado até o fim do percurso.

5.4.1 Sensores

Essa parte do sistema inclui os sensores que captam eventos do ambiente, como desvio da trajetória ou obstáculo e publicam no respectivo canal, para que o componente CPBM possa fazer os ajustes necessários de controle.

O componente RealPosSensor fornece a posição real do veículo no plano xy para fins de correção do erro de integração; o componente AngleSensor é a bússola eletrônica; e o componente CrashSensor é o sensor de colisão. Esses componentes publicam eventos que são utilizados pelo controlador CPBM.

Uma parte do código do componente AngleSensor é listado em Código 5.1; nesse trecho, primeiramente o componente sensor faz a assinatura no canal de configuração (CONFIG), linha um, para depois entrar em *loop*, linha quatro até linha dez, esperando mensagens do mesmo canal e também publicando mensagens no canal ANGLE através de interrupções, linha 12 até a linha 23.

Código 5.1: Sensor Angle

```
1  if(!subscribe(CONFIG)) //api cosmic, assinatura − canal CONFIG
2          exit(1);
3
4  while(!done){ ////aguardando mensagens
5    if(get_msg(&m)){ //api cosmic, chegada de mensagens
6      switch (m.subject){ //api cosmic, identificador do canal
7                  case "CONFIG":{
8                          setitimer (ITIMER_REAL, &interval, NULL);
9                          signal(SIGALRM, sigalrmHandler);
10                         siginterrupt(SIGALRM, 1);
11
12 void sigalrmHandler (int egal) {
13         real_pos_msg.subject = ANGLE;
14         real_pos_msg.data[0] = 1;   //priori msg
15         real_pos_msg.len = 16;   //tamanho msg
16         AngleSensor ∗angle_Sensor = new AngleSensor(); //interface
17         pos = angle_Sensor−>getAngleRelRad(); //angle rad
18         pos = angle_Sensor−>getAngleRelRadI(); //angle radI
19         real_pos_msg.data[1..len] = pos_sensor.data[data]; //dados
20         if (publish(&real_pos_msg))
21                 printf("\nEnviou msg p/ canal ANGLE\n");
22         else
23                 printf("\n Erro ao fazer publish no canal ANGLE\n"); }
```

5.4.2 Controlador

O componente CPBM é o algoritmo de controle geral que pode ser reaproveitado para controlar outros sistemas. Como o sistema a ser controlado é um veículo, foram utilizados dois componentes auxiliares PurePursuit e Integrator.

O Código 5.2 ilustra alguns trechos do controlador, o qual, inicialmente na linha 1 faz *subscribe* no canal de configuração *(subscribe(CONFIG))*, linha 1; só posteriormente ele poderá receber as mensagens desse canal, linha 7, e, uma vez tendo recebido essa mensagem, estará apto a fazer *subscribe* no outros canais, linha 9 até a linha 12, a fim de receber eventos para o cálculo da ação de controle.

Quando o controlador recebe mensagem do canal CONFIG, linha 7, ele faz *subscribe* nos outros

canais, REFERENCE, POSITION, CRASH e ANGLE, e gera ações de controle publicando eventos *(publish(&h_controller))* no canal CONTROLLER, linha 46, e no canal SPEED.

Código 5.2: Controle

```
1  if(!subscribe(CONFIG))
2      exit(1);
3
4  while(!done) {
5   if(get_msg(&m)) {
6     switch (m.subject) {
7        case "CONFIG": {
8                  //assinatura − canais −> REFERENCE, POSITION, CRASH, ANGLE
9                  subscribe(REFERENCE);
10                 subscribe(POSITION);
11                 subscribe(CRASH);
12                 subscribe(ANGLE);
13                 setitimer (ITIMER_REAL, &interval, NULL);
14                 signal(SIGALRM, sigalrmHandler);
15                 siginterrupt(SIGALRM, 1);}
16                 ..
17       case "REFERENCE":{
18                 if (m.data[1] == 1) //msg_id, número de pontos
19                 // allocates memory, xref[], yref[], tetaref[] and firef[]
20                 if already knows the amount of points {
21                       xref[position] = m.data[data];
22                       yref[position] = m.data[data];
23                       tetaref[position] = m.data[data];
24                       firef[position] = m.data[data];}
25                 if was received whole the coordinates of the trajectory {
26                       //creates the controller
27                       MBPC ∗meuControlador = new MBPC(T, 20);
28                       //adjust the controller
29                       meuControlador−>setRef(0, total_points, 'm');
30                       ..
31         case "POSITION":{
32                 pos_x = m.data[position_x];
33                 pos_y = m.data[position_y];
34                 ..
```

```
35          case "CRASH":{
36                      crash = m.data[crash];
37                      ..
38          case "ANGLE":{
39                      angle = m.data[angle];
40                      ..
41  void sigalrmHandler (int egal) {
42    h_controller.subject = CONTROLLER; //canal Heading controller
43    h_controller.len = length;
44    h_controller.data[data_speed] = speed_v;
45    h_controller.data[data_angle] = angle;
46    if (publish(&h_controller))
47      printf("published msg in the channel 0x%llx",h_controller.subject); }
```

5.4.3 Referência

O componente Referência, entendido como um componente de um banco de dados, refere-se à trajetória que o veículo terá que realizar. Quando o veículo for percorrer determinado trajeto, esse componente deverá fazer assinatura do canal e pegar os pontos da trajetória escolhida. O componente sempre dará como resposta o número de pontos da trajetória e os respectivos pontos. Então, desse modo, pode ser dividido em duas partes. A primeira, responsável por disponibilizar o número total de pontos da trajetória, é importante para o algoritmo de controle, já que há a necessidade de alocação de memória por parte desse algoritmo, que terá que inscrever-se no canal REFERENCE. Na segunda parte, os respectivos pontos serão disponibilizados; então, será feito um *publish* pelo componente Referência nesse canal, onde o CPBM, pelo fato de utilizar esses dados, terá que fazer subscribe. A ordem é importante, pois o CPBM precisa primeiro alocar memória, para depois utilizar os pontos; desse modo, ele capta do canal o evento que contém o total de pontos, para depois captar outro evento que contenha os pontos reais.

O Código 5.3 mostra detalhes do componente Referência, o qual primeiro faz subscribe no canal CONFIG (subscribe(CONFIG), primeira linha, depois aguarda mensagem desse canal, a fim de poder publicar um evento com o número total de pontos da trajetória (send_reference_data[number_points]) no canal REFERENCE. Por fim, esse componente envia os pontos da trajetória xref, yref, tetaref e firef, linha 21.

Código 5.3: Referência

```
1  if(!subscribe(CONFIG)) //api cosmic, assinatura − canal CONFIG
2      exit(1);
3  while(!done) {
4   if(get_msg(&m)) {
5    switch (m.subject) {
6     case "CONFIG": {
7                    //define the reference
8                    Reference ∗minhasReferencias = new Reference(LE, V, T);
9                    //initialization of the reference
10                   minhasReferencias−>setStartPos(1, 1, 0, 0);
11                   //trajectory
12                   nref = minhasReferencias−>square2Dir(xref, yref, tetaref, firef);
13                   send_reference.data[number_points] = nref; //número de pontos
14                   for(i=0;i<total_points;i++) {
15                           send_reference.data[x] = xref[i];
16                           send_reference.data[y] = yref[i];
17                           send_reference.data[teta] = tetaref[i];
18                           send_reference.data[fi] = firef[i];
19                           send__reference.subject = REFERENCE;
20                           if (publish(&send__reference))
21                                   printf("published msg in the channel 0x%llx",h_controller.subject);
22                   ..
```

5.4.4 Atuadores

Os componentes definidos aqui como atuadores, HeadingController e SpeedController, executam as ações determinadas pelo cálculo da lei de controle do controlador.

O componente *HeadingController* captura eventos publicados pelo CPBM no canal CONTROLLER, fazendo ajustes das rodas do veículo e atuando na barra de direção. Diferentemente dos outros componentes, este não gera eventos. Todavia, pelo fato de capturar e executar as ações de controle, ele pode provocar geração de eventos em outros componentes como sensores.

O Código 5.4 ilustra aspectos desse componente; primeiro subscreve no canal CONFIG *(subscribe(CONFIG))*, linha 3, depois fica em *loop*, linha 5, aguardando as ações de controle geradas pelo controlador. Quando recebe alguma mensagem no canal CONTROLLER, linha 13 canal de controle,

o componente atuará sobre a velocidade do veículo e posicionamento das rodas.

Código 5.4: Atuadores

```
signal(SIGINT,siginthandler);
siginterrupt(SIGINT, 1);
if(!subscribe(CONFIG)) //api cosmic, assinatura − canal CONFIG
   exit(1);
while(!done) {
   if(get_msg(&m)) {
          switch (m.subject) {
            case "CONFIG": {
               printf("event msg from channel ", m.subject);
                        if(!subscribe(CONTROLLER))
                           exit(1);
               ..
            case "CONTROLLER":{
               printf("\nevent msg from channel \n", m.subject);
                    speed = m.data[speed];
               position = m.data[position];
               //control action
               HeadingController ∗heading = new HeadingController();
               //adjust the speed and position
               heading−>setPosition(speed);
               heading−>setSpeed(position);
               ..
```

5.5 Experimentos Realizados

Nesta seção é apresentado o comportamento do sistema embarcado distribuído de controle preditivo baseado em modelo. Para avaliar o comportamento do veículo durante o trajeto, realizaram-se simulações considerando o novo modelo com componentes distribuídos e autônomos interagindo através de eventos. Na Figura 5.7 apresenta-se o gráfico de uma das trajetórias. Trata-se de um percurso em *S*, onde o ponto inicial da referência (linha contínua) está localizado na coordenada (1,1) do plano (x,y); as curvas foram projetadas para que o veículo faça-a com um raio de dez metros; as retas têm comprimento igual a 20 metros e o número de *S* é igual a três. Nota-se também que o veículo inicia

o percurso a partir das coordenadas (-1,-5) (linha tracejada). Essa posição difere-se do ponto inicial da referência, mostrando que o veículo não precisar estar necessariamente no mesmo ponto inicial da referência.

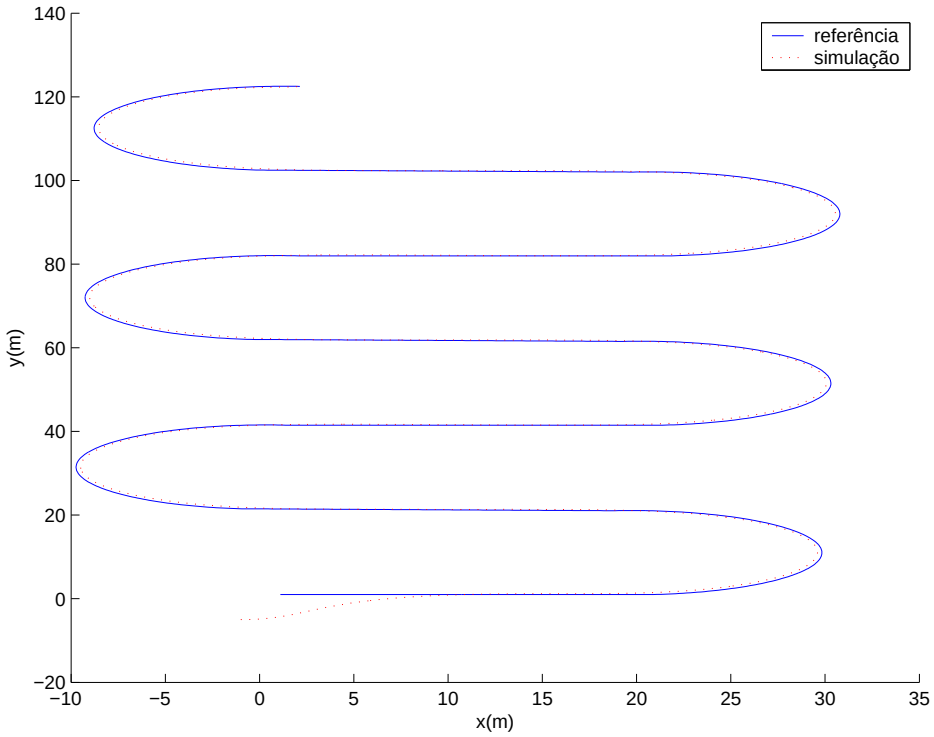

Figura 5.7: Trajetória *xy* do veículo para uma referência em *S*

O gráfico da Figura 5.7 mostra que o desempenho do veículo durante o percurso se manteve em conformidade com a trejatória de referência, mesmo com os possíveis atrasos de comunicação (ex: latência da rede). Nos experimentos realizados assumiu-se que os protocolos de comunicação oferecidos pela infra-estrutura subjacente são confiáveis.

5.6 Considerações Finais

O sistema apresentado por (GOMES, 2006) está atrelado a uma configuração específica de *hardware*, seja ela x86, DSP ou PowerPC e, além disso, utiliza-se como meio de comunicação um único protocolo de rede. Essa utilização de um único protocolo de rede e de apenas uma plataforma de *hardware* não condiz com a realidade dos veículos atuais, os quais possuem geralmente mais de um tipo de rede e/ou sub-redes, e os dispositivos físicos não estão interligados diretamente.

A modelagem apresentada por (GOMES, 2006) mostra detalhes da interação entre os objetos; essa

comunicação é realizada através de chamada de métodos. Porém, nota-se no novo modelo o desacoplamento entre os componentes que se comunicam, a disponibilidade de alterar o sistema em tempo de execução, e a facilidade de incorporar novos componentes.

Do ponto de vista do desenvolvedor, os dois sistemas diferem muito. No sistema desenvolvido por (GOMES, 2006), o desenvolvedor tem que conhecer toda a estrutura do sistema, já que este opera através de chamadas de métodos. No novo modelo pode-se desenvolver componentes de modo separado por desenvolvedores que não conhecem todo o sistema, mas que tenham informações necessárias sobre cada parte em separado. A única informação necessária sobre o sistema refere-se aos canais e aos respectivos eventos, já que são a base do novo sistema.

5.7 Conclusões

Neste capítulo, foi apresentada a reestruturação de um sistema de controle que operava de forma centralizada para veículos autônomos. Na nova proposta, a distribuição dos componentes representou de forma fiel a comunicação presente atualmente nos automóveis, utilizando rede CAN com objetos distribuídos.

O desacoplamento entre os componentes da rede CAN deu-se através do modelo *Publisher/Subscriber*. Além disso, pôde-se realizar a comunicação entre uma rede CAN e uma rede TCP/IP, ou seja, o protocolo foi abstraído; desse modo, os componentes presentes neste estudo de caso podem estar em uma rede CAN, TPC ou nas duas. Dessa forma foi possível gerar um novo sistema que interage através de eventos.

Com relação à modelagem, pôde-se representar a nova estrutura dos componentes do veículo autônomo. Os diagramas puderam apresentar como os componentes estão distribuídos: o diagrama de componentes e deployment, modelando desde a comunicação interna até a comunicação entre os nodos presentes em diferentes plataformas de *hardware*, demonstrando como interagem; diagrama de seqüência e diagrama de caso de uso; e o diagrama de classes, representando como as classes estão dispostas.

Capítulo 6

Conclusões e Trabalhos Futuros

Nesta dissertação dedicou-se ao estudo, o desenvolvimento e a implementação de uma arquitetura de comunicação para sistemas embarcados distribuídos baseada no protocolo *Publisher/Subscriber* denominada DOCAS (*Distributed Object-based architeture for Controlling Autonomous vehicleS*). Ela serve para interligar, através de eventos, os diversos componentes distribuídos que comumente estão presentes em diferentes tipos de redes. Na implementação atual, desenvolveu-se suporte aos protocolos CAN e TCP/IP.

Os objetivos pretendidos com a utilização da arquitetura proposta, motivada pelo fato de que torna-se possível programar, de forma simplificada, a comunicação entre os elementos envolvidos na aplicação, foram atingidos. Neste trabalho desenvolveram-se também alternativas para representar o problema de comunição num nível de abstração mais elevado.

Na etapa de modelagem da arquitetura proposta, definiu-se a utilização do esteriótipo *BasicObject* como forma de definir um conjunto de propriedades que são utilizadas por todas as aplicações que contêm essa especificação, ou seja, apresentou-se o *middleware* como uma interface provida pela classe ECH. Sendo assim, todos os componentes de software interagem através dessa interface publicando ou recebendo eventos. Com isso, pôde-se abstrair a parte de comunicação da arquitetura.

Na etapa de implementação, utilizaram-se três plataformas de *hardware* para validar a arquitetura proposta. As configurações e alterações implementadas mostraram que é possível implementar a arquitetura DOCAS em diferentes plataformas de *hardware*. Os ajustes na rede CAN basearam-se no suporte provido pelas plataformas. Essas alterações não causaram impacto no *middleware*, ou seja, pôde-se utilizar os recursos sem alterar a sua semântica.

No estudo de caso realizado puderam ser analisados os benefícios de se utilizar o modelo *Publisher/Subscriber*. Os resultados experimentais, obtidos com a aplicação do sistema de controle preditivo baseado em modelo de veículos autônomos através de uma estrutura descentralizada e interagindo por meio de eventos de forma desacoplada, comprovaram que se pode aplicar essa arquitetura para esse tipo de sistema. A utilização do modelo *Publisher/Subscriber* facilitou o desenvolvimento dos componentes de software da aplicação, já que não há ligação direta entre os componentes e, desse modo, não se necessita por parte do programador, de conhecimento de todo o sistema que se pretende desenvolver. Outro fator importante é a utilização, neste estudo de caso, de mais de um protocolo de rede e mais de uma plataforma de *hardware*; ou seja, isso representa uma evolução presente cada vez mais nos veículos atuais.

Como sugestões para estudos futuros, pode-se adicionar o suporte a diferentes protocolos, como os já citados neste trabalho, o LIN e o FlexRay. Outro tipo interessante de rede é a 802.11, que poderia ser utilizada, por exemplo, como no estudo de caso aqui apresentado, porém utilizando sensores sem fio para informar as condições de tráfego de uma rodovia.

Apêndice A

Instalação e Configuração do Kit M5484

Devido ao grande número de ajustes realizados tanto no hardware quanto no software e as dificuldades encontradas na instalação do sistema operacional embarcado no *kit* de desenvolvimento da Coldfire modelo M5484EVB realiza-se neste apêndice uma descrição da documentação de suporte como forma de auxilio aos possíveis usuários que venham a utilizar esse tipo de hardware e software.

A.1 Introdução

Possíveis passos para a implantação da distribuição linux LTIB no *kit* de desenvolvimento Freescale M5484Lite utilizando como *host* um PC com a distribuição linux Ubuntu[1]:

- Instalação e configuração do programa de comunicação serial Minicom no host;
- Instalação e configuração do servidor tftp no host;
- Instalação do servidor nfs no host;
- Instalação de ferramentas de configuração do LTIB no host;
- Configuração e compilação do LTIB;
- Configuração do software DBug para utilizar a comunicação *ethernet* da placa;
- Baixando a partir do DBug a imagem do Colilo (*bootloader*) para a RAM da placa e executando o Colilo;

[1] http://www.ubuntu.com/

- Baixando a partir do Colilo a imagem do kernel linux para a placa e executando o linux.

Objetivos: copiar, a partir do software dBUG presente no *kit* via *ethernet* e protocolo tftp, a imagem de um *bootloader* (no caso o Colilo) para a RAM da placa. A partir do Colilo, transferir via *ethernet* e protocolo tftp a imagem do *kernel* linux para a RAM da placa, e no final passar os parâmetros adequados para o kernel de tal forma que ele carregue, usando como raiz da sua árvore de diretórios, um diretório presente no PC *host* (usando no *host* um servidor nfs para isso).

A.2 Instalação e Configuração do Programa de Comunicação Serial Minicom no Host

Utilizando a ferramenta Synaptic, Figura A.1, pode-se instalar o Minicom no Ubuntu.

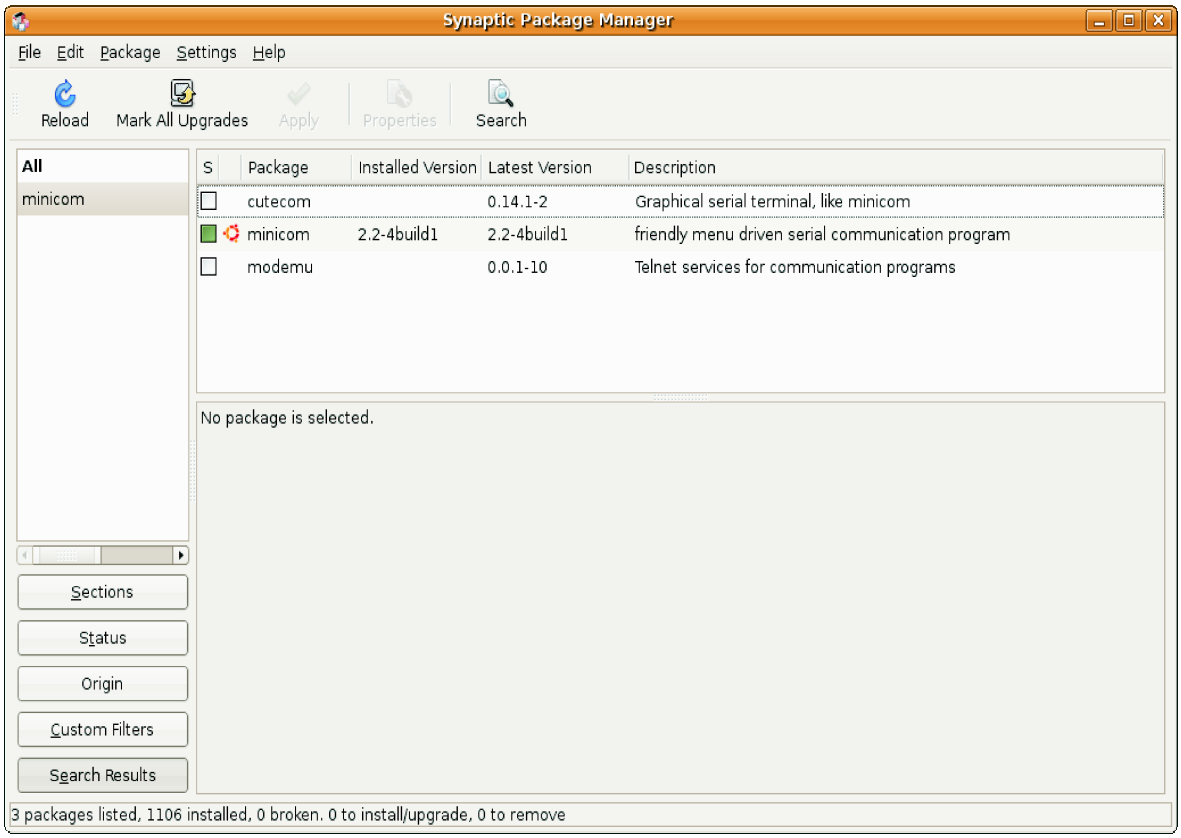

Figura A.1: Synaptic - instalar o aplicativo Minicom no Ubuntu.

Depois da instalação do aplicativo Minicom, inicia-se a configuração do software.

```
 ronaldo@lab:~$ dmesg | grep tty
[ 27.766988] serial8250: ttyS0 at I/O 0x3f8 (irq = 4) is a 16550A
```

```
[ 27.767236] serial8250: ttyS1 at I/O 0x2f8 (irq = 3) is a 16550A
[ 27.768031] 00:07: ttyS0 at I/O 0x3f8 (irq = 4) is a 16550A
[ 27.768401] 00:08: ttyS1 at I/O 0x2f8 (irq = 3) is a 16550A
```

A porta serial do host está em /dev/ttyS0.

Verificar permissões da porta serial:

```
ronaldo@lab:~$ ls /dev/ -l |grep ttyS 0
crw-rw---- 1 root dialout 4, 64 2007-09-12 12:16 ttyS0
ronaldo@lab:~$ groups ronaldo
ronaldo : ronaldo adm dialout cdrom floppy audio dip video plugdev scanner
netdev lpadmin powerdev admin
```

No minicom:

- Utilizar o comando Ctrl-A Z para entrar no menu;
- Utilizar a tecla O para entrar em configure minicom;
- Em Serial port setup, modificar o serial device para /dev/ttyS0;
- Ainda em serial port setup, retirar o software e *hardware flow control*, para comunicar-se com a placa e modificar o *baud rate* (*default* da placa é 19200), *data bits* (8), *parity bits* (*none*) e *stop bit* (*none*);
- Configurar também em serial port setup, no menu Modem & Dialing; retirar o texto para Init String, Reset String e Hang-up String.

Ligando-se ou reiniciando-se a placa, deve-se observar no minicom o seguinte prompt:

```
ColdFire MCF548X on the M5484LITE
Firmware v4a.1a.1d (Built on Dec 6 2004 11:53:07)
Copyright 1995-2004 Freescale Semiconductor, Inc.
Enter 'help' for help.
dBUG>
```

A.3 Instalação e Configuração do Servidor tftp no Host

Pode-se usar também o software Synaptic, Figura A.2 para instalar o servidor tftp-hpa e o seu respectivo cliente no *host*.

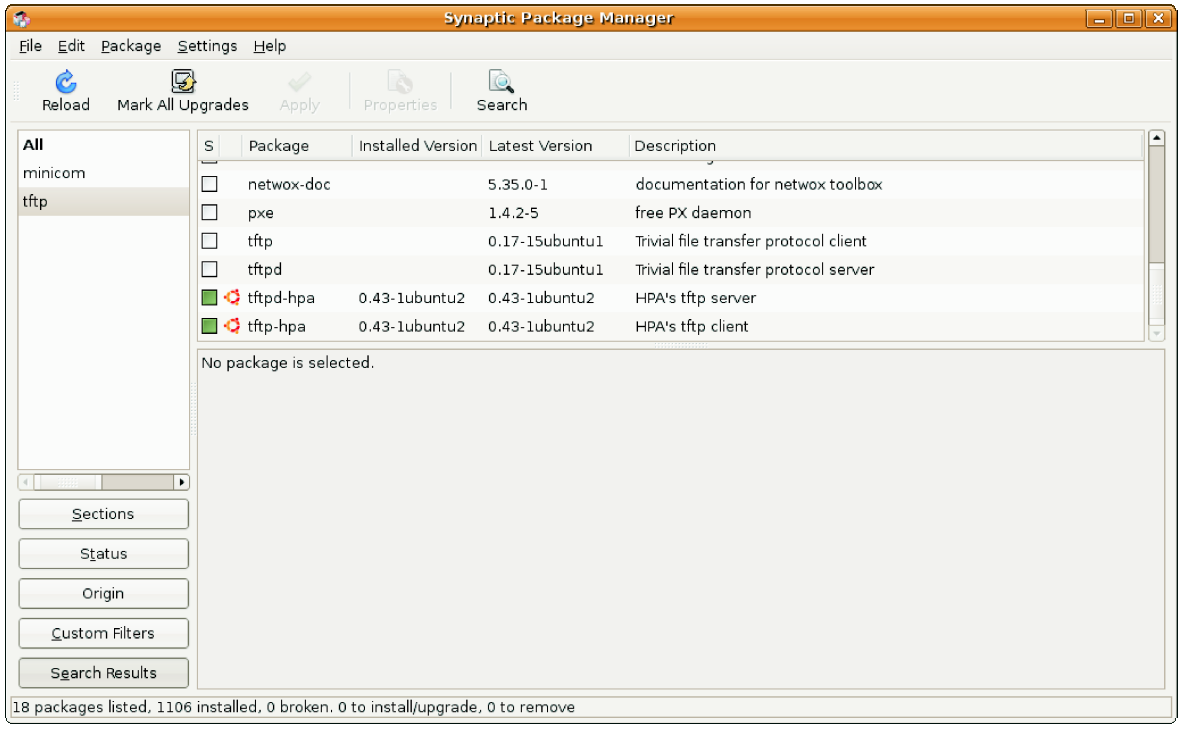

Figura A.2: Synaptic - instalar o servidor tftp-hpa.

No ubuntu, depois de instalar o tftp-hpa necessita-se configurar o arquivo /etc/default/tftpd-hpa modificando o parâmetro de RUN_DAEMON de *no* para *yes*.

Como root pode-se reiniciar o servidor tftp:

```
root@lab:/home/ronaldo# /etc/init.d/tftpdhpa
restart
Restarting HPA's tftpd: in.tftpd.
root@lab:/home/ronaldo#
```

Para verificar se o servidor esta rodando:

```
ronaldo@lab:~$ netstat -a | grep tftp
udp 0 0 *:tftp *:*
```

Usar o cliente tftp para testar o servidor tftp. O diretório usado pelo servidor tftp para encontrar arquivos é o /var/lib/tftpboot.

```
ronaldo@lab:~/testetftp$ ls /var/lib/tftpboot/
notas
ronaldo@lab:~/testetftp$ ls
ronaldo@lab:~/testetftp$ tftp
(to) localhost
```

```
tftp> get notas
tftp> quit
ronaldo@lab:~\/testetftp$ ls
notas
```

A.4 Instalação do Servidor nfs no Host

Instalar o pacote nfs-kernel-server a partir do Synaptic.

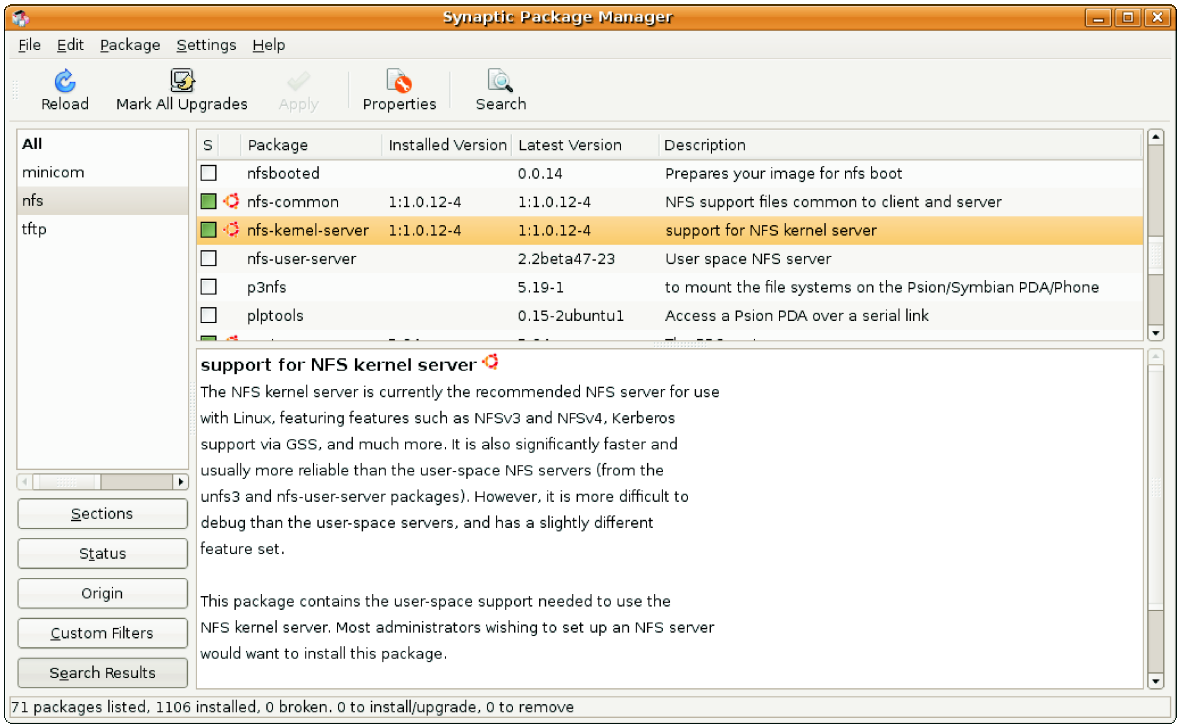

Figura A.3: Synaptic - instalar o pacote nfs-kernel-server.

Modificar o arquivo /etc/exports adicionando a seguinte linha:

```
</home/ronaldo/ltib/rootfs> *(rw,no_root_squash)
```

onde </home/ronaldo/ltib/rootfs> é o diretório no qual se encontra a raiz da árvore de diretórios da distribuição LTIB.

A.5 Instalação das Ferramentas de Configuração do LTIB no *Host*

No *site* da freescale[2], em products > code warrior development tools > downloads > linux board support packages > BSPs for coldfire architectures, pode-se encontrar o Linux BSP for Freescale MCF5474/84 LITE & MCF5475/85 EVB.

Esse pacote vem no formato de um arquivo *iso*, e para montá-lo utiliza-se um comando do tipo:

```
root@lab:/home/ronaldo/plaquinha# ls -l
dr-xr-xr-x 5 root root 2048 2007-01-07 12:22 ltib
-rw-r--r-- 1 ronaldo ronaldo 660414464 2007-08-27 14:03 mcf547x_8x-20070107-ltib.iso
root@lab:/home/ronaldo/plaquinha# mount -o loop mcf547x_8x-20070107-ltib.iso ltib
root@lab:/home/ronaldo/plaquinha# ls ltib/
EULA Help install pkgs Release_Notes.txt
Freescale_Color_Web_Logo.jpg images ltib.tar.gz RELEASE_INFO START_HERE.htm
root@lab:/home/ronaldo/plaquinha#
```

Pode-se utilizar o programa *install* listado acima para instalar o LTIB no host.

```
ronaldo@lab:~/plaquinha/ltib$ ./install
```

Esse programa irá perguntar em qual diretório deve ser instalada a distribuição LTIB no *host*. Nesse caso foi utilizado o diretório ltib.

A.6 Configuração e Compilação do LTIB

Para compilar o kernel deve-se mudar o *link* simbólico do Ubuntu, que originalmente aponta para /bin/bash:

```
ronaldo@lab:~$ ls -l /bin/sh
lrwxrwxrwx 1 root root 9 2007-09-06 16:37 /bin/sh -> /bin/bash
```

Dentro do diretório ltib, com o commando ./ltib –configure chega-se a um menu, Figura A.4 para configuração da distribuição:

```
ronaldo@lab:~/ltib$ ./ltib --configure
```

Em *Package list* pode-se escolher os pacotes a serem instalados na distribuição. Selecionando-se *configure the kernel*, pode-se configurar o kernel a ser compilado após configuração geral do ltib.

[2]http://www.freescale.com

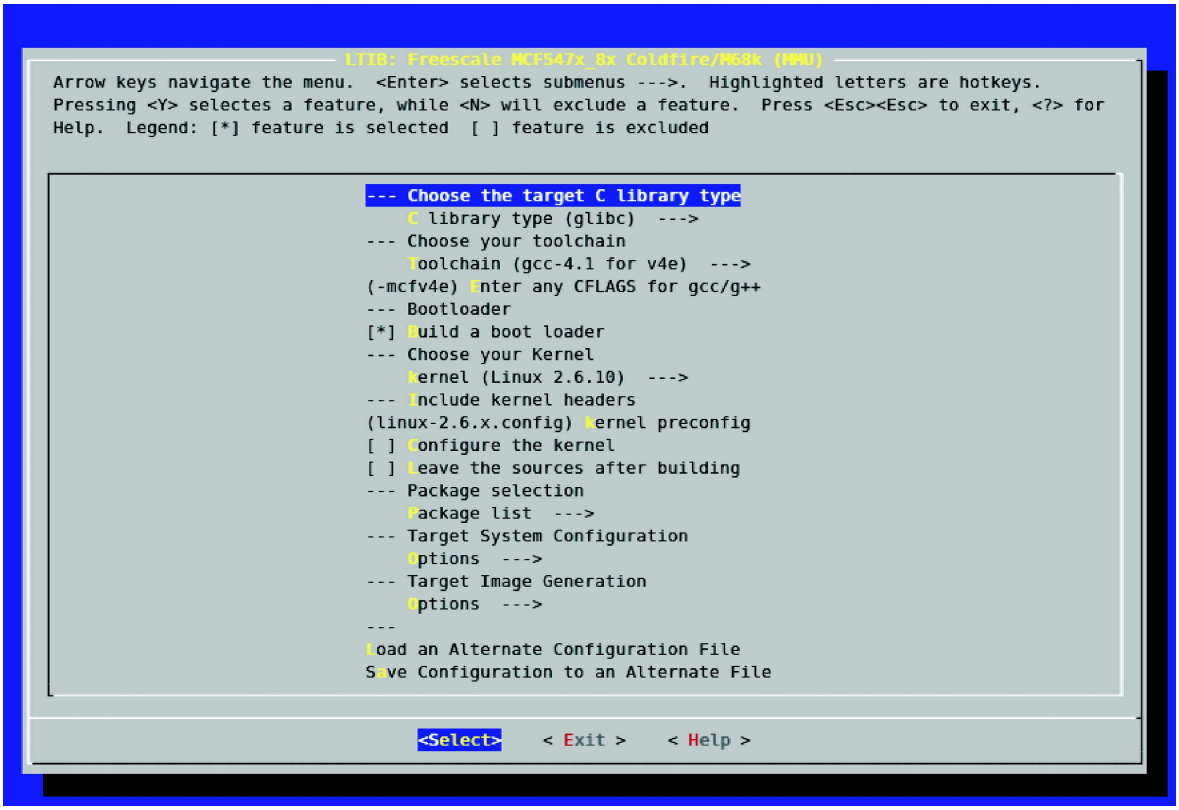

Figura A.4: Menu LTIB.

Para configurar o kernel, seleciona-se *platform dependent setup -> processor type* MCF5485AFE para o caso do kit M5484LiteKite.

Depois de concluir a configuração do kernel, a distribuição está pronta e pode-se utilizar o kernel compilado. A imagem do kernel e o bootloader Colilo são transferidos para o diretório ltib/rootfs/-boot:

```
ronaldo@lab:~/ltib$ ls rootfs/boot/
bootable_kernel colilo_mcf5485_flash.srec System.map
colilo_mcf5475_flash.srec colilo_mcf5485_only.bin vmlinux
colilo_mcf5475_only.bin colilo_mcf5485.srec vmlinux.bin
colilo_mcf5475.srec linux.config
```

Deve-se copiar os arquivos colilo_mcf5485.srec e vmlinux.bin para o diretório /var/lib/tftpboot para que possam ser copiados via protocolo tftp para a placa.

A.7 Configuração do Software DBug para utilizar a Comunicação *Ethernet* da placa

No prompt do dBUG:

```
dBUG> set filename colilo_mcf5485.srec
dBUG> show
base: 16
baud: 19200
server: 192.168.1.100
client: 192.168.1.4
gateway: 0.0.0.0
netmask: 255.255.255.0
filename: colilo_mcf5485.srec
filetype: S-Record
ethaddr: 00:CF:54:85:CF:01
```

O comando *show* lista as configurações para comunicação via *ethernet* do dBUG e com o comando *set* é possível modificar as configurações do dBUG. Nesse caso, o *host* está configurado com o IP 192.168.1.100:

```
root@lab:/home/ronaldo/ltib# ifconfig eth0
eth0 Link encap:Ethernet HWaddr 00:01:02:BD:FC:BF
inet addr:192.168.1.100 Bcast:192.168.1.255 Mask:255.255.255.0
inet6 addr: fe80::201:2ff:febd:fcbf/64 Scope:Link
UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:65 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:0 (0.0 b) TX bytes:11388 (11.1 KiB)
Interrupt:18 Base address:0x4000
```

A.8 Baixando a partir do DBug a imagem do Colilo (bootloader) para a RAM da placa e executando o Colilo

Com o comando dn é possível baixar uma imagem via tftp:

```
dBUG> dn
Downloading S-Record 'colilo_mcf5485.srec' from 192.168.1.100
```

```
S-record download successful!
TFTP transfer completed
Read 188124 bytes (368 blocks)
```

E para executar o colilo:

```
dBUG> go 1000400
Code Manufactor detected = F000 (S/B 0x89)
Code Device detected = F000 (S/B 0x8803)
Boot Device detected = 88C5 (S/B 0x88C3)
-- Colilo M547X - M548X Bootloader --
-- 2005 Freescale/2004 Motorola(c) --
colilo>
```

A.9 Baixando a partir do DBug a imagem do Colilo (bootloader) para a RAM da placa e executando o Colilo

No prompt do Colilo:

```
colilo>?
...
Current TCP/IP configuration:
Board MAC: 00:01:02:03:AA:BB
Board IP: 192.168.1.89
Netmask: 255.255.0.0
Board gateway: 192.168.10.1
TFTP server: 192.168.1.88
Image name: linux_tftp.bin
```

Com o comando ? é possível observar as configurações atuais de rede do Colilo, e com o comando *set* é possível modificá-las. Além disso, é necessário modificar o item cl (*command line*) para passar argumentos de linha de comando para o kernel linux e o item kfl (*kernel in flash*) para indicar ao Colilo que o kernel vai ser copiado para a RAM.

```
colilo>set server 192.168.1.100
TFTP Server: '192.168.1.100'
colilo>set netmask 255.255.255.0
Netmask: '255.255.255.0'
colilo>set image vmlinux.bin
```

```
Image: ’vmlinux.bin’
colilo>set kfl 0
colilo>set cl mac0=00:01:02:03:AA:BB noinitrd
root=/dev/nfs rw nfsroot=192.168.1.100:/home/ronaldo/ltib/rootfs
ip=192.168.1.89:192.168.1.100:0.0.0.0:255.255.255.0:ColdFire:eth0:off
mtdparts=phys_mapped_flash:256k(Colilo),2816k(User)
```

Agora é possível copiar o kernel linux para a RAM da placa:

```
colilo>tftp 0x1000
TFTP download:
Download address:1000
Using FEC:0
Board MAC: ’00:01:02:03:AA:BB’
Board IP: ’192.168.1.89’
Board netmask:’255.255.255.0’
Board gateway:’192.168.10.1’
TFTP server:’192.168.1.100’
Image:’vmlinux.bin’
Initializing TCP/IP stack...
Receiving file ’vmlinux.bin’ from ’192.168.1.100’
|
Image size = 2723840 bytes
colilo>
```

E após, isso carregar o kernel, cuja raiz da árvore de diretórios vai estar localizada no *host* por meio do servidor de arquivos nfs:

```
colilo>g 2000
Linux version 2.6.10 (ronaldo@lab)
...
-sh-2.05b#
```

Neste ponto, obtém-se um *prompt* do *shell bash* com uma distribuição completa a ser utilizada no *kit* de desenvolvimento.

Referências Bibliográficas

APPLICATIONS, M. D. *Lin (Local Interconnect Network Solutions)*. [S.l.], 2007.

BECKER, L. B. et al. On evaluating interaction and communication schemes for automation applications based on real-time distributed objects. *In: Int. Symp. on Object-Oriented Real-Time Distributed Comp.*, Magdeburg, Germany, p. 217–224, 2001.

BERWANGER, J.; SCHEDL, A.; TEMPLE, C. Flexray hits the road. *Automotive Design Line*, nov. 2006. Disponível em: <http://www.automotivedesignline.com>. Acesso em: 20 Janeiro 2007.

BIRMAN, K. *Building Secure and Reliable Network Applications*. [S.l.]: Manning Publications, 1996.

BIRRELL, A.; NELSON, B. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, ACM Press, New York, NY, USA, v. 2, n. 1, p. 39–59, 1984.

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. *The Unified Modeling Language User Guide*. [S.l.]: Addison Wesley, 1998.

BOSCH, R. *Can specification version 2.0*. [S.l.], September 1991. Technical Report.

BRANCO, H. J. L. *A Informática na Indústria Automóvel*. Dissertação (Mestrado) — Departamento de Engenharia Informática - Instituto Superior de Engenharia do Porto, Porto - Portugal, Janeiro 2006.

BRUDNA, C. *Publisher/Subscriber Implementation for TCP/IP and CN-Bus (pubsub-1.4)*. [S.l.], 12th November 2003.

BRUDNA, C. et al. *Preliminary Definition of CORTEX System Architecture*. [S.l.], July 2003.

CASIMIRO, A.; KAISER, J.; VERISSIMO, P. An architectural framework and a middleware for cooperating smart components. In: *CF '04: Proceedings of the 1st Conference on Computing frontiers*. New York, NY, USA: ACM Press, 2004. p. 28–39. ISBN 1-58113-741-9.

CIA. *CAN in Automation*. [S.l.], 2006. Disponível em: <http://www.can-cia.org>. Acesso em: 20 Novembro 2006.

COMPUTERTECHNIK, T. *TTP/C Protocol*. Vienna, 1999.

COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T. *Distributed Systems: Concepts and Design Edition 3*. [S.l.]: Addison Wesley, 2001.

DEITEL, H. M.; DEITEL, P. J. *Java How to Program 4th. ed.* [S.l.]: Prentice Hall, 2002.

DIAS, T. Bmw x5: combinação entre dinâmica, funcionalidade e condução. *Revista Mecânica Online*, set. 2006. Disponível em: <http://www.mecanicaonline.com.br>. Acesso em: 15 Janeiro 2007.

EDWARDS, W. K. *Core Jini, 2nd edition*. [S.l.]: Prentice Hall, 2000.

ENGINEERING, D. S. *Embedded Linux Development Kit (ELDK)*. [S.l.], 2006. Disponível em: <http://www.denx.de>. Acesso em: 10 Outubro 2006.

EUGSTER, P. T. et al. The Many Faces of Publish / Subscribe. *ACM Computing Surveys (CSUR)*, ACM Press, New York, USA, v. 35, n. 2, p. 114 – 131, JUN 2003.

FüHRER, T. et al. *Time triggered communication on CAN*. [S.l.], 2000. Disponível em: <http://www.can-cia.org/can/ttcan/fuehrer.pdf>.

FLEXRAY. *FlexRay Communications System Protocol Specification*. [S.l.], 2005.

GAMMA, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. [S.l.]: Addison-Wesley Professional, 1995.

GOMES, G. K. *Controle Preditivo em Tempo Real para Seguimento de Trajetória de Veículos Autônomos*. Dissertação (Mestrado) — Dissertação de Mestrado, Universidade Federal de Santa Catarina, Programa de Pós-Graduação em Engenharia Elétrica, Florianópolis, SC, Brasil., 2006.

GUIMARãES, A. de A. *Barramento Controller Area Network*. [S.l.], 2006. Disponível em: <http://www.pcs.usp.br/l̃aa/Grupos/EEM>. Acesso em: 10 Novembro 2006.

HARRISON, T. H.; LEVINE, D. L.; SCHMIDT, D. C. The design and performance of a real-time corba event service. In: *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA: ACM Press, 1997. p. 184–200. ISBN 0-89791-908-4.

INSTRUMENTS, N. *FlexRay Automotive Communication Bus*. [S.l.], 2006. Disponível em: <ftp://ftp.ni.com/pub/devzone/pdf/tut_3352.pdf>. Acesso em: 10 Dezembro 2006.

JUNG, C. et al. Projeto e implementação de veículos autônomos inteligentes. *XXV Congresso da SBC - JAI2005 - Jornada de Atualização em Informática*, p. 1358–1406, 2005.

KAISER, J.; BRUDNA, C.; MITIDIERI, C. Cosmic: a real-time event-based middleware for the can-bus. *J. Syst. Softw.*, Elsevier Science Inc., New York, NY, USA, v. 77, n. 1, p. 27–36, 2005. ISSN 0164-1212.

KIM, K. et al. Integrating subscription-based and connection-oriented communications into the embedded CORBA for the CAN bus. In: *IEEE Real Time Technology and Applications Symposium*. [S.l.: s.n.], 2000. p. 178–187.

KIM, K. H. K.; SERRO, C. Can real-time local area network protocols be made robust? *ftdcs*, IEEE Computer Society, Los Alamitos, CA, USA, v. 00, p. 0324, 1995.

KON, F. *Tópicos (avançados) de programação orientada a objetos*. [S.l.], 2007. Disponível em: <http://www.ime.usp.br/k̃on/MAC5715/>. Acesso em: 15 Janeiro 2007.

LANKES, S.; JABS, A.; BEMMERL, T. Integration of a CAN-based Connection-oriented Communication Model into Real-Time CORBA. In: *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS 2003), 11th Workshop on Parallel and Distributed Real-Time Systems (WPDRTS 2003)*. Nice, France: [s.n.], 2003.

LIN. *Local Interconnect Network*. [S.l.], 1999. Disponível em: <http://www.lin-subbus.org>. Acesso em: 20 Novembro 2006.

LINCHY, N. *Distributed Algorithms*. [S.l.]: Morgan Kaufmann Publishers, 1996.

LIU, J. W. S. *Real-Time Systems*. New Jersey, USA: Prentice-Hall, 2000.

MOTOROLA. *MSCAN data sheet*. [S.l.], 2004. Technical Report, freescale.

OLIVIERA, R. S.; FRAGA, J. S.; MONTEZ, C. B. *X Escola de Informaática da SBC-Sul : Programação em Sistemas Distribuídos In: X Escola de Informática da SBC-SUL, Eri 2002 ed*. [S.l.]: Porto Alegre, 2002.

OMG. *Event Service*. [S.l.], 2004.

OMG. *UML 2 Superstructure Final Adopted specification.* [S.l.], 2004.

PAPAIOANNOU, I. N. *Estudo da Eletrônica Embarcada Automotiva e sua situação atual no Brasil*. Dissertação (Mestrado) — Universidade de São Paulo - Escola Politécnica, São Paulo - Brasil, 2005.

PARET, D. *Multiplexed Networks for Embedded Systems: CAN, LIN, FlexRay, Safe-by-Wire...* [S.l.]: John Wiley & Sons, Ltd, 2007.

PELGRIMS, P.; VELDE, G. V. de; VONDEL, B. V. de. *Installing Embedded Linux on the Motorola MPC5200 Lite Evaluation Board*. [S.l.], abril 2004.

PELLER, M.; BERWANGER, J.; GRIESSBACH, R. *Byteflight Specification*. Vienna, 1999.

PHILIPS. *SJA1000 data sheet*. [S.l.], 2000. Technical Report, Philips.

POP, P.; ELES, P.; PENG, Z. *Analysis and Synthesis of Distributed Real-Time Embedded Systems*. [S.l.]: Kluwer Academic Publishers, 2004.

SAE. *Survey of Known Protocols*. [S.l.], April 1993. SAE Information Report J2056/2.

SAE. *The Society of Automotive Engineers*. [S.l.], 2007. Disponível em: <http://www.sae.org/>. Acesso em: 15 Dezembro 2006.

SCHMIDT, D. et al. *Pattern Oriented Software Architecture Volume 2*. [S.l.]: John Wiley & Sons Ltd, 2001.

SPECKS, J. W.; RAJNáK, A. Lin - protocol, development tools, and software interfaces for local interconnect networks in vehicles. *9th International Conference on Electronic Systems for Vehicles*, Oct 2000.

SUN. *Jini Architectural Overview: Technical White Paper, Sun Microsystems, Inc.* [S.l.], 1999.

SUN. *Jini Network Technology: An executive Overview, Sun Microsystems, Inc.* [S.l.], 2001.

TANENBAUM, A. S.; STEEN, M. van. *Distributed Systems: Principles and Paradigms*. [S.l.]: Prentice Hall, 2002.

TECHNIK, P.-S. *PEAK CAN (PCAN)*. [S.l.], 2006. Disponível em: <http://www.peak-system.com/linux/index.htm>. Acesso em: 20 Novembro 2006.

TENNENHOUSE, D. Proactive computing. *Commun. ACM*, ACM, New York, NY, USA, v. 43, n. 5, p. 43–50, 2000. ISSN 0001-0782.

WOLF, W. *Computers as components: principles of embedded computing system design*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001. ISBN 1-55860-541-X.