**Universidade de Aveiro**
**2010**

Departamento de Electrónica, Telecomunicações e Informática

**Fábio Miguel Nogueira Amado**

# Implementação de pilha protocolar tempo-real para vídeo industrial

**Universidade de Aveiro**
**2010**

Departamento de Electrónica, Telecomunicações e Informática

**Fábio Miguel Nogueira Amado**

# Implementação de pilha protocolar tempo-real para vídeo industrial

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Electrónica e Telecomunicações, realizada sob a orientação científica do Professor Doutor Paulo Pedreiras, Professor Auxiliar do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro.

Apoio financeiro da FCT por via do projecto HaRTES - Comutação Ethernet de Tempo-Real, com referência PTDC/EEA-ACR/73307/2006.

Dedico este trabalho
aos meus pais,
à minha namorada,
à minha família,

ao meu avô…

**o júri**

presidente

Prof. Doutor José Alberto Gouveia Fonseca
Professor Associado do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro

vogais

Prof. Doutor Paulo José Lopes Machado Portugal
Professor Auxiliar do Departamento de Engenharia Electrotécnica e de Computadores da Faculdade de Engenharia da Universidade do Porto

Prof. Doutor Paulo Bacelar Reis Pedreiras (Orientador)
Professor Auxiliar do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro

**agradecimentos**

Gostaria de agradecer às seguintes pessoas no âmbito pessoal:

Aos meus pais, que sempre acreditaram em mim, sempre me apoiaram, sempre me motivaram, e me deram condições para ser o que sou hoje. O meu muito obrigado!

À minha namorada, Mariana da Vitória Costa, que me mudou como pessoa, me tornou um ser mais humano, sociável e ouvinte. Sem esquecer que é a minha alma gémea que me completa. Obrigado amor!

A toda a minha família, pelo apoio, amor e carinho dados e pela quota-parte de responsabilidade na minha educação.

A todos os meus amigos que ao longo dos tempos coloriram a minha vida. Quem me conhece sabe bem o valor que dou à amizade. Não serão esquecidos!
"Feliz daquele que encontra um amigo digno desse nome" *Menandro*

À Associação de Defesa Pessoal, na pessoa do Professor Vítor Gomes, que contribuiu activamente para me melhorar como pessoa, dia-a-dia, um pouco de cada vez.

A todos os bons professores que tive a felicidade de encontrar; mais importante do que saberem transmitir o conteúdo programático, souberam incutir o gosto pelo conhecimento, pelo estudo. Sem vocês provavelmente não teria chegado onde cheguei.

No âmbito pessoal e académico que a esta dissertação diz respeito, gostaria de agradecer:

Ao professor Paulo Pedreiras, por me ter acompanhado sempre de uma forma atenta, pela motivação incutida, e pelo conhecimento técnico irrepreensível. Agradeço ainda pelo bom ambiente criado entre orientador e orientando.

Ao Sr. Carlos Breda e Sra. Rosário Breda, por terem apostado em mim, e me terem facilitado a conclusão deste meu processo de formação. Obrigado também pelas palavras amigas e de apoio dadas em momentos de indecisão, demonstrando sempre total disponibilidade e abertura.

Ao meu colega e amigo Nuno Marujo, por me ter ajudado na revisão desta dissertação e ter contribuído com alguns conselhos para a elaboração da mesma.

À empresa Riamolde - Engenharia e Sistemas S.A. por ter emprestado algum do *hardware* utilizado no âmbito desta dissertação.

A todos os meus colegas que ao longo do meu percurso académico de alguma forma contribuíram para a minha formação.

**palavras-chave**

Ethernet, FTT-Ethernet, FTT-SE, Industrial Ethernet, tempo-real, controlo, controlo distribuído, sistemas distribuídos, comunicações industriais

**resumo**

Actualmente as necessidades de sistemas de tempo-real estão presentes em muitos campos da nossa sociedade. Desde um simples sistema de vídeo-conferência até à automação de uma fábrica.
A tendência destes últimos anos é a de uniformizar meios de transmissão de informação tendo em vista a redução de custos.
A rede *Ethernet* está bastante disseminada em vários sectores, sejam eles fabris, domésticos, telecomunicações, etc.
No entanto, esta não consegue fornecer garantias de tempo-real dado aos seus mecanismos indeterminísticos de acesso ao meio.
Existem várias soluções desenvolvidas por diversos fabricantes para tentar colmatar estas limitações inerentes ao protocolo.
Esta dissertação utiliza uma dessas soluções de Industrial Ethernet: o protocolo FTT-SE (*Flexible Time Trigger over Switched Ethernet*), na implementação de uma *stack* tempo-real para uma câmara de vídeo industrial. Adicionalmente são apresentados alguns resultados e conclusões da implementação.

**keywords**

Ethernet, FTT-Ethernet, FTT-SE, Industrial Ethernet, real-time, control, distributed control, distributed systems, industrial communications

**abstract**

Nowadays real-time systems are present in several fields of our society. From a simple video conference system to factory automation.
The trend of recent years is to standardize the means of transmitting information with the purpose of cost reduction.
The Ethernet network is widely disseminated in various sectors, whether manufacturing, domestic, telecommunications, etc.
However, it cannot deliver real-time guarantees given to its indeterministic mechanisms for medium access.
There are several solutions developed by different manufacturers to help overcome these limitations inherent to the protocol.
This dissertation uses one of these solutions for Industrial Ethernet: the FTT-SE protocol (Flexible Time Trigger over Switched Ethernet), on the implementation of a real-time stack for an industrial video camera. Additionally, some results and conclusions of the implementation are presented.

# Conteúdo


1 Enquadramento e motivação ..... 1
1.1 Descrição da dissertação ..... 1
1.1.1 Trabalho anterior ..... 2
1.1.2 Objectivos ..... 2
1.2 Estrutura da Dissertação ..... 3
2 Conceitos Fundamentais ..... 5
2.1 Definição de sistema ..... 5
2.2 Definição de sistema de controlo ..... 5
2.3 Evolução do controlo digital ..... 5
2.4 Sistema de Tempo-Real ..... 6
2.4.1 Significado ..... 6
2.4.2 Tipos de sistemas de tempo-real ..... 7
2.4.3 Classificação do tipo de mensagens em relação à sua periodicidade ..... 8
2.5 Características de um sistema de tempo-real ..... 8
2.6 Sistemas de controlo centralizado *versus* distribuído ..... 9
3 A rede *Ethernet* ..... 11
3.1 Breve descrição do protocolo ..... 11
3.2 Protocolos *Ethernet* existentes com garantias de tempo-real ..... 13
3.2.1.1 PROFINET ..... 13
3.2.1.2 ETHERNET POWERLINK ..... 14
3.2.1.3 TTETHERNET ..... 15
4 *Flexible Time Trigger* (FTT) ..... 19
4.1 Introdução ..... 19
4.2 Breve resumo do paradigma FTT ..... 19
4.2.1 Arquitectura base ..... 19
4.2.2 Funcionalidades ..... 20
4.3 *Flexible Time Trigger over Switched Ethernet* (FTT-SE) ..... 21
4.3.1 Trama FTT-SE ..... 21
4.3.1.1 Tipos de mensagens ..... 21
4.3.1.2 Trigger Message ..... 22
4.3.1.3 Synchronous Data Message – SDM ..... 23
4.3.1.4 Asynchronous Data Message – ADM ..... 24
4.3.1.5 Asynchronous Signalling Message – ASM ..... 24

4.3.2 *Elementary cycle* do protocolo FTT-SE ........ 25
4.4 Variantes do paradigma FTT ........ 26
5 Sistema desenvolvido ........ 27
5.1 Ambiente de desenvolvimento ........ 27
5.2 Breve descrição do *hardware* ........ 28
5.2.1 Placa de desenvolvimento ........ 28
5.2.2 Placa de expansão *Ethernet* ........ 29
5.2.3 Emulador ........ 29
5.2.4 Computadores utilizados ........ 30
5.3 Software existente ........ 30
5.3.1 Código exemplo ........ 30
5.3.2 *Master* simulado ........ 31
5.4 Software desenvolvido ........ 32
5.4.1 Inicializações ........ 32
5.4.2 Rotina de atendimento a eventos *Ethernet* ........ 32
5.4.3 Acções para o envio de uma trama *Ethernet* ........ 34
5.4.4 Acções para a recepção de uma trama *Ethernet* ........ 35
5.4.5 Mecanismo de troca de *buffers* ........ 36
5.5 Descrição dos testes ........ 38
5.5.1 Experiência 1: Tempo de resposta ........ 39
5.5.1.1 Master ........ 39
5.5.1.2 Slave ........ 40
5.5.2 Experiência 2: Integridade dos dados ........ 41
5.5.2.1 Master ........ 42
5.5.2.2 Slave ........ 43
5.5.3 Experiência 3: Fragmentação e reagrupamento ........ 44
5.5.3.1 Master ........ 44
5.5.3.2 Slave ........ 45
5.5.4 Experiência 4: Mecanismo duplo *buffer* ........ 46
5.5.4.1 Master ........ 48
5.5.4.2 Slave ........ 49
5.5.5 Experiência 5: Mecanismo de troca de *buffers* com interferência em N/2 ........ 50
5.5.5.1 Master ........ 50
5.5.5.2 Slave ........ 50
5.5.6 Experiência 6: Mecanismo de troca de *buffers* com interferência de *timer* ........ 52
5.5.6.1 Master ........ 55

5.5.6.2 Slave ....................................................................................55
6 Resultados experimentais....................................................................................57
6.1 Experiência 1: Tempo de resposta....................................................................................57
6.2 Experiência 2: Integridade dos dados ....................................................................................58
6.3 Experiência 3: Fragmentação e reagrupamento ....................................................................................58
6.4 Experiência 4: Mecanismo duplo *buffer*....................................................................................59
6.5 Experiência 5: Mecanismo de troca de *buffers* com interferência em N/2 ...........60
6.6 Experiência 6: Mecanismo de troca de *buffers* com interferência de *timer*..........60
7 Conclusões ....................................................................................63
7.1 Análise de resultados ....................................................................................63
7.2 Análise de objectivos ....................................................................................63
7.3 Trabalho Futuro ....................................................................................63

# Lista de figuras

Figura 3.1 - Formato da trama *Ethernet* ............ 12
Figura 3.2 – Modularidade do protocolo PROFINET ............ 14
Figura 3.3 - Exemplo de um ciclo POWERLINK ............ 15
Figura 3.4 - Os três tipos de tráfego TTEthernet ............ 16
Figura 3.5 - Rede TTEthernet com *bus guardians* redundantes ............ 17
Figura 4.1 – Arquitectura base do paradigma FTT ............ 20
Figura 4.2 - Ciclo elementar do paradigma FTT ............ 20
Figura 4.3 - Trama FTT-SE ............ 21
Figura 4.4 - Estrutura de uma *trigger message* ............ 22
Figura 4.5 – Campos introduzidos por cada mensagem síncrona de dados ............ 23
Figura 4.6 - Campos introduzidos por cada mensagem assíncrona de dados ............ 23
Figura 4.7 – Exemplo genérico de uma *trigger message* ............ 23
Figura 4.8 - Exemplo específico da formação de uma *trigger message* ............ 23
Figura 4.9 - Estrutura de uma mensagem de dados síncrona ............ 24
Figura 4.10 – Estrutura de uma mensagem de dados assíncrona ............ 24
Figura 4.11 – Exemplo genérico de uma mensagem de sinalização de tráfego assíncrono. 25
Figura 4.12 – Campos introduzidos por cada ASM ............ 25
Figura 4.13 - Exemplo de um EC do FTT-SE ............ 26
Figura 5.1 – *Screenshot* do ambiente de desenvolvimento Analog Visual DSP++ ............ 27
Figura 5.2 Versão do ambiente de desenvolvimento ............ 28
Figura 5.3 - Placa de desenvolvimento BF533 EZ-KIT Lite ............ 28
Figura 5.4 - Placa de expansão ADZS-USBLAN-EZEXT ............ 29
Figura 5.5 - Emulador ADZS-USB-ICE ............ 30
Figura 5.6 – Algoritmo das inicializações do software para o nó *slave* ............ 32
Figura 5.7 – Algoritmo da rotina de atendimento de eventos *Ethernet* ............ 33
Figura 5.8 – Algoritmo condensado da gestão de *buffers* ............ 36
Figura 5.9 – Algoritmo da gestão de *buffers* (escrita) ............ 37
Figura 5.10 - Algoritmo da gestão de *buffers* (leitura) ............ 38
Figura 5.11 – Diagrama de *software* do *master* para a experiência 1 ............ 39
Figura 5.12 – Diagrama de *software* do *slave* para a experiência 1 ............ 40
Figura 5.13 - Estrutura do pacote Ethernet para a experiência 2 ............ 41
Figura 5.14 - Diagrama de *software* do *master* para a experiência 2 ............ 42
Figura 5.15 - Diagrama de *software* do *slave* para a experiência 2 ............ 43
Figura 5.16 – Diagrama de *software* do *master* para a experiência 3 ............ 44
Figura 5.17 – Diagrama de *software* do *slave* para a experiência 3 ............ 45
Figura 5.18 - Exemplo do problema de um buffer sem acesso em exclusão mútua ............ 46
Figura 5.19 - Diagrama temporal do teste 4: Mecanismo sequencial de troca de *buffers* ... 47
Figura 5.20 - Diagrama de *software* do *master* para a experiência 4 ............ 48
Figura 5.21 - Diagrama de *software* do *slave* para a experiência 4 ............ 49
Figura 5.22 - Diagrama temporal do teste 5: Mecanismo de troca de *buffers* com interferência em N/2 ............ 50
Figura 5.23 - Diagrama de *software* do *slave* para a experiência 5 ............ 51
Figura 5.24 - Diagrama temporal do teste 6: Mecanismo de troca de *buffers* com interferência de *timer* (1.) ............ 52
Figura 5.25 - Diagrama temporal do teste 6: Mecanismo de troca de *buffers* com interferência de *timer* (2.) ............ 53

Figura 5.26 - Diagrama temporal do teste 6: Mecanismo de troca de *buffers* com interferência de *timer* (3.) ............ 54
Figura 5.27 - Diagrama temporal do teste 6: Mecanismo de troca de *buffers* com interferência de *timer* (4.) ............ 54
Figura 5.28 - Diagrama temporal do teste 6: Mecanismo de troca de *buffers* com interferência de *timer* (5.) ............ 54
Figura 5.29 - Diagrama temporal do teste 6: Mecanismo de troca de *buffers* com interferência de *timer* (6.) ............ 55
Figura 5.30 - Diagrama temporal do teste 6: Mecanismo de troca de *buffers* com interferência de *timer* (7.) ............ 55
Figura 5.31 - Diagrama de *software* do *slave* para a experiência 6 ............ 56
Figura 6.1 – Latência entre a *Trigger Message* do *master* e resposta do *slave* ............ 57
Figura 6.2 – *Jitter* relativo entre pacotes sucessivos de resposta ao *master* (teste 1) ............ 58

# Lista de tabelas

Tabela 6.1 - Valores de latência entre o pedido do *master* e a resposta do *slave* (experiência 1) ..................58
Tabela 6.2 - Valores de latência entre o pedido do *master* e a resposta do *slave* (experiência 3) ..................59
Tabela 6.3 - Valores de latência entre o pedido do *master* e a resposta do *slave* (experiência 4) ..................59
Tabela 6.4 - Valores de latência entre o pedido do *master* e a resposta do *slave* (experiência 5) ..................60
Tabela 6.5 - Valores de latência entre o pedido do *master* e a resposta do *slave* (experiência 6), varrendo um ciclo elementar ..................61

# Lista de siglas e acrónimos

**ADM** – Asynchronous Data Message
**ASM** – Asynchronous Signaling Message
**B&R** – Bernecker + Rainer Industrie-Elektronik
**CAN** – Controller Area Network
**CN** – Controlled Node
**CPU** – Central Processing Unit
**CRC** – Cyclic Redundancy Check
**CSMA/CD** – Carrier Sense Multiple Access/Collision Detection
**EC** – Elementary Cycle
**EPSG** – Ethernet POWERLINK Standardization Group
**FCS** – Frame Check Sequence
**FPGA** – Field-Programmable Gate Array
**FTT-CAN** – Flexible Time Trigger over Controller Area Network
**FTT-Ethernet** – Flexible Time Trigger over Ethernet
**FTT-SE** – Flexible Time Trigger over Switched Ethernet
**IEEE** – Institute of Electrical and Electronics Engineers
**IFG** – InterFrame Gap
**IRT** – Isochronous Real-Time
**ISO** – International Organization for Standardization
**JTAG** – Joint Test Action Group
**LAN** – Local Area Network
**LED** – Light-Emitting Diode
**MN** – Managing Node
**NIC** – Network Interface Controller
**OUI** – Organizationally Unique Identifier
**PDU** – Protocol Data Unit
**PLC** – Programmable Logic Controller
**PReq** – Poll Request
**PRes** – Poll Response
**PROFIBUS** – PROcess FIeld BUS
**PROFINET** – PROcess FIeld NET
**PROFINET CBA** – PROFINET Component Based Automation
**PROFINET IO** – PROFINET Input/Output
**QoS** – Quality of Service
**RAM** – Random Access Memory
**RC** – Rate Constrained [traffic]
**RS-232** – Recommended Standard 232
**RSI** – Rotina de Serviço à Interrupção
**RT** – Real-time
**SAE** – Society of Automotive Engineers
**SDM** – Synchronous Data Message
**SDRAM** – Synchronous Dynamic Random Access Memory
**SoC** – Start of Cycle
**SPOF** – Single Point Of Failure
**TCP/IP** – Transmission Control Protocol/Internet Protocol
**TDMA** – Time Division Multiple Access

**TM** – Trigger Message
**TT** – Time Triggered [traffic]
**USB** – Universal Serial Bus
**VoIP** – Voice over Internet Protocol

## Capítulo 1

# 1 Enquadramento e motivação

Neste capítulo é feita inicialmente uma introdução ao tema desta dissertação, de seguida é apresentado trabalho anteriormente desenvolvido que serviu de base a este e que partilha alguns aspectos, no campo do estudo de sistemas distribuídos e sistemas de tempo-real.

De seguida são apresentados os objectivos desta dissertação e no final deste capítulo é apresentada a estrutura da mesma.

## 1.1 Descrição da dissertação

Ao longo dos últimos 30 anos, têm aparecido inúmeras redes de campo (*fieldbuses*), para dar resposta a carências específicas no âmbito do controlo e automação industrial. No entanto, sofrem de algumas limitações inerentes ao facto de serem soluções particulares e inflexíveis.

Ao longo dos tempos, a tendência tem-se alterado, o mercado tem pedido respostas mais flexíveis e compatíveis. Pretende-se ainda que a solução disponibilize alguma possibilidade de evolução ao longo do tempo, que não implique uma mudança drástica quando estiver datada. Além de tudo isto, enquanto no passado as exigências de largura de banda eram muito reduzidas (pois a quantidade de informação a transmitir era diminuta), hoje em dia, com soluções de controlo baseadas por exemplo, em sistemas de visão, as necessidades de largura de banda são muito elevadas e com tendência para crescer sustentadamente.

A rede *Ethernet* aparentemente responde a todas estes pontos: é uma tecnologia bem conhecida, testada exaustivamente, livre de patentes e bastante normalizada. Oferece ainda a possibilidade de integrar uma única solução de comunicação integrada. Tomando o exemplo de uma fábrica, pode permitir a comunicação desde o controlo de automação, fazer a ponte entre o meio fabril e a parte administrativa, e por fim, responder a todas as necessidades de tecnologias de informação dessa mesma parte administrativa: interligação com a *internet*, rede interna (dados, *Voice over Internet Protocol* – VoIP, vídeo conferência), etc.

Infelizmente, a rede *Ethernet* como a conhecemos tipicamente associada à camada TCP/IP (*Transfer Control Protocol/Internet Protocol*), é incapaz de dar garantias a tráfego de tempo-real. O mecanismo CSMA/CD (*Carrier Sense Multiple Access With Collision Detection*), que existe para prevenir colisões, adiciona um grau de indeterminismo, atrasando o tráfego, e pior que isso, torna impossível de prever os instantes da sua transmissão.

Para colmatar esta limitação da rede *Ethernet*, foi criado um *standard* que evita as colisões: *switched Ethernet*. Além de evitar as colisões, maximiza ainda a largura de banda disponível. No entanto, não resolve o problema de imprevisibilidade do tráfego: em vez de haver indeterminismo ao nível das colisões, existe agora graças ao mecanismo de filas de espera do *switch*. Actualmente, devido à descida de preços dos *switches*, a versão *switched* da rede *Ethernet* substituiu quase totalmente a versão *shared*, seja em ambiente industrial, empresarial ou doméstico.

De forma a dotar a rede *Ethernet* com garantias de tempo-real, diversas entidades têm desenvolvido várias formas e tentado promover as mesmas.

A estas adaptações da rede *Ethernet*, dá-se o nome de *Industrial Ethernet*. Esta dissertação utiliza como base uma dessas adaptações, o protocolo FTT-SE (*Flexible Time Trigger over Switched Ethernet*). Um dos objectivos é o de criar uma camada de *software* que permita fazer o *interface* entre uma câmara de vídeo com porta *Ethernet* e o protocolo FTT-SE.

### 1.1.1 Trabalho anterior

O protocolo FTT-SE [1] utilizado neste trabalho, é uma evolução do protocolo FTT-Ethernet [2], que por sua vez sofre de influências do protocolo FTT-CAN [3]. Este último trabalho visava conferir propriedades de flexibilidade e determinismo ao protocolo CAN (Controller Area Network) [4].

Em [2], foi apresentado o mesmo paradigma sobre *Ethernet* e em [1] sobre *switched Ethernet*. Na dissertação de mestrado [5], foi caso de estudo uma aplicação prática do protocolo FTT-SE.

### 1.1.2 Objectivos

Tem-se assistido a um crescimento enorme no mercado de áreas de visão. Seja na área da indústria (controlo de processos e/ou inspecção de qualidade), na área da robótica móvel (como auxílio à navegação), no auxílio de condução (sistemas que detectam a saída do automóvel da via), ou ainda na área da militar e/ou segurança (detecção de intrusão, etc).

Existindo um grande interesse na área de visão artificial, muitas vezes com uma malha de controlo associada, controlo esse com requisitos temporais de actuação sobre o sistema, seria um tema de estudo muito interessante aplicar uma solução de visão artificial a um protocolo de tempo real, como forma de validação do mesmo.

Tem-se ainda observado a descentralização do poder de processamento nos sistemas em geral. Essa tendência deve-se a vários factores, como por exemplo a redundância, o custo, etc. Os sistemas passam de centralizados para distribuídos. Para um sistema distribuído funcionar, a comunicação entre os vários componentes é vital.

É importante então criar condições para que essa comunicação seja bem sucedida, e para isso existem protocolos de comunicação de tempo-real.

No âmbito desta dissertação, implementou-se a camada de transporte dos dados adquiridos por uma câmara com *interface Ethernet*, tendo em vista o seu futuro tratamento por um controlador. A câmara com *interface Ethernet* e respectivo suporte foram disponibilizados pela empresa Riamolde - Engenharia e Sistemas S.A. .

Os objectivos desta dissertação são os seguintes:

- Estudo de conceitos básicos sobre tempo-real e comunicações tempo-real;
- Estudo da rede *Ethernet* e do standard IEEE 802.1D-1998;
- Estudo do protocolo FTT-SE;
- Estudo do *hardware* e *software* das câmaras RiaVision;
- Implementação e validação da *stack* FTT-SE na câmara RiaVision;
- Implementação e validação de uma *stack* TCP/IP com gestão de *Quality of Service* (QoS) (layer 2) na câmara RiaVision;

- Avaliação das funcionalidades de gestão dinâmica de QoS do protocolo FTT-SE;
- Integração no demonstrador.

## 1.2 Estrutura da Dissertação

Esta dissertação encontra-se dividida em sete capítulos, divididos da seguinte forma:

- **Capítulo 1 - Enquadramento e motivação**
  Neste primeiro capítulo, é apresentada a motivação que levou a este trabalho, é descrito algum trabalho anterior feito neste âmbito, e por fim é descrita a estrutura desta dissertação.
- **Capítulo 2 - Conceitos Fundamentais**
  Este capítulo apresenta alguns conceitos fundamentais necessários para a compreensão desta dissertação.
- **Capítulo 3 - A rede *Ethernet***
  Sendo hoje em dia um *standard* importantíssimo nas comunicações, e sendo ainda a rede que esta dissertação utiliza como base de trabalho, neste capítulo a rede *Ethernet* é apresentada com maior detalhe. Inicialmente é relatada alguma história desta rede, seguidamente é apresentado o protocolo de comunicação, depois são descritas algumas formas que foram criadas para complementar a rede *Ethernet* de formas a conseguir ter garantias de tempo-real.
- **Capítulo 4 - *Flexible Time Trigger (FTT)***
  Este capítulo é dedicado à explicação em maior detalhe do paradigma utilizado nesta dissertação, o paradigma FTT. É descrita a sua história e evolução ao longo dos tempos, é apresentada a sua arquitectura base e o protocolo FTT-SE é apresentado em maior detalhe.
- **Capítulo 5 - Sistema desenvolvido**
  Neste capítulo é apresentado em maior detalhe o sistema desenvolvido que serviu como base de trabalho desta dissertação. É apresentado o *hardware* utilizado, e o *software* existente que lhe serve de suporte. Finalmente é apresentado o *software* desenvolvido no âmbito da dissertação.
- **Capítulo 6 - Resultados experimentais**
  Neste capítulo apresenta-se um conjunto de testes experimentais que foram desenvolvidos com o objectivo de verificar a correcção da implementação da camada de comunicação, bem como o respectivo desempenho temporal.
- **Capítulo 7- Conclusões**
  No último capítulo desta dissertação, é feita uma análise aos resultados obtidos e objectivos alcançados, e são feitas algumas sugestões de trabalho futuro tendo em vista a continuação desta dissertação.

## Capítulo 2

# 2 Conceitos Fundamentais

Neste capítulo são introduzidos conceitos fundamentais que suportam teoricamente o trabalho desenvolvido no âmbito desta dissertação.

São introduzidos os conceitos de sistema, sistema de controlo, sistema de tempo-real, sistema centralizado *versus* distribuído.

## 2.1 Definição de sistema

Um sistema, é um conjunto de entidades que interagem entre si ou são interdependentes, formando um todo-integrado [6].

Este conceito de todo-integrado, significa que cada componente do sistema coopera com outros à sua volta, tendo um objectivo comum.

Um exemplo de um sistema, será o de um automóvel. O volante serve para mudar de direcção, o pedal do travão para reduzir a velocidade do automóvel, etc. No entanto, todos os componentes do automóvel servem o objectivo final, de formar um meio de transporte.

Existem vários tipos de sistema, entre eles mecânicos, eléctricos, hidráulicos, económicos, etc.

## 2.2 Definição de sistema de controlo

A um sistema que consiga gerir, comandar, dirigir ou regular outros elementos ou sistemas, define-se como sistema de controlo [7].

A teoria de sistemas de controlo visa modelar matematicamente o sistema de controlo em estudo, fazendo com que este manipule as causas, tendo em vista o efeito desejado.

## 2.3 Evolução do controlo digital

Com a diminuição dos custos da electrónica digital, tem-se assistido à evolução de uma tendência da substituição do controlo analógico pelo controlo digital.

A disponibilidade de processadores, quer na variedade de escolha, quer na disponibilidade por parte dos fabricantes por produção em grande escala, tem sido responsável por esta evolução.

O controlo digital tem também muitas outras vantagens, entre elas:

- Baixo custo;
- Generalidade (tanto se pode controlar um processo hidráulico, como um eléctrico, por exemplo);
- Parâmetros de controlo facilmente ajustáveis (edição de software);
- Insensibilidade ao meio, garantindo fiabilidade e reprodutibilidade de resultados;
- Escalabilidade;

Com o aparecimento deste tipo de sistemas computacionais, foi necessário criar respostas para tratar de sistemas com restrições temporais.

## 2.4 Sistema de Tempo-Real

Um sistema de tempo-real, é um sistema computacional com restrições temporais precisas em relação ao ambiente que o rodeia [8].

Como tal, o correcto comportamento do sistema deixa de ser apenas baseado nos resultados produzidos na parte de controlo, mas também nos instantes temporais em que estes ocorrem. ”A reacção [de um sistema] que ocorra fora de tempo pode ser inútil ou mesmo até perigosa” [8].

Os sistemas cruciais nos quais a nossa sociedade assenta estão muitas vezes dependentes em parte, ou totalmente, de sistemas de tempo-real. Alguns exemplos incluem sistemas:

- Militares,
- Aviónicos,
- Automotivos.

### 2.4.1 Significado

A característica base de um sistema de tempo-real é, tal como o nome indica, o tempo. O sistema depende portanto, do meio e do tempo em que as interacções ocorrem.

Além disso, a parte “real” do nome, implica que a reacção do sistema a eventos externos terá de ocorrer durante a sua evolução. Para tal acontecer, é fundamental que a escala temporal usada para medir o tempo interno do sistema, seja a mesma que é usada para medir o tempo externo do sistema, ou seja o tempo em que os eventos ocorrem.

Existe no entanto, um equívoco comum ao considerar que um sistema de tempo-real é um sistema rápido. Se tal fosse verdade, todos os problemas aliados ao tempo-real, seriam resolvidos com processadores com grande poder de processamento.

Num sistema de tempo-real, é usual haver processamento paralelo, e consequentemente várias tarefas simultâneas. Assim, pode acontecer que umas tarefas bloqueiem outras, gerando atrasos imprevisíveis, não adiantando o facto de ter um processador mais rápido. É necessário que exista um escalonamento adequado.

É portanto óbvio que não interessa a um sistema de tempo-real que seja rápido, mas sim adaptado ao meio em que se insere. Espera-se que o sistema seja reactivo.

Importa também introduzir o conceito de *deadline*. Este é o tempo máximo em que uma tarefa tem de terminar a sua execução.

As tarefas podem ser classificadas segundo as consequências de perdas de *deadlines*:

***Hard real-time*** – Tarefas em que a perda de *deadline* pode levar a uma falha catastrófica do sistema. Pode implicar perdas materiais elevadas ou perdas de vidas humanas. Em relação à utilidade do resultado do processamento fora de tempo das tarefas, este pode ser inútil ou até mesmo prejudicial.

Um exemplo de uma *deadline* perdida com consequências catastróficas: um condutor de um automóvel carregar no pedal do travão e este não é accionado. As consequências podem ser gravíssimas (danos no automóvel e/ou ocupantes do veículo, outros condutores, etc).

Um exemplo em que o resultado do processamento de uma tarefa seja prejudicial: um atraso na leitura da posição de um *flap* de um avião. A leitura terá informação errada

nesse instante, levando o sistema a agir de forma contrária ao necessário no momento, contribuindo para a perda do controlo do avião.

***Firm real-time*** – Tarefas em que a perda de *deadline* leva a um comportamento problemático mas não catastrófico do sistema. Se o sistema for modelado para conseguir conviver com perdas de *deadlines,* pode-se tentar minimizar as suas consequências. Um exemplo, *streaming* de vídeo e/ou voz. No caso do incumprimento de uma *deadline* de um pacote, deixaria de haver imagem/ voz. A utilidade dos pacotes que chegam fora de tempo é nula, e como tal são descartados.

No entanto, é possível contornar este problema utilizando técnicas para tentar recriar os pacotes em falta a partir de anteriores.

***Soft real-time*** – Tarefas em que a perda de *deadlines* causa degradação do desempenho do sistema. A utilidade dos resultados do processamento das tarefas após perda de *deadline* vai diminuindo ao longo do tempo. Por exemplo, apesar das imagens adquiridas por um *robot* através de visão artificial estarem ligeiramente atrasadas devido ao processamento de imagem, estas poderão ter alguma utilidade para tomadas de decisões em relação ao posicionamento do *robot* (desvio de obstáculos).

***Non real-time*** – As tarefas que não têm restrições temporais. A utilidade dos dados gerados por estas tarefas têm sempre utilidade ao longo do tempo. São normalmente executadas num regime de *best-effort*, ou seja, se houver recursos após a execução das tarefas *hard* e *soft-real time*, então as *non real-time* serão executadas.

Tipicamente os sistemas têm tarefas *hard*, *soft* e *non real-time*, como visto no exemplo anterior. É então necessário agendar as diferentes tarefas conforme as diferentes prioridades. Neste tipo de sistema híbrido, é preciso garantir os tempos das *hard tasks*, e minimizar os tempos de resposta das *soft tasks*.

### 2.4.2 Tipos de sistemas de tempo-real

Existem duas classificações de sistemas de tempo-real. Cada sistema é classificado conforme a sua tolerância à perda de *deadlines* por parte das suas tarefas.

***Non real-time***

Os sistemas do tipo *non real-time* são aqueles que não apresentam nenhum requisito temporal do tipo *hard, soft* ou *firm.*

***Soft real-time***

Os sistemas do tipo *soft* são aqueles que apresentam pelo menos um requisito temporal do tipo *soft* ou *firm*. No caso de incumprimento destes mesmos requisitos, o sistema pode ficar comprometido mas não de uma maneira que possa ser considerada perigosa.

***Hard real-time***

Os sistemas do tipo hard são aqueles que apresentam, pelo menos, um requisito temporal do tipo hard. No caso de incumprimento destes mesmos requisitos, o sistema

ficará comprometido de uma forma grave, e as consequências que daí advêm são perigosas e/ou catastróficas.

### 2.4.3 Classificação do tipo de mensagens em relação à sua periodicidade

Num sistema distribuído de tempo-real, o tráfego pode ser dividido em três grupos. Estes grupos categorizam-no quanto à natureza de activação das mensagens.

Estes três grupos são:

- **Tráfego periódico** – mensagens que são despoletadas com uma cadência certa, tendo como intervalo entre si um período bem definido (por ex: leitura do valor de um sensor);

- **Tráfego esporádico** – mensagens que são despoletadas com um intervalo indefinido, mas com um tempo mínimo entre si (por ex: tráfego numa rede de computadores – pode não haver, como haver uma rajada de mensagens, mas nunca mais do que uma num intervalo individual);

- **Tráfego aperiódicas** – mensagens cujo padrão de activação só consegue ser caracterizado probabilisticamente (por ex: o carregar num botão por parte de um utilizador).

## 2.5 Características de um sistema de tempo-real

Existem algumas características necessárias para se tratar de um sistema de tempo-real:

- **Temporais:** As acções que o sistema pratica têm de estar correctas e no tempo certo;

- **Carga máxima:** As tarefas têm de ser agendadas para que não colidam temporalmente, ou seja, que não fiquem suspensas quando não era suposto. Ora para tal acontecer, o sistema deve ser projectado para quando as tarefas estão a funcionar no pico da sua actividade (*peak-load*). Tem de se garantir que o sistema está preparado para lidar com o **pior caso** (*worst-case scenario*);

- **Previsibilidade:** Para se garantir um mínimo de desempenho, o sistema tem de conhecer todas as consequências das suas decisões. Se alguma tarefa não puder ser garantida, têm de ser tomadas medidas alternativas, para que não ocorram problemas de maior;

- **Tolerância a falhas:** Um sistema de tempo-real é na maior parte dos casos um sistema de alta disponibilidade [9] e é sempre responsável por tarefas críticas. É um sistema que se pretende sempre funcional. Não pode o sistema falhar por causa de uma falha de um componente individual. Os componentes críticos do sistema ou SPOF [10] (*Single Point Of Failure*) terão então de ser tolerantes a falhas, através de redundância.

- **Fácil manutenção/Modularidade:** O sistema deve ser modular em cada componente para ser fácil a sua manutenção, quer por necessidade (avaria de um componente) ou para fácil introdução de novas funcionalidades.

## 2.6 Sistemas de controlo centralizado *versus* distribuído

A forma mais simples e intuitiva, é a de controlo centralizado. Uma só entidade controla todo um sistema. Este tipo de controlo é eficaz quando existem poucas variáveis a controlar: quer pelo poder de processamento exigido, quer pela complexidade de interligação da entidade central com todas as outras.

Tomando como exemplo uma fábrica, em que existem milhares de sensores, ter um só computador central poderia ser problemático: milhares de metros de cabo, e seria preciso um super-computador, com elevado poder de processamento e com inúmeras ligações. Além de ser difícil tal acontecer, teria ainda um custo elevado.

Estes factores, conjugados com o aumento de disponibilidade de processadores de baixo custo, levou à vulgarização do conceito de arquitectura distribuída. Com a disseminação desta, surgiu a necessidade de criar uma rede comum a todos os dispositivos: surgiu assim o conceito de *fieldbus* ou rede de campo.

## Capítulo 3

# 3 A rede *Ethernet*

A rede *Ethernet* [11] surgiu no centro de investigação Palo Alto da Xerox, criação de Bob Metcalfe. Foi criada a 22 de Maio de 1973, a partir da necessidade de ligar um computador a uma impressora laser. Na altura conseguia disponibilizar uma largura de banda de 2.94Mbit/s.

Foi adoptada como um standard do IEEE [12], 802.3 [13-14], e tem sofrido várias evoluções ao longo dos tempos. Apresentam-se algumas normas mais relevantes:

- 802.3i-1990 – 10BASE-T over twisted pair
- 802.3u-1995 – 100Base-T Fast Ethernet and Auto-Negotiation
- 802.3x-1997 – Full Duplex standard
- 802.3z-1998 – 1000Base-T Gigabit Ethernet over fiber-optic
- 802.3ab-1998 – 1000BASE-T Gbit/s Ethernet over twisted pair
- 802.3ae-2003 – 10 Gbit/s (1,250 MB/s) Ethernet over fiber-optic
- 802.3af-2003 – Power over Ethernet (12.95 W)
- 802.3at-2009 – Power over Ethernet enhancements (25.5 W)
- 802.3ba-2010 – 40 Gbit/s and 100 Gbit/s Ethernet

## 3.1 Breve descrição do protocolo

Inicialmente, a rede *Ethernet* apresentava uma topologia em *bus* com cabo coaxial, aparecendo posteriormente a topologia estrela, com *hubs* a permitirem a ligação de vários nós ou *Network Interface Cards* (NIC's). Porém em ambos os casos existe apenas um domínio de colisão. Para resolver colisões, o protocolo utiliza um esquema de acesso ao meio (CSMA/CD).

De uma forma simplificada, quando um NIC, pretende transmitir, actua da seguinte forma:

- Espera até que o meio esteja livre.
- Espera um determinado tempo (*interframe gap period*).
- Começa a transmissão.
- Houve colisão? Se sim, então salta para o método de colisão detectada.
- Limpa os contadores de retransmissão e termina a transmissão.

Caso haja uma colisão:

- Efectua a transmissão de uma sequência especial (*jam signal*), de forma a todos os NIC's detectarem a colisão.
- Incrementa o contador de retransmissão.
- Foi atingido o número máximo de tentativas de retransmissão? Se sim, é assinalado como erro de retransmissão.
- Espera um tempo pseudo-aleatório, na tentativa de não voltar a transmitir ao mesmo tempo que outro NIC.
- Recomeça o procedimento inicial de tentativa de transmissão.

O tempo pseudo-aleatório, é calculado com base no número de retransmissões, pois duplica a cada colisão detectada. Este mecanismo tem o nome de *exponential back-off*.

Após 10 colisões o valor de espera não aumenta mais (*truncated exponential back-off)*. Se ao final de 16 tentativas não se conseguir transmitir a mensagem, dá-se um erro de transmissão.

Existindo por exemplo apenas dois NIC's na rede, a probabilidade de colisão é muito baixa. No entanto, à medida que uma rede cresce, a probabilidade de colisões aumenta. Ao se verificar este aumento, o desempenho da rede desce consideravelmente (*thrashing*). Para colmatar este problema, surgiu em 1989 na empresa Kalpana, o primeiro *switch Ethernet*.

O s*witch* tem características muito vantajosas sobre os *hubs*:

- Cria um domínio de colisão por porta. No caso de cada porta ter ligada apenas um NIC elimina por completo as colisões.
- Contém uma tabela em que associa a cada porta o(s) NIC('s) a si ligados, possibilitando o reencaminhamento de tráfego apenas para a estação destino.
- Permite transmissões simultâneas, aumentado a largura de banda útil para cada estação.

Resolve ainda um problema do protocolo *Ethernet*, não ligado ao desempenho. A filosofia "um fala, todos ouvem", levava a que uma pessoa mal intencionada pudesse interceptar todo o tráfego da rede. Facilmente se imagina os problemas que daí poderiam advir (roubo de passwords ou outro tipo de informação privilegiada, etc).

O formato da trama *Ethernet* está representado na Figura 3.1 contém os seguintes campos:

- **Preamble Field** (8 *bytes*) – Contém uma sequência padrão, usada para sincronização entre estações.
- **Destination** (6 *bytes*) – Endereço físico da estação-destino.
- **Source** (6 *bytes*) – Endereço físico da estação-origem.
- **Data** (0-1500 *bytes*) – Dados a serem enviados.
- **Padding** (0-46 *bytes*) – Usado no caso em que o tamanho da trama é inferior ao tamanho mínimo imposto pelo standard usado. Para 100Base-T por exemplo, é de 46 Bytes para o campo de dados, 64Bytes total.
- **Frame Check Sequence (FCS)** (4 *bytes*) – Utilizado para detecção e correcção de erros.
- **Interframe Gap (IFG)** (12 *bytes*) – Tempo mínimo entre frames consecutivas, para facilitar a sincronização entre estações,

| Preamble + Start-of-Frame-Delimiter 8 bytes | Mac Address Destination 6 bytes | Mac Address Source 6 bytes | Ethertype 2 bytes | Data 42-1496 bytes | Frame Check Sequence 4 bytes | Interframe Gap 12 bytes |
|---|---|---|---|---|---|---|

**Figura 3.1 - Formato da trama *Ethernet***

Cada estação é obrigada a possuir um endereço físico único. Seguindo a norma do IEEE, os 3 primeiros *bytes* (OUI – *Organizational Unique Identifier*) identificam

unicamente o fabricante da NIC, sendo a atribuição dos outros 3 *bytes* da responsabilidade do fabricante, de modo a cumprir com a unicidade de endereços.

Existem muitos protocolos que pretendem conferir características de tempo-real à rede *Ethernet*, seguidamente serão apresentados alguns dos mais importantes.

## 3.2 Protocolos *Ethernet* existentes com garantias de tempo-real

Nas últimas duas décadas, têm aparecido inúmeras redes de campo para resolver problemas específicos, apresentando uma solução personalizada para cada problema. Com uma variedade tão grande, torna-se difícil perceber qual a certa para uma determinada dificuldade, e é necessário conhecer e dominar vários protocolos. Além disso, muitas das vezes são soluções proprietárias de difícil manutenção e diagnóstico de problemas.

Além disso, com a corrente demanda por largura de banda, torna-se claro que algumas redes de campo não conseguem dar resposta a necessidades emergentes de multimédia (controlo por visão ou vigilância por exemplo) e que não têm margem para expansão futura.

A rede *Ethernet* está bem disseminada, é livre de patentes, tem tido um aumento sustentando de capacidade de largura de banda ao longo dos tempos, acompanhando as necessidades dos utilizadores. Todavia, como previamente referido, o standard *Ethernet* não permite garantias de tempo-real. No entanto existem várias iniciativas que têm como objectivo colmatar essas lacunas; a este tipo de soluções que assentam no standard *Ethernet* e que o pretendem completar de modo a dar essas mesmas garantias, dá-se o nome de *Industrial Ethernet*.

Vamos de seguida abordar alguns destes protocolos conhecidos que têm como objectivo tornar a rede *Ethernet* numa rede de tempo-real.

#### 3.2.1.1 PROFINET

PROFINET [15] (*PROcess FIeld NET*) é o padrão de *Industrial Ethernet* da PROFIBUS & PROFINET International [16] (PI) para automação. O conceito inerente a esta tecnologia propõe uma abordagem modularizada do sistema, dando ao utilizador a possibilidade de criar sistemas complexos de uma forma estruturada e sistemática. Aborda uma série de conceitos, tais como, automação distribuída (PROFINET CBA) e dispositivos de campo descentralizados (PROFINET IO).

No que ao PROFINET CBA (*Component Based Automation*) diz respeito, este parte do princípio que um sistema de automação pode frequentemente ser dividido em subsistemas autónomos, permitindo assim estruturá-lo de uma forma mais clara e organizada.

Um sistema PROFINET CBA é composto por vários módulos que englobam, cada um, componentes eléctricos, mecânicos, de electrónica e *software*. A funcionalidade de um módulo é encapsulada no mesmo, podendo ser visto como uma "caixa negra" pelo sistema, dado que é acedido por uma *interface* padrão. Assim, é possível criar sistemas compostos, efectuando a interligação de vários módulos entre si. Este método apresenta ainda a vantagem de tornar os módulos altamente reutilizáveis.

A PROFINET CBA faz uso de TCP/IP e RT (*real-time communication*) apresentando ciclos de transmissão com valores desde a ordem da centena de milissegundos (TCP/IP) à dezena de milissegundos (RT), o que a torna bastante adequada para comunicações entre controladores (ex: PLC's – *Programmable Logic Controller*).

Esta característica de modularidade, que pretende responder a todas as necessidades de comunicação, é apresentada na Figura 3.1 (retirada de [17]).

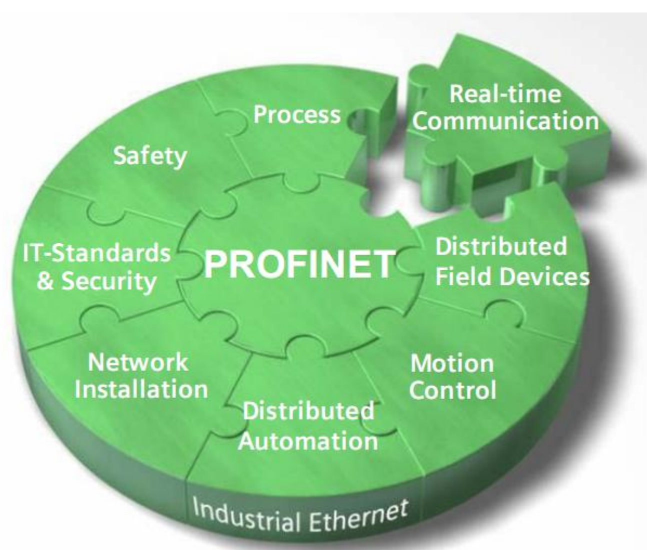


**Figura 3.2 – Modularidade do protocolo PROFINET**

A parte PROFINET IO permite a utilização de dispositivos de campo descentralizados sobre *Ethernet*. Define o acesso a periféricos de uma forma homogénea, especificando o método de troca de dados entre controladores e dispositivos. Estes seguem uma filosofia produtor-consumidor e têm como característica a elevada velocidade na troca de dados entre si. A PROFINET IO utiliza comunicações RT (*real-time communication*) e IRT (*Isochronous Real-Time*). As comunicações RT apresentam ciclos de transmissão da ordem dos milissegundos sendo adequadas para a ligação a periféricos. As comunicações IRT apresentam ciclos de transmissão da ordem dos milissegundos, o que as torna adequadas a aplicações de controlo de movimento.

Em conjunto, PROFINET CBA e PROFINET IO, cobrem a generalidade das necessidades para aplicações no âmbito da automação de uma fábrica. É precisamente no meio industrial que o PROFINET está bastante implantado, principalmente suportado pela Siemens.

O PROFINET tem ainda a vantagem de ter evoluído do PROFIBUS, e como tal teve um enorme mercado receptivo à sua entrada.

#### 3.2.1.2 ETHERNET POWERLINK

O protocolo Ethernet POWERLINK [18-23] é um protocolo determinístico de tempo-real sobre o *standard Ethernet.* Foi inicialmente introduzido pela empresa de automação Bernecker & Rainer Industrie-Elektronik [24] (B&R) em 2001 e em 2002 foi fundado o grupo Ethernet POWERLINK Standardization Group (EPSG), que é responsável pela sua manutenção. Em 2003 foi apresentada a versão 2, que acrescenta a camada de aplicação como extensão à versão 1. Esta camada é baseada nos mecanismos definidos pelo CANopen.

Tem como principal objectivo manter a máxima compatibilidade com dispositivos *Ethernet* existentes. É uma solução implementada por software e como tal não utiliza necessariamente *hardware* proprietário.

Para evitar colisões e maximizar a utilização da largura de banda, é utilizado um esquema de *Time Division Multiple Access* [25] (TDMA), chamado neste caso de *Time Slicing*. O tempo de acesso ao barramento é dividido em POWERLINK *cycles*, e por sua vez cada um deste ciclo é divido em pequenas partes e cada nó apenas pode transmitir no tempo que lhe é reservado.

O acesso ao meio é regulado por um nó *master*, chamado de *Managing Node* (MN). Além do controlo das comunicações, está ainda encarregue de gerar o sinal de relógio para sincronização dos nós na rede. Os nós *slave* ou *Controlled Nodes* (CN) apenas transmitem quando interrogados pelo MN.

Um ciclo elementar é dividido em quatro partes, como ilustrado na Figura 3.3 (retirado de [21]):

- ***Start Period***: Nesta fase o MN transmite uma mensagem de início de ciclo (*Start Of Cycle* – SoC) para todos os CN. Serve para sincronização de todos os nós.
- ***Cyclic Period***: Fase isócrona em que é possível a transmissão de dados. De acordo com um escalonamento pré-configurado, os CN são convidados a transmitir os seus dados através de uma mensagem *Poll Request* (PReq). E reposta a este pedido é feita com mensagens do tipo *Poll Response* (PRes).
- ***Asynchronous Period***: Fase em que se podem transmitir dados assíncronos sem necessidades de tempo-real.
- ***Idle Period***: Tempo de inactividade do barramento até o próximo ciclo começar.

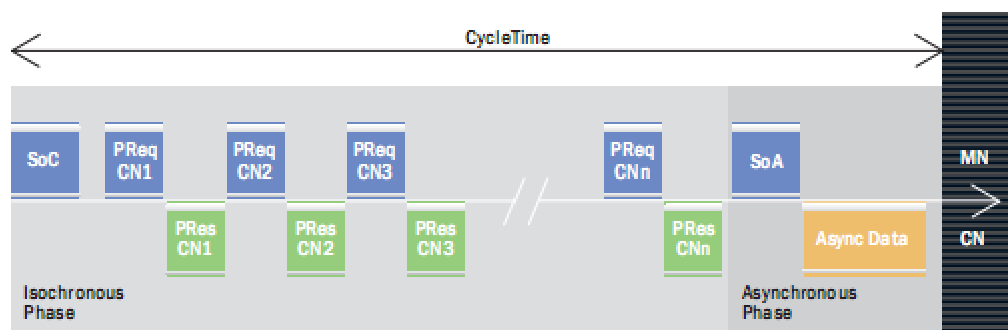


**Figura 3.3 - Exemplo de um ciclo POWERLINK**

Qualquer tipo de topologia pode ser usado devido à utilização de *hubs Ethernet*. Como não existe a possibilidade de colisões, o número máximo de *hubs* em cascata deixa de estar limitado a dois.

#### 3.2.1.3 TTETHERNET

Desenvolvido pela empresa TTTech Computertechnik AG [26], o protocolo TTEthernet [27] é escalável permitindo a expansão de novos nós na rede. É ainda determinístico e tolerante a falhas. Caracteriza-se por dispor de sincronização do relógio por todos os componentes da rede, e utiliza um esquema de TDMA. Em caso de falha num dos nós, em que estes estejam a transmitir dados errados, ou o próprio nó se apercebe da falhar e suspende a sua função, ou os *switches* detectam a falha na integridade das mensagens por CRC (*Cyclic Redundancy Check*).

Existem três tipos de mensagem: *time-triggered traffic* (TT), *rate-constrained traffic* (RC) e *best-effort traffic* (BE), como se pode observar na Figura 3.4 (retirada de [27]).

A forma principal de comunicação é através de mensagens TT. Existem blocos elementares de tempo, chamados de *cluster cycle*, em que os nós podem comunicar dentro da janela temporal. Este tipo de tráfego tem precedência sobre os outros, e segundo os autores, são o tipo óptimo de mensagens para comunicação em sistemas distribuídos de tempo real. Um exemplo clássico será o tráfego gerado por um nó sensor, que se prevê que seja periódico.

Para mensagens que não têm requisitos de tempo-real tão elevados e que são menos determinísticas, existe o tipo RC. Este tipo de tráfego garante largura de banda por aplicação e que os atrasos e desvios temporais sofridos por este tipo de tráfego são bem definidos. Por exemplo, o sinal de um componente X-by-wire [28].

Por fim existe o tipo BE, que utiliza a largura de banda não utilizada pelos tipos TT ou RC, em que não existem garantias de tempo-real. Um bom exemplo deste tipo de tráfego seria o de disponibilizar o serviço de internet aos utilizadores de um avião que tem a sua rede de tempo-real baseada em TTEthernet. Evita-se assim a necessidade de utilizar duas redes para dois serviços diferentes.

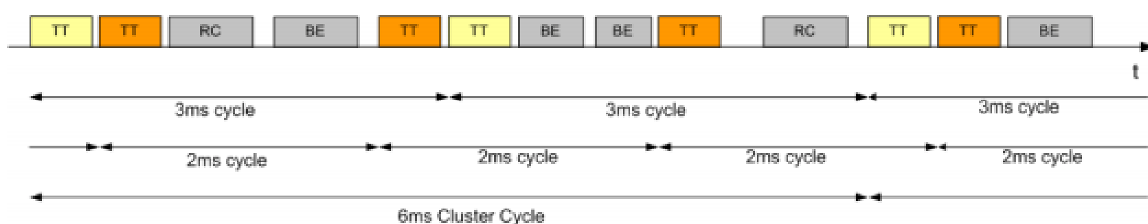

**Figura 3.4 - Os três tipos de tráfego TTEthernet**

O protocolo é escalável, pois tanto permite uma aplicação simples com dois nós para uma aplicação de controlo industrial, como permite cobrir toda uma rede necessária, por exemplo, para um avião. A grande vantagem deste facto, é que se consegue descer os custos gerados da necessidade de mão-de-obra qualificada para trabalhar com o protocolo.

Os *switches* TTEthernet podem ser configurados como *bus-guardian*s centrais, isto é, no caso de detectarem transmissões defeituosas por parte dos nós, impedem as suas mensagens de se propagarem na rede. Permitem ainda bloquear o comportamento *babbling idiot* (quando um nó não pára de transmitir).

São suportadas todas as camadas físicas definidas no standard IEEE 802.3 para *switched networks*, e ainda *sub-networks* com diferentes larguras de banda (100Mbit/s, 1Gbit/s, …).

É possível ainda aumentar a tolerância a falhas, adicionando múltiplos *bus guardians* redundantes, como se pode observar na Figura 3.5 (retirada de [27]).

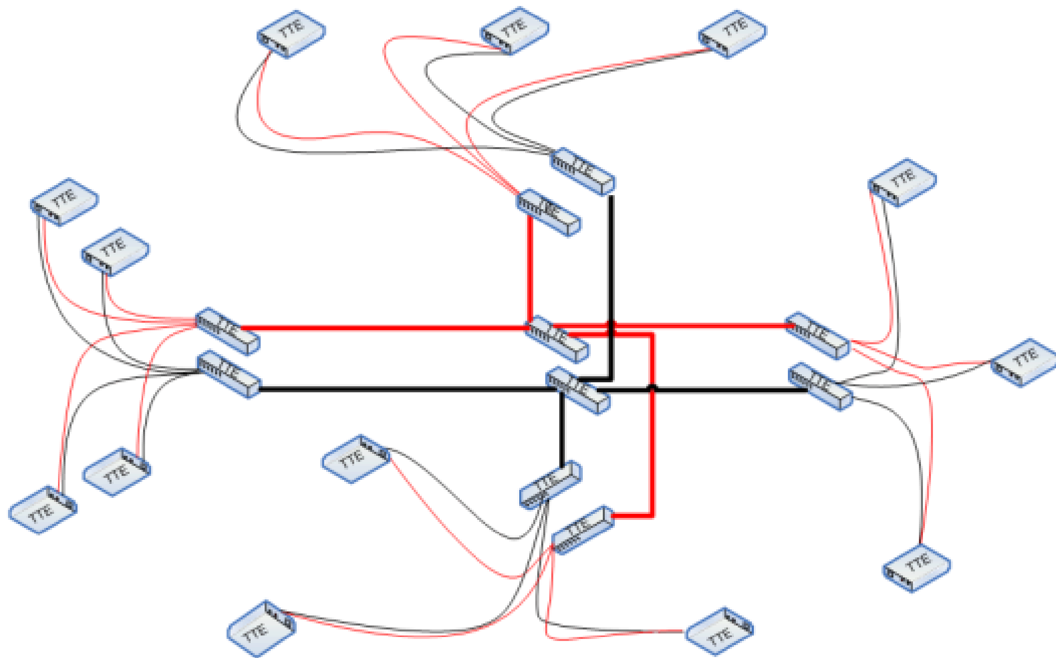


**Figura 3.5 - Rede TTEthernet com *bus guardians* redundantes**

Existem vários produtos existentes, tanto de desenvolvimento como de aplicação. É portanto um protocolo já com aplicação real. Estão em curso à data procedimentos que têm como objectivo a estandardização em algumas organizações com importância no mundo da indústria (ISO, SAE, IEEE) com data prevista de 2012.

A empresa Honeywell foi a primeira a utilizar o protocolo TTEthernet nos seus programas de produção aeroespacial e automotivo.

Devido ao facto do trabalho desta dissertação se basear no protocolo FTT-SE, este será discutido em maior detalhe no próximo capítulo.

# Capítulo 4

# 4 *Flexible Time Trigger* (FTT)

Neste capítulo será apresentado com maior detalhe o protocolo *Flexible Time Trigger over Switched Ethernet* (FTT-SE), dada a importância que este tem no âmbito desta dissertação.

## 4.1 Introdução

O paradigma *Flexible Time Trigger* (FTT [2]) foi criado na Universidade de Aveiro no Laboratório de Sistemas Electrónicos, com base no protocolo FTT-CAN [3]. Na sua génese esteve o objectivo conciliar flexibilidade operacional com requisitos de tempo-real, propriedades que os protocolos existentes à altura tendiam a considerar como antagónicas. Conseguiu-se tal objectivo adoptando uma arquitectura centralizada baseada numa extensão do paradigma *master-slave*, em que o *master* concentra os requisitos de comunicação e gere o acesso ao barramento. Durante o desenvolvimento do FTT-CAN, chegou-se à conclusão que haviam conceitos fundamentais que eram possíveis de serem abstraídos da base CAN, e criar um paradigma mais generalista, denominado paradigma FTT.

Dado o crescente interesse em redes de campo utilizando *Ethernet*, foi um passo lógico tentar portar essa arquitectura para esse meio, e surgiu então o FTT-Ethernet. O problema do indeterminismo neste caso, está relacionado com o mecanismo CSMA/CD do protocolo *Ethernet*.

O protocolo sofreu várias evoluções, sendo aplicado nomeadamente a *shared Ethernet* e posteriormente a *switched Ethernet*. Esta última evolução teve um grande impacto em termos de facilidade de implementação e eficiência, pois deixou de ser necessário realizar o controlo estrito dos instantes de transmissão, devido à capacidade de serialização dos *switches Ethernet*. Houve também uma grande evolução na largura de banda disponível, devido à existência de caminhos paralelos, que permitem em certos casos a transmissão simultânea de dados.

Foram já desenvolvidos outros trabalhos como aplicação prática do trabalho teórico desenvolvido, como por exemplo na área da mecatrónica [5].

## 4.2 Breve resumo do paradigma FTT

### 4.2.1 Arquitectura base

O paradigma FTT implementa uma arquitectura produtor-distribuidor-consumidor, com um nó central, o *master*, que é responsável pelo escalonamento das mensagens. Este nó é o responsável por servir de árbitro quando os nós *slaves* tentam aceder ao barramento para comunicar.

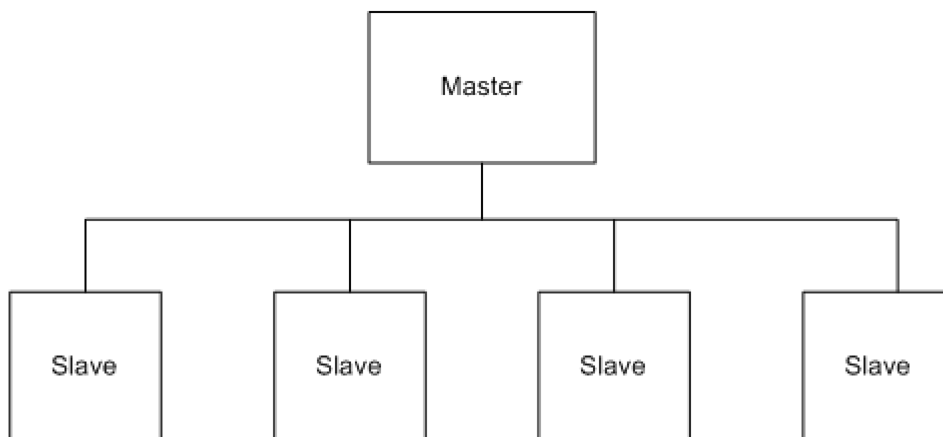

**Figura 4.1 – Arquitectura base do paradigma FTT**

Essa gestão é possível através da utilização de uma unidade temporal mínima, chamada de *Elementary Cycle* (EC) ou ciclo elementar. Este período de tempo é a divisão mínima do acesso ao barramento.

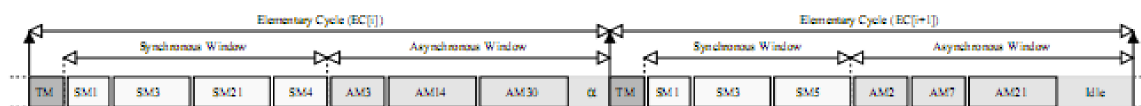

**Figura 4.2 - Ciclo elementar do paradigma FTT**

Na Figura 4.2 (retirada de [2]), podem-se observar representados dois ECs com os seus diversos componentes:

- ***Trigger Message* (TM)**
  Esta mensagem é enviada pelo *master* para o barramento, e além de permitir a sincronização entre todos os nós da rede (pois marca o início do EC), indica também que mensagens poderão ser enviadas nesse EC.
- ***Synchronous Window* (Janela Síncrona)**
  Intervalo de tempo em que é possível enviar mensagens síncronas (periódicas) que tenham sido escalonadas pelo *master*.
- ***Asynchronous Window* (Janela Assíncrona)**
  Intervalo de tempo em que é possível enviar mensagens assíncronas (aperiódicas) que tenham sido escalonadas pelo *master*.
- ***Idle Time* (Tempo de inactividade)**
  Tempo de inactividade do barramento necessário para garantir o início atempado do EC seguinte. Note-se que pode dever-se a não existirem de facto mensagens prontas ou a o fim do ciclo estar próximo e então a transmissão de uma eventual mensagem poder exceder o tempo de ciclo restante (EC *overrun*).

### 4.2.2 Funcionalidades

O paradigma FTT prevê que o escalonamento das mensagens seja efectuado de uma forma central pelo *master*, sendo assim possível alterar facilmente o escalonamento a ser utilizado, e que posteriormente é disseminado pela TM. Graças a esta característica, o protocolo suporta admissão e remoção *online* de *streams* de mensagens, com garantias temporais e maximização da largura de banda disponível. Além disso o algoritmo de escalonamento é uma entidade de *software* residente no *master*, logo podem ser

implementados algoritmos de escalonamento arbitrários e independentes do protocolo base. Finalmente, o FTT suporta quer tráfego síncrono (i.e., activado pela passagem do tempo) quer assíncrono (i.e., activado por um evento externo). Deste conjunto de propriedades advém o adjectivo *flexible* do nome do protocolo. De uma forma mais concisa, o protocolo FTT-SE exibe as seguintes funcionalidades [2]:

- Comunicação *time-triggered* com flexibilidade operacional;
- Suporte para alterações dinâmicas tanto no conjunto de mensagens, como no escalonamento das mesmas;
- Controlo de admissão *online* para garantir a pontualidade do tráfego tempo-real;
- Indicação da exactidão temporal das mensagens tempo-real;
- Suporte de diferentes tipos de tráfego: *event-triggered, time-triggered, hard real-time, soft real-time* e *non real-time;*
- Isolamento temporal: os diferentes tipos de tráfego não devem interferir entre si;
- Uso eficiente da largura de banda;
- Suporte eficiente de mensagens *multicast*.

## 4.3 *Flexible Time Trigger over Switched Ethernet* (FTT-SE)

Nesta secção é apresentada a evolução natural do FTT-Ethernet: o protocolo FTT-SE.

### 4.3.1 Trama FTT-SE

A trama FTT-SE (Figura 4.3) é uma trama específica baseada na trama *Ethernet*. Contém os seguintes campos:

- MAC (*Media Access Control*) Address Destination (6 *bytes*): Endereço físico do nó destino.
- MAC Address Source (6 *bytes*): Endereço físico do nó origem.
- Ethertype (2 *bytes*): campo identificador do protocolo. No caso do protocolo FTT-SE tem o valor 0x8FF0.
- FTT-SE PDU (Protocol Data Unit) (4 a 1500 *bytes*): Campo de dados (de configuração ou no sentido estrito).
- Padding (0 a 42 *bytes*): Campo utilizado no caso de a mensagem ter menos que 64 *bytes*.

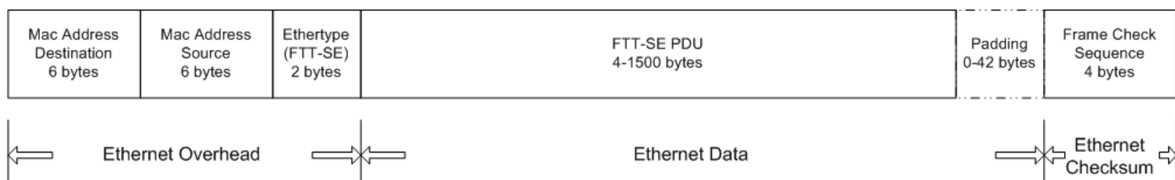


**Figura 4.3 - Trama FTT-SE**

#### 4.3.1.1 Tipos de mensagens

Nesta secção serão apresentados os tipos de mensagens que são possíveis de serem transmitidos pelo protocolo FTT-SE:

1. *Trigger Message*
2. Mensagem síncrona de dados (*synchronous data message* – SDM)

3. Mensagem assíncrona de dados (*asynchronous data message* – ADM)
4. Mensagem assíncrona de sinalização (*asynchronous signaling message* – ASM)
5. Mensagem de inactividade (*Idle*)

Apenas duas ressalvas em relação à representação das tramas que serão apresentadas seguidamente:

1. Todas as mensagens são encapsuladas em tramas *Ethernet*, sendo precedidas do cabeçalho *Ethernet* e terminadas pelo FCS e IFG.
2. Na eventualidade da trama *Ethernet* ser menor que o tamanho mínimo de 64 *bytes*, está implícita a necessidade de *padding*.

#### 4.3.1.2 Trigger Message

A Figura 4.4 apresenta a estrutura base de uma *trigger message*. Esta contém os seguintes campos:

- **type** (2 *bytes*): tipo de mensagem. Neste caso, como se trata de uma *trigger message*, contem o valor **FTT_MST_MSG**.
- **seq_no** (1 *byte*): número de sequência da *trigger message* enviada. Serve para que os nós consigam detectar se perderam alguma *trigger* message.
- **flag_s** (1 *byte*): campo reservado para *flags* de controlo.
- **nsm** (2 *bytes*): número de pedidos de mensagens síncronas contidas no campo de dados.
- **nam** (2 *bytes*): número de pedidos de mensagens assíncronas contidas no campo de dados.
- **data** (até 1492 *bytes* ou limitação do protocolo): dados propriamente ditos da *trigger message*. Aqui serão inseridos N campos, tantos quantos referenciados por **nsm** e **nam**. Na Figura 4.4 podem-se observar os campo inseridos por cada **nsm**, e na Figura 4.5 estão representados os campos introduzidos por cada **nam**.

| type<br>2 bytes | seq_no<br>1 byte | flag_s<br>1 byte | nsm<br>2 bytes | nam<br>2 bytes | Data<br>[ ] |
|---|---|---|---|---|---|

**Figura 4.4 - Estrutura de uma *trigger message***

Na Figura 4.5 apresentam-se os campos inseridos na *trigger message* necessários para efectuar um pedido de transmissão de uma mensagem síncrona por parte de um *slave*. São eles:

- **SMesgId** (2 *bytes*): identificador lógico e único de mensagem a ser transmitida, sendo o seu escalonamento efectuado pelo *master*.
- **Smesg_frag_no** (2 *bytes*): o FTT-SE suporta fragmentação de mensagens. Este campo identifica qual o fragmento a ser enviado.

De igual forma, existe um mecanismo análogo para a gestão de mensagens assíncronas (Figura 4.6). Note-se que o formato é semelhante. A diferença entre mensagens síncronas e assíncronas prende-se essencialmente com a sua activação. As primeiras são escalonadas autonomamente pelo *master*, de acordo com a passagem do tempo, enquanto as segundas

são escalonadas também pelo *master* mas na sequência de um pedido explícito, geralmente decorrente de um evento externo.

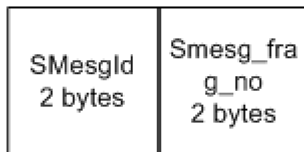


**Figura 4.5 – Campos introduzidos por cada mensagem síncrona de dados**

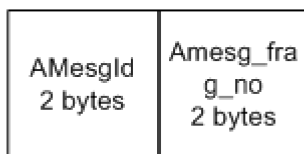


**Figura 4.6 - Campos introduzidos por cada mensagem assíncrona de dados**

Na Figura 4.7, é apresentada de uma forma genérica a formação de uma *trigger message*. Na Figura 4.8, é apresentado um exemplo específico.

**Figura 4.7 – Exemplo genérico de uma *trigger message***

Exemplo:

Os campos **nsm** e **nam** contem os valores 2 e 1 respectivamente. Então a *trigger message* gerada conteria dois pedidos de mensagens síncronas e um pedido de mensagem assíncrona.

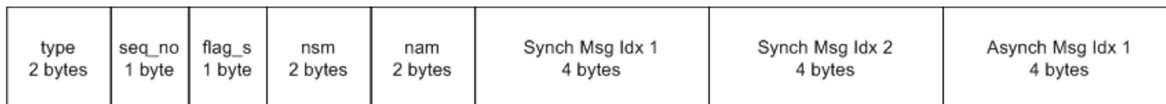


**Figura 4.8 - Exemplo específico da formação de uma *trigger message***

#### 4.3.1.3 Synchronous Data Message – SDM

Quando existe a necessidade de envio de dados de uma forma periódica (por exemplo, a temperatura de um sensor), estes dados são modelados como tráfego síncrono, e como tal, é utilizada esta trama para o seu envio.

A trama SDM (Figura 4.9) contém os seguintes campos:

- **type** (2 *bytes*): tipo de mensagem. Neste caso, como se trata de uma mensagem de dados síncrona, contem o valor **FTT_SDATA_MSG**.
- **id** (2 *bytes*): identificador lógico e único da mensagem.
- **seq_no** (1 *byte*): número de sequência da SDM a enviar. Serve para detectar eventuais perdas de mensagens.
- **flag_s** (1 *byte*): campo reservado para *flags* de controlo.
- **frag_seqno** (2 *bytes*): número identificador do fragmento a ser enviado pela estação visada.
- **data** (até 1492 *bytes*): dados propriamente ditos que são possíveis de serem transmitidos.

| type<br>2 bytes | id<br>2 bytes | seq_no<br>1 byte | flag_s<br>1 byte | frag_seqn<br>o<br>2 bytes | Data<br>38-1492 bytes |
|---|---|---|---|---|---|

**Figura 4.9 - Estrutura de uma mensagem de dados síncrona**

#### 4.3.1.4 Asynchronous Data Message – ADM

Quando existe a necessidade de envio de dados de uma forma aperiódica ou esporádica (por exemplo, um alarme ou a pressão de um botão de um utilizador), estes dados são modelados como tráfego assíncrono, e como tal, é utilizada esta trama para o seu envio.

A trama ADM (Figura 4.10) contém os seguintes campos:

- **type** (2 *bytes*): tipo de mensagem. Neste caso, como se trata de uma mensagem de dados síncrona, contem o valor **FTT_ADATA_MSG**.
- **id** (2 *bytes*): identificador lógico e único da mensagem a ser transmitida.
- **seq_no** (2 *bytes*): número de sequência da mensagem ADM enviar. Serve para detectar eventuais perdas de mensagens.
- **flag_s** (1 *byte*): campo reservado para *flags* de controlo.
- **frag_seqno** (2 *bytes*): número identificador do fragmento a ser enviado pela estação visada.
- **data** (até 1492 *bytes*): dados propriamente ditos que são possíveis de serem transmitidos.

| type<br>2 bytes | id<br>2 bytes | seq_no<br>1 byte | flag_s<br>1 byte | frag_seqn<br>o<br>2 bytes | Data<br>38-1492 bytes |
|---|---|---|---|---|---|

**Figura 4.10 – Estrutura de uma mensagem de dados assíncrona**

#### 4.3.1.5 Asynchronous Signalling Message – ASM

Tal como anteriormente referido, na eventualidade de existirem dados assíncronos a serem enviados é necessário notificar o *master* para que este as inclua no escalonamento. As ASM servem para este propósito, reportando periodicamente ao *master* o estado interno das filas dos diversos nós.

A trama ASM (Figura 4.11) contém os seguintes campos:

- **enum** (2 *bytes*): tipo de mensagem. Neste caso, como se trata de uma mensagem de sinalização de dados assíncrona, contem o valor **FTT_ASTATUS_MSG**.
- **id** (2 *bytes*): identificador lógico e único do nó.
- **seq_no** (2 *bytes*): número de sequência da ASM.
- **flag_s** (1 *byte*): campo reservado para *flags* de controlo.
- **nam** (2 *bytes*): número de pedidos de mensagens assíncronas contidas no campo de dados.
- **data** (até 1494 *bytes*): estado de cada uma das filas de mensagens assíncronas.

| type<br>2 bytes | seq_no<br>1 byte | flag_s<br>1 byte | nam<br>2 bytes | Asynch Msg Queue Idx 1<br>4 bytes | (...) | Asynch Msg Queue Idx N<br>4 bytes |
|---|---|---|---|---|---|---|

**Figura 4.11 – Exemplo genérico de uma mensagem de sinalização de tráfego assíncrono**

De uma forma semelhante com o que acontece com a TM, o tamanho da trama ASM é de tamanho variável, sendo função do número contido em **nam**.

O conjunto de campos apresentado na Figura 4.12, é acrescentado no campo de dados por cada mensagem aperiódica que se pretende transmitir.

| AMesgId<br>2 bytes | AMesgQu<br>eueLen<br>2 bytes |
|---|---|

**Figura 4.12 – Campos introduzidos por cada ASM**

Imaginando um exemplo em que seja preciso enviar uma mensagem assíncrona (por exemplo um alarme e respectiva hora), a trama seria constituída da seguinte forma:

- o campo **nam** conteria o valor 1, pois era necessário enviar uma mensagem;
- o campo **AMesgId** conteria o valor do identificador lógico da mensagem a transmitir;
- o campo **AMesgQueueLen** conteria o valor 1, pois era necessário escalonar o envio de 1 mensagem.

### 4.3.2 *Elementary cycle* do protocolo FTT-SE

Foi já apresentado no início deste capítulo o EC do FTT-Ethernet. O meio partilhado era único, a rede *Ethernet.* Com a introdução de *switches,* passa a ser possível vários nós comunicarem ao mesmo tempo. A Figura 4.13 (adaptada de [1]) ilustra exactamente isso: vários nós a comunicarem simultaneamente.

A Figura 4.13 mostra ainda a existência de quatro sub-divisões temporais do EC:

1. ***Guard Window:*** Período em que o *master* transmite as TM's para os nós que deverão transmitir informação nesse EC.
2. ***Turn-around:*** Tempo de resposta, valor configurável. Deve-se ter algum cuidado com este valor. Se for um valor muito alto, resta pouco tempo para transmitir informação, se for um valor muito baixo, alguns nós com pior desempenho podem não conseguir responder a tempo. Tipicamente é um valor configurado com o tempo de resposta do pior nó da rede.
3. ***Synch Window:*** Janela de tempo em que se pode transmitir tráfego síncrono/periódico.
4. ***Assynch Window:*** Janela de tempo em que se pode transmitir tráfego assíncrono (aperiódico ou esporádico).

Os valores da janela síncrona e da janela assíncrona, não são valores fixos, maximizando assim o aproveitamento da largura de banda.

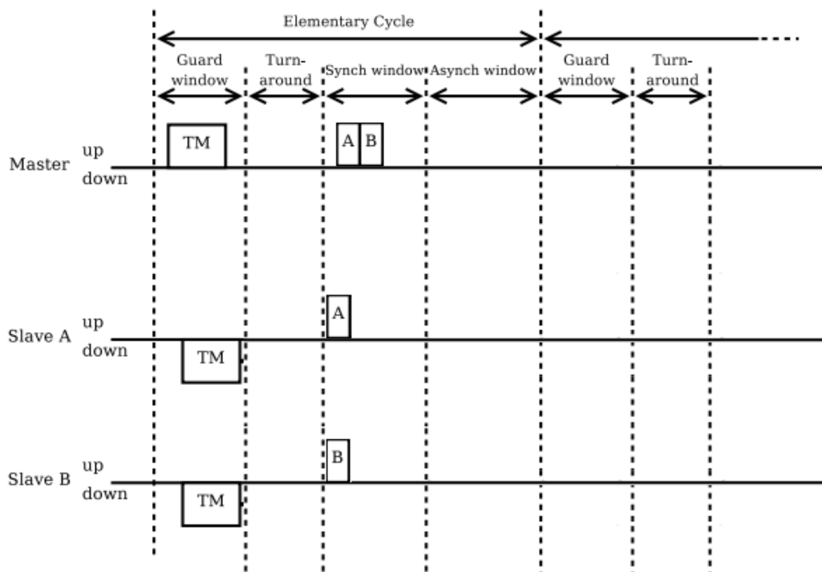


**Figura 4.13 - Exemplo de um EC do FTT-SE**

## 4.4 Variantes do paradigma FTT

As secções anteriores incidiram especificamente sobre o protocolo FTT-SE. Há todavia outras implementações do paradigma FTT sobre tecnologia *Ethernet*. Foi dada ênfase ao FTT-SE devido ao facto de ter sido esta a variante específica sobre a qual o trabalho reportado nesta dissertação incidiu. Para efeitos de complementar a informação sobre esta família de protocolos, referem-se de seguida as outras duas implementações existentes, bem como as suas principais diferenças em relação ao FTT-SE.

A primeira implementação do paradigma FTT foi efectuada sobre *shared Ethernet* (FTT-Ethernet) [2]. Esta implementação tem algumas diferenças em relação ao FTT-SE na parte que respeita ao mecanismo de controlo de acesso ao meio. Enquanto no FTT-SE os nós podem transmitir as mensagens concorrentemente, pois o *switch* efectua a sua serialização, no FTT-Ethernet é necessário disparar as transmissões das mensagens em instantes diferentes, a fim de evitar colisões. Assim, a TM é ligeiramente modificada, de forma a incluir informação temporal de disparo. Adicionalmente, trata-se de um meio *broadcast* e *half-duplex*, logo a caracterização das mensagens é distinta (e.g. não é necessário indicar os portos a que os nós se ligam).

Mais recentemente foi desenvolvido o *FTT-Enabled switch*. Trata-se de um *switch Ethernet* modificado, implementado em tecnologia FPGA (*Field-Programmable Gate Array*). A principal diferença em termos da operação do protocolo é que deixou de ser necessário notificar o *master* da existência de mensagens assíncronas. Os nós enviam as mensagens assíncronas em instantes arbitrários. Quando chegam ao *switch,* este coloca-as numa fila e notifica internamente o *master*, para que este as possa incluir no escalonamento. Adicionalmente o *switch* traz benefícios importantes em termos de tolerância a falhas, pois pode efectuar verificações sobre as mensagens que chegam e, caso detecte anomalias (e.g. um nó com comportamento do tipo *babbling idiot*), pode bloquear o respectivo tráfego.

## Capítulo 5

# 5 Sistema desenvolvido

Neste capítulo são apresentados os componentes do sistema criado. Numa primeira fase é apresentado o ambiente de desenvolvimento utilizado, de seguida o *hardware* usado, depois o *software* que serviu de base ao sistema, e finalmente o *software* desenvolvido no âmbito da dissertação.

## 5.1 Ambiente de desenvolvimento

O ambiente de desenvolvimento utilizado foi o fornecido pela empresa Analog Devices [29], a mesma empresa que fabrica o *hardware* que será descrito de seguida. Foi utilizada a versão 5.0.7.0 (Update 7) do programa Visual DSP++.

A Figura 5.1 apresenta um *screenshot* do ambiente de desenvolvimento utilizado e a Figura 5.2 apresenta a versão desse mesmo ambiente de desenvolvimento.

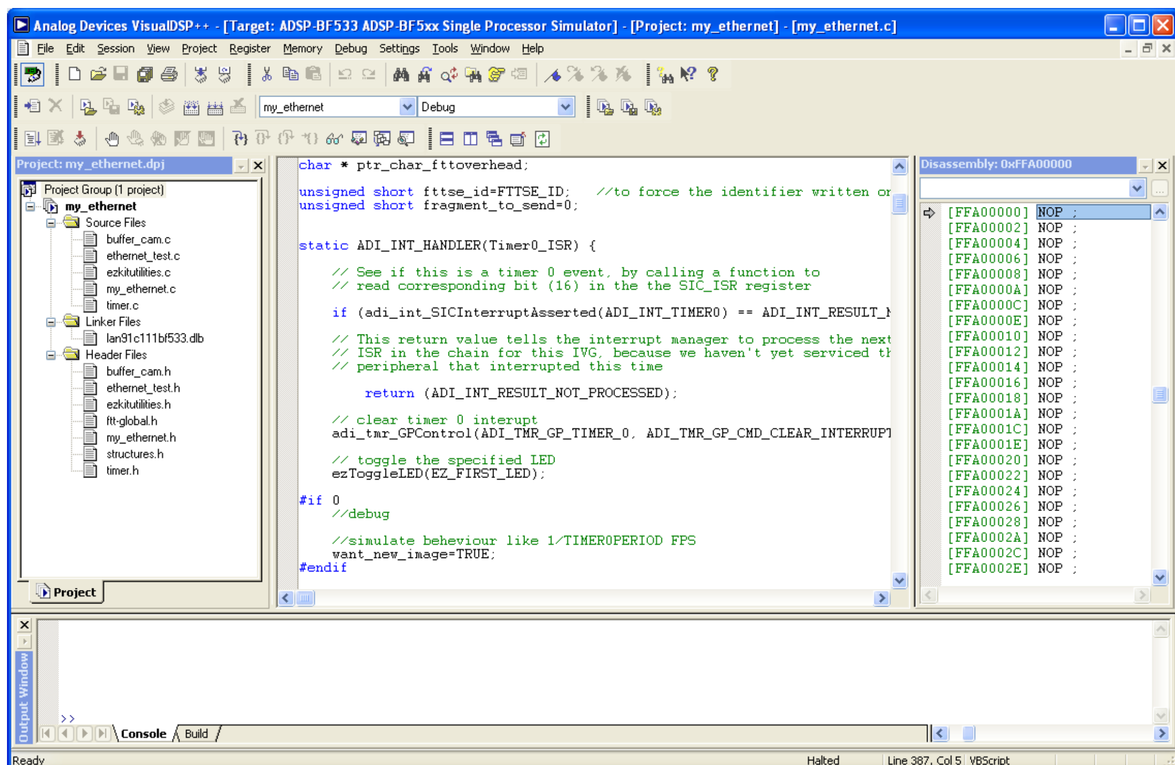


**Figura 5.1 – *Screenshot* do ambiente de desenvolvimento Analog Visual DSP++**

O ambiente de desenvolvimento Visual DSP++ tem as seguintes características:

- Suporta programação em linguagem C e *assembly*.
- Inclui diversas bibliotecas de código, entre elas *software* que permite inicializar/interagir com a *interface Ethernet, drivers* de baixo nível para botões, LEDs, *timers*, etc
- No que à facilidade de *debug* diz respeito, a ferramenta permite o *debug* passo-a-passo, e análise do conteúdo da memória de dados e de programa.

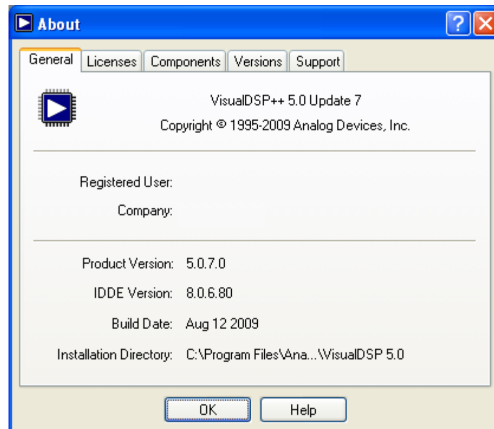


**Figura 5.2 Versão do ambiente de desenvolvimento**

## 5.2 Breve descrição do *hardware*

O *hardware* utilizado pode ser dividido em três partes: a placa de desenvolvimento propriamente dita, uma placa de expansão (que permite adicionar uma porta *Ethernet* e uma porta USB – *Universal Serial Bus*), e o emulador.

Além do que diz respeito à placa de desenvolvimento, foram utilizados também dois computadores, um para emular um *master* FTT, e outro para ser utilizado como plataforma de desenvolvimento, a par do *software* já apresentado.

### 5.2.1 Placa de desenvolvimento

A placa utilizada é a ADZS-BF533-EZLITE [30] da empresa Analog Devices, que se apresenta na Figura 5.3 [31]. Trata-se de uma placa de desenvolvimento para avaliação do processador ADSP-BF533. Complementando o ambiente de desenvolvimento Visual DSP++, é possível projectar um produto que futuramente será produzido em massa para o consumidor final.

**Figura 5.3 - Placa de desenvolvimento BF533 EZ-KIT Lite**

Actualmente, a placa referida possui as seguintes características:

- Processador ADSP-BF533 Blackfin®
- 64 MB (32M x 16-bit) SDRAM
- 2 MB (512K x 16-bit x 2) memória flash

- ADV7183 video decoder com 3 jacks RCA de entrada
- AD1836 96 kHz audio codec com 4 entradas e 6 saídas jacks RCA
- ADV7171 video encoder com 3 saídas jacks RCA
- ADM3202 RS-232 line driver/receiver

Foi no entanto utilizada a revisão 1.2 da placa, que possuía apenas 32MB de SDRAM.

Como se pode observar, esta placa de desenvolvimento contém ligações que permitem a utilização em muitos outros campos, como por exemplo na área de processamento de sinal. Não possui no entanto, uma porta *Ethernet* necessária para o âmbito desta dissertação. No entanto, possui uma ligação de expansão, na qual é possível ligar outros periféricos, personalizando assim a placa conforme as necessidades do utilizador.

### 5.2.2 Placa de expansão *Ethernet*

A placa de expansão utilizada, é a Blackfin USB-LAN EZ-Extender [32], que adiciona uma ligação *Ethernet* e USB. Para o âmbito desta dissertação, apenas interessa a ligação *Ethernet*, pois o trabalho proposto não envolve o protocolo USB. A placa de expansão é mostrada na Figura 5.4 [33].

**Figura 5.4 - Placa de expansão ADZS-USBLAN-EZEXT**

### 5.2.3 Emulador

Além da placa de desenvolvimento, e da placa de expansão, é ainda necessário um emulador. Este emulador é necessário, para permitir o *debug* passo-a-passo com possibilidade de ver e alterar o conteúdo da memória ou dos registos do processador, entre *break-points*.

Foi utilizado o emulador ADZS-USB-ICE [34], apresentado na Figura 5.5 [35]. Este emulador detecta automaticamente e suporta todos os processadores da Analog que possuam um *interface* JTAG [36] (*Joint Test Action Group*). Este *interface* é bastante usado para o *debug* de sistemas embutidos.

Liga-se ao PC através de um *interface* USB. Este emulador permite então carregar o código escrito e compilado no ambiente de desenvolvimento VisualDSP++.

**Figura 5.5 - Emulador ADZS-USB-ICE**

### 5.2.4 Computadores utilizados

Foram utilizados dois computadores para a avaliação do sistema: o computador (A) permitia criar, editar, compilar e carregar o programa na placa de desenvolvimento, e um segundo computador (B), permitia simular um *master* FTT-SE, correndo um programa específico para o efeito.

Características do computador (A):

- Intel Celeron 2.80Ghz
- 512 MB Ram
- Placa de rede com *chipset Ethernet* 3Com 3C920B-EMB-WNM
- Windows Xp Service Pack 3

Características do computador (B):

- Notebook Asus A3000
- Intel Centrino 1600 MHz
- Placa de rede com *chipset Ethernet* Realtek RTL8139C
- 512MB Ram
- Linux Fedora 10
- Kernel 2.6.25.5-117.fc10.i686 SMP

## 5.3 Software existente

Nesta secção é apresentado o *software* que serviu de base ao trabalho desenvolvido no âmbito desta dissertação.

### 5.3.1 Código exemplo

O ponto de partida para a elaboração deste trabalho foi o de correr código-exemplo fornecido juntamente com o ambiente de desenvolvimento. De início exemplos simples, como acender um LED, fazê-lo piscar segundo um *timer*, etc, até haver um mínimo de conhecimento do sistema em causa.

Após este período inicial, utilizou-se o exemplo Ethernet_test.c presente em "C:\Program Files\Analog Devices\VisualDSP 5.0\Blackfin\Examples\ADSP-BF533 EZ-KIT Lite\Power_On_Self_Test\EZ-USBLAN POST", devidamente alterado para permitir que este exemplo corresse inserido num projecto independente. Verificou-se então que este exemplo inicializava partes fundamentais para o funcionamento do *device driver Ethernet*.

Foi então utilizado este código como sendo a base do trabalho, alterando o seu comportamento de modo a cumprir os requisitos.

### 5.3.2 *Master* simulado

Dada a natureza *master-slave* do protocolo implementado, a realização dos testes experimentais requer o envio de pacotes de controlo. Pacotes estes gerado pelo *master*, elemento fundamental na rede como decisor máximo dos instantes precisos em que a comunicação deverá ocorrer. Devido ao facto desta dissertação versar sobre um nó *slave* e não sobre um nó *master*, foi sendo criado um *master* simulado à medida das necessidades do momento. *Com* a finalidade de agilizar o processo de testes, desenvolveu-se uma aplicação executada em ambiente PC, que simula um FTT-SE *master*, gerando os pacotes de controlo adequados a cada um dos testes realizados.

O comportamento do *master* FTT-SE foi simulado por um programa desenvolvido com base num programa já existente criado pelo Professor Paulo Pedreiras. O programa existente enviava e recebia pacotes *Ethernet* em Linux, pelo que foi aproveitado o código de inicialização do *device driver*. Uma vez mais, o comportamento base foi alterado para cumprir os requisitos necessários para ser um *master* FTT.

## 5.4 Software desenvolvido

Nesta secção é descrito o *software* desenvolvido para a implementação da *stack* FTT-SE no *hardware* apresentado na Secção 5.3. Os resultados da implementação serão enunciados no capítulo seguinte.

### 5.4.1 Inicializações

Para que a *stack* esteja em condições de execução é necessário que se efectue uma série de inicializações, para configurar o *hardware* e *software*.

Começa-se por alocar memória para a *heap* do programa. De seguida, aloca-se memória para os dados do programa e para o mecanismo de troca de *buffers.*

As interrupções são inicializadas e seguidamente LEDs, botões e *timers*. De seguida são efectuadas as atribuições das RSIs (Rotinas de Serviço à Interrupção) aos *timers* e efectuada a inicialização do driver *Ethernet*. Por fim, são activados os *timers*.

A figura Figura 5.6 apresenta o diagrama que implementa o algoritmo acima descrito.

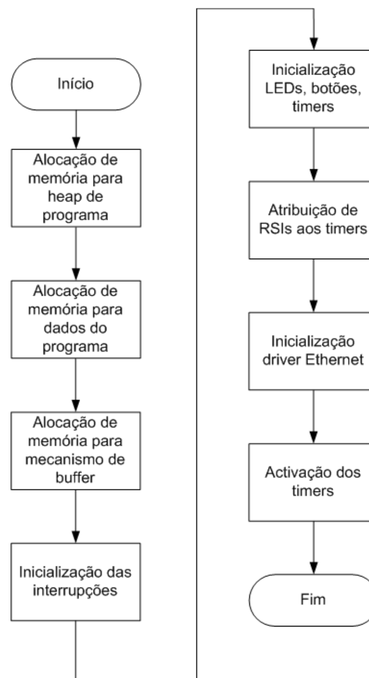


**Figura 5.6 – Algoritmo das inicializações do software para o nó *slave***

### 5.4.2 Rotina de atendimento a eventos *Ethernet*

Para lidar com a *interface Ethernet* utilizou-se uma rotina (*LanCallBack*) para atendimento das interrupções despoletadas pela mesma, englobando nela os mecanismos de decisão necessários para a implementação da *stack.*

A Figura 5.7 descreve sucintamente o algoritmo que esta rotina implementa.

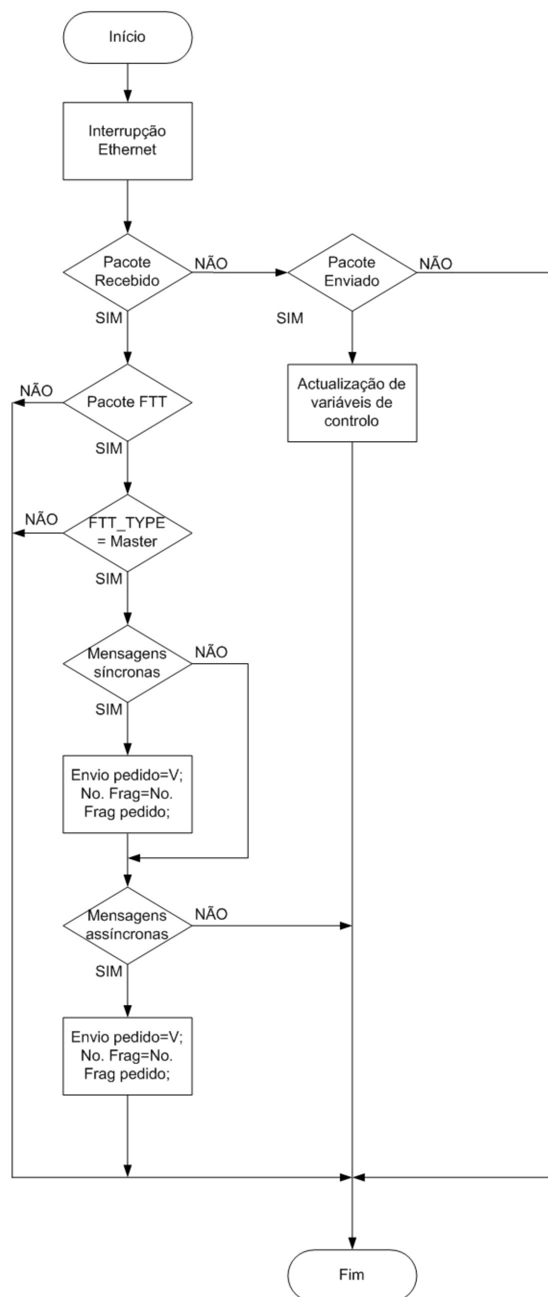


**Figura 5.7 – Algoritmo da rotina de atendimento de eventos *Ethernet***

Quando um evento é despoletado, a rotina é executada. Caso tenha sido recebido um pacote, é verificado se este é do tipo ETH_FTT_TYPE (valor em hexadecimal 0x8FF0), sendo descartado em caso negativo.

Sendo do tipo ETH_FTT_TYPE, é verificado se se trata de uma mensagem do tipo FTT_MST_MSG (valor 0), ou seja, se é uma *Trigger Message*, sendo descartada em caso negativo.

Se o pacote contiver pedidos de mensagens síncronas é marcado que existe um pedido de envio e qual o número do fragmento a enviar. Caso o pacote contenha pedidos de mensagens assíncronas é efectuado um procedimento idêntico.

No caso de o evento sinalizar o envio de um pacote as variáveis de controlo são actualizadas.

### 5.4.3 Acções para o envio de uma trama *Ethernet*

Para se enviar uma trama *Ethernet*, são necessários seguir alguns passos.

Zona de programa que deseja enviar uma trama
- ⇨ É chamada a função IssueWrite()
  - ⇨ É chamada a função adi_dev_Write( )

Como muitas vezes acontece, a programação segue uma forma modular, em que se vão criando camadas de abstracção de código. Enquanto que a função IssueWrite() pretende abstrair o programador de toda a preocupação de preparação de dados aquando do envio de uma trama, a função adi_dev_Write() pretende abstrair qualquer forma de comunicação com *hardware* em que a comunicação é feita através de *buffer*. Seja uma *interface Ethernet*, uma placa de som, etc, todas elas usariam a adi_dev_Write() para escrever dados. Já a função IssueWrite() é específica para a placa de rede.

A função adi_dev_Write() faz parte da biblioteca do sistema, não carecendo de alterações. A função IssueWrite() já existia mas foi alterada, de forma a preencher os dados de controlo de envio da informação da maneira desejada.

A função IssueWrite tem o seguinte cabeçalho:
**int IssueWrite(ADI_DEV_DEVICE_HANDLE lan_handle,char * overhead, char *data_to_send, unsigned int number_bytes)**
Em que o 1º parâmetro é o identificador associado à *interface* de rede, de seguida um ponteiro para uma região de memória com o cabeçalho da trama, depois um ponteiro para uma zona de memória que contém os dados propriamente ditos, e por fim o número de *bytes* de dados a serem enviados, contando deste o endereço de origem, até ao FCS exclusive.

A função adi_dev_Write tem o seguinte cabeçalho:
**u32 adi_dev_Write (ADI_DEV_DEVICE_HANDLE DeviceHandle, ADI_DEV_BUFFER_TYPE BufferType, ADI_DEV_BUFFER *pBuffer)**
Em que o 1º parâmetro é o identificador associado à *interface* de rede, o seguinte discrimina que tipo de buffer é (uma ou duas dimensões) e por fim, o *buffer* propriamente dito, que contém a trama a ser enviada, e dados necessários ao seu envio.

Para se proceder ao envio de uma trama, é necessário preencher uma série de campos, e estes estão definidos pelo tipo ADI_ETHER_FRAME_BUFFER, que está discriminado no ficheiro "C:\Program Files\Analog Devices\VisualDSP 5.0\Blackfin\include\drivers\ethernet\ADI_ETHER.h" e a sua definição é a seguinte:

```
typedef struct adi_ether_frame_buffer {
        u16         NoBytes;        // the no. of following bytes
        u8          Dest[6];        // destination MAC address
        u8          Srce[6];        // source MAC address
        u8          LTfield[2];     // length/type field
        u8          Data[0];        // payload bytes
} ADI_ETHER_FRAME_BUFFER;
```

Em que são respectivamente: o número de *bytes* que a trama irá conter, o endereço de destino, o endereço de origem, o código que singulariza o protocolo *Ethernet* utilizado e por fim um ponteiro para os dados a serem enviados.

Portanto o que acontece, é que o programador chama a função IssueWrite() tendo como parâmetro de entrada o ponteiro do identificador da *interface* de rede, os ponteiros para o cabeçalho e zona de dados, e o tamanho da trama a ser enviada.

Dentro da função chamada, os campos são preenchidos: o endereço de destino é lido da placa do *hardware* e o de destino é preenchido com o endereço de *broadcast.*O tipo é preenchido com o valor ETH_FTT_TYPE e os dados são copiados para a respectiva zona.

No final destes passos, é chamada a função adi_dev_Write() com os seguintes parâmetros: identificador de rede, valor para *buffer* unidimensional, e zona do tipo ADI_ETHER_FRAME_BUFFER previamente preenchida.

### 5.4.4 Acções para a recepção de uma trama *Ethernet*

Analogamente, existe um mecanismo de utilização semelhante par receber uma trama *Ethernet*. Enquanto que no envio, é preenchido um *buffer*, e chamada a função IssueWrite(), para recepção o princípio de funcionamento já não é o mesmo. É primeiro alocado o espaço, e só depois é preenchido com os dados recebidos. Para tal é utilizada a função IssueRead() com o seguinte cabeçalho:

**int IssueRead(ADI_DEV_DEVICE_HANDLE lan_handle);**

Em que o 1º parâmetro é o identificador associado à *interface* de rede.

Durante a inicialização, a função IssueRead() é chamada NO_RCVES vezes, sendo que este número define o número máximo de *buffers* dedicados exclusivamente à recepção de tramas *Ethernet*. Este número é o compromisso entre minimizar o espaço em memória dedicado à recepção de tramas, e ser grande o suficiente para poder acomodar várias recepções num curto espaço de tempo, caso hajam. No fundo o que a função IssueRead() faz, é atribuir espaço em memória para que a *interface Ethernet* possa guardar os dados recebidos. Tal acontece pois a função adi_dev_Read() aloca dinamicamente espaço para um *buffer* e passa o ponteiro à camada inferior.

A função adi_dev_Read() com o seguinte cabeçalho:

**u32 adi_dev_Read(ADI_DEV_DEVICE_HANDLE DeviceHandle, ADI_DEV_BUFFER_TYPE BufferType, ADI_DEV_BUFFER *pBuffer);**

Em que o 1º parâmetro é o identificador associado à *interface* de rede, o seguinte discrimina que tipo de buffer é (uma ou duas dimensões) e por fim, o *buffer* propriamente dito, alocado dinamicamente, que é reservado e eventualmente conterá a trama a ser recebida e outros dados relevantes para recepção da mesma.

Zona de programa responsável pela inicialização de zona para recepção

- ⇨ É chamada a função IssueRead()
  - ⇨ É chamada a função adi_dev_Read()

} **Repetido NO_RCVES vezes**

Estando os *buffers* alocados, quando chega uma trama válida à *interface Ethernet* (bem construída e destino coincidente com o seu endereço), o *buffer* é preenchido e passado à camada superior. É despoletada uma interrupção, que faz com que a função LanCallBack() seja chamada, tendo como argumentos de entrada o *buffer* que acabou de chegar, e o evento que despoletou a chamada, que é do tipo

ADI_ETHER_EVENT_FRAME_RCVD. É nesta função LanCallBack() que o tratamento da trama recebida é feito, conforme ilustrado na Figura 5.7.

### 5.4.5 Mecanismo de troca de *buffers*

Para que não haja corrupção de dados, em casos de escritas e leituras simultâneas, foi implementado um mecanismo de gestão de *buffers*. A problemática que levou à criação deste mecanismo de gestão de *buffers* será explicada com maior detalhe na secção 5.5.4, pois numa fase inicial da implementação da *stack* não será necessário este mecanismo. Nesta secção será descrito o algoritmo que o implementa.

Quando é pedido um *buffer*, o mecanismo começa por verificar se a acção que requer o *buffer* é uma escrita ou uma leitura.

O algoritmo é mostrado de uma forma condensada na Figura 5.8 para comodidade do leitor, e para ser possível separar o diagrama algo extenso. Na Figura 5.9 apresenta-se o algoritmo representado pelo rectângulo 1 (escrita), e na Figura 5.10 o algoritmo representado pelo rectângulo 2 (leitura). O tracejado a negrito representa o respectivo rectângulo na forma de algoritmo condensado.

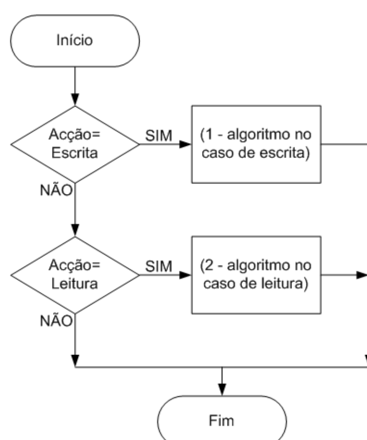


**Figura 5.8 – Algoritmo condensado da gestão de *buffers***

No caso da escrita, se o último *buffer* escrito tinha sido o número 2, é verificado se o número 1 está disponível para alocação e caso esteja, é marcado como passando a estar ocupado e o ponteiro para esse *buffer* é retornado.

Se o *buffer* 1 estiver ocupado, é verificado se o *buffer* 2 está disponível para alocação e em caso afirmativo, este é marcado como ocupado e o ponteiro para o mesmo é retornado. Para o caso de nenhum dos *buffers* estar disponível é retornado um código indicando a indisponibilidade de *buffer* para escrita.

Se o último *buffer* escrito não tiver sido o número 2, é verificado se foi o número 1. De notar que inicialmente é forçado um destes valores, de modo a que seja sempre uma destas duas situações. Em caso afirmativo, é verificado se algum dos *buffers* está disponível. Se algum deles estiver, é marcado como ocupado e é retornado o ponteiro para o mesmo. De notar que é dada preferência ao *buffer* 2 pela regra de alternância de *buffer*. Caso nenhum dos dois esteja disponível é retornado o código indicando indisponibilidade de *buffer* para escrita.

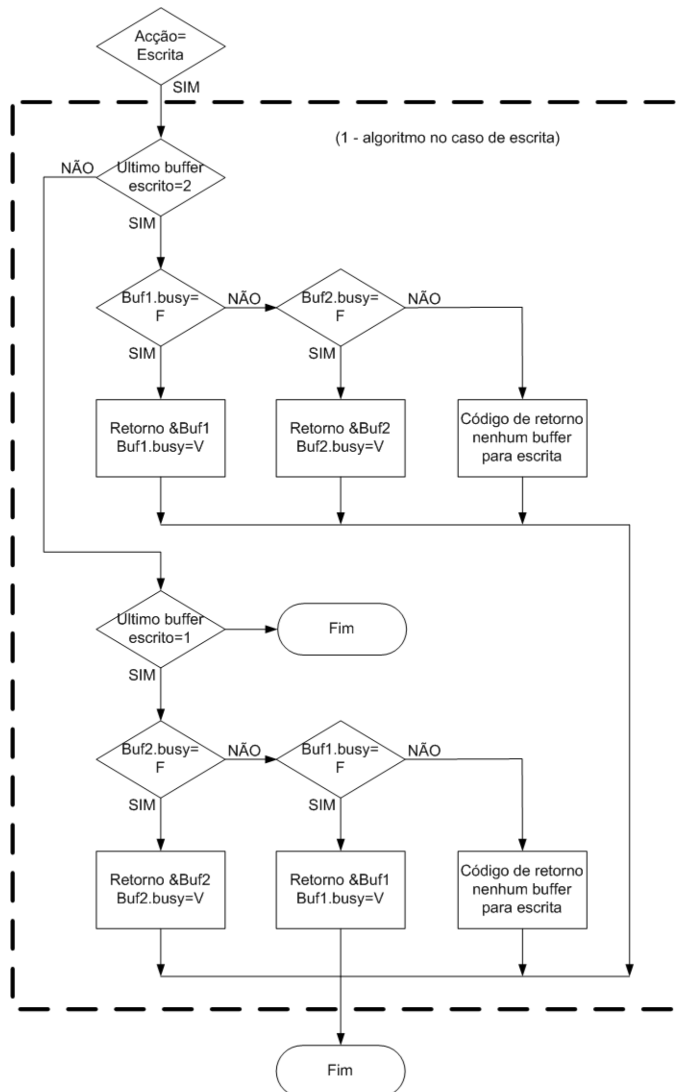


**Figura 5.9 – Algoritmo da gestão de *buffers* (escrita)**

Caso a acção a executar seja uma leitura, é verificado se os dados que constam do *buffer* 1 são novos. Entendem-se por novos, dados adquiridos e ainda não enviados pela rede. Sendo novos, é verificado se este *buffer* se encontra livre e, se assim estiver, é marcado como ocupado e retornado o ponteiro para o mesmo. Em caso de este estar ocupado é verificado se o *buffer* 2 está livre. Caso esteja livre, é marcado como ocupado e é retornado o seu ponteiro. Caso também não esteja disponível é retornado o código de indisponibilidade de *buffers* para leitura.

Caso os dados do *buffer* 1 não sejam novos, é verificado se os do *buffer* 2 o são. Sendo novos, é escolhido o *buffer* que estiver livre e este é marcado como ocupado e é retornado o ponteiro para o mesmo. De notar que é dada preferência ao *buffer* 2 pela regra de alternância de *buffer*. Caso nenhum dos dois esteja disponível é retornado o código indicando indisponibilidade de *buffer* para leitura.

No caso de nenhum dos *buffers* conter dados novos, é verificado qual o último *buffer* que foi escrito, pois é dada preferência ao *buffer* que contiver dados mais recentes, mas que em caso de indisponibilidade do mesmo, o outro será escolhido. A menos que os dois estejam ocupados, caso em que é retornado o código de indisponibilidade de *buffer* para leitura.

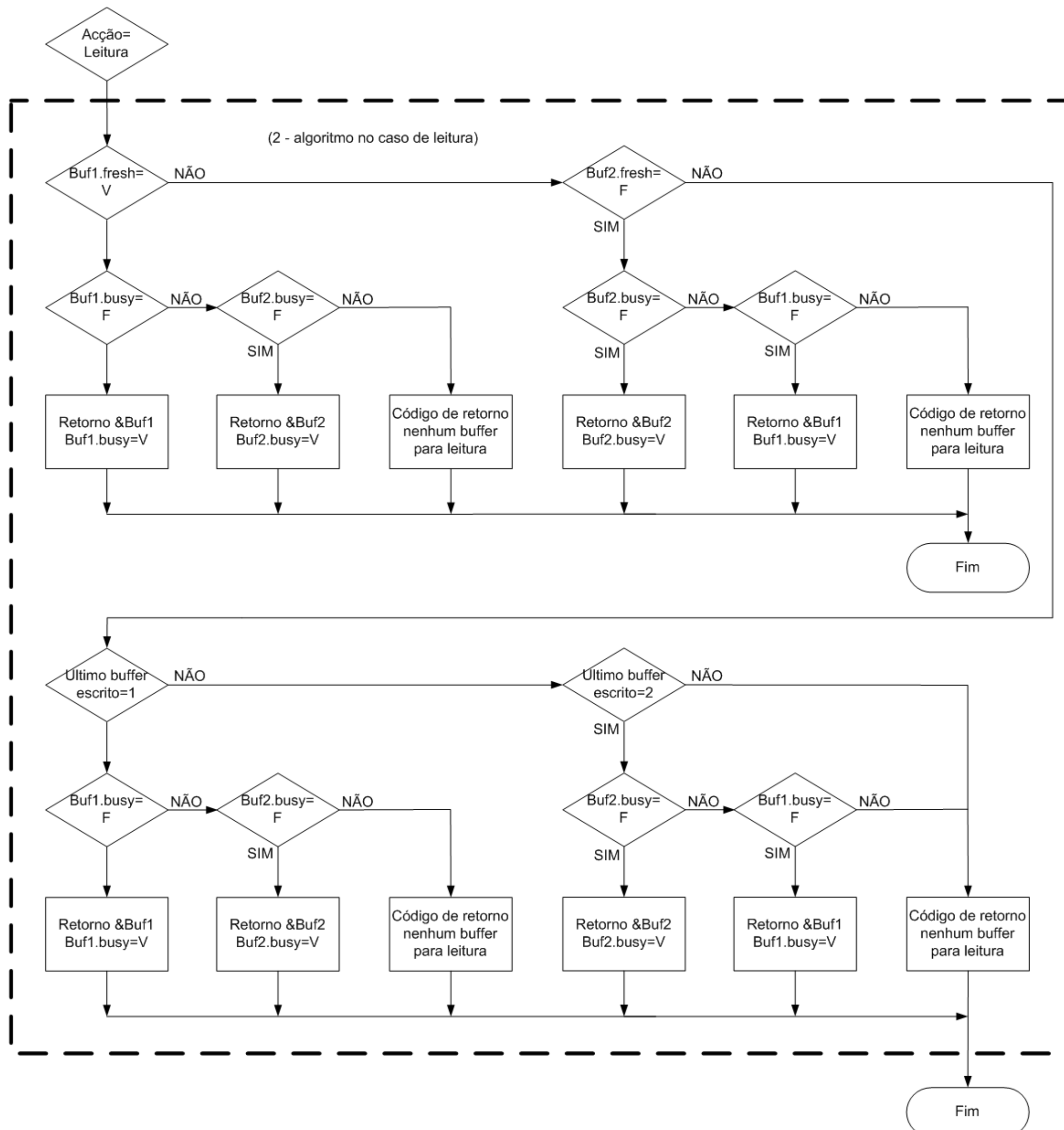


**Figura 5.10 - Algoritmo da gestão de *buffers* (leitura)**

É ainda importante referir que todo este mecanismo é executado em exclusão mútua, para impedir a corrupção dos dados.

## 5.5 Descrição dos testes

Estando descrito o *software* que implementa a *stack* e o restante necessário para o seu funcionamento, nesta secção é apresentado o *software* criado para testar e validar a mesma.

Do lado da câmara (nó *slave*), foi desenvolvida uma aplicação de teste que emula a aquisição de uma imagem com uma resolução de 640 por 480 pixéis e 8 bits por pixel de profundidade de cor (escala de cinzentos). Não foram utilizadas capturas reais devido ao facto de a empresa que ficou responsável por desenvolver o *hardware* e bibliotecas de aquisição não os ter disponibilizado em tempo útil.

A interligação entre o emulador do nó *master* e a câmara foi efectuada por meio de um “*cross cable*”.

### 5.5.1 Experiência 1: Tempo de resposta

Este primeiro teste consiste em enviar uma *trigger message* FTT-SE do *master* para o *slave*, indicando a transmissão de uma mensagem síncrona específica. Este teste permite avaliar a latência da reposta do nó *slave*, que é essencial para estimar a duração da janela que medeia entre a transmissão da TM e o início efectivo da janela síncrona, que é um dos parâmetros de configuração fundamentais do FTT-SE.

#### 5.5.1.1 Master

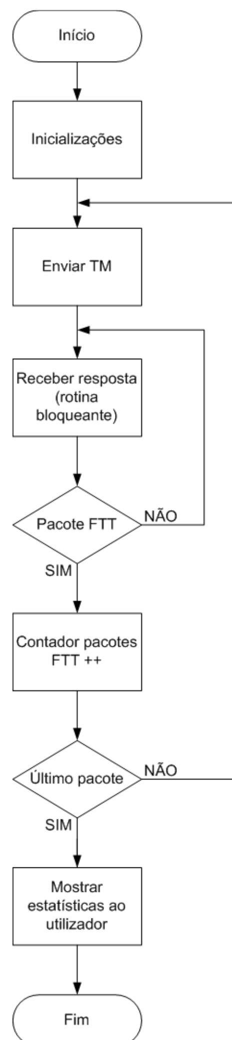


**Figura 5.11 – Diagrama de *software* do *master* para a experiência 1**

A Figura 5.11 explica o funcionamento do software do nó *master* no programa 1. Tal como é observável, o nó começa por efectuar as inicializações necessárias, entrando posteriormente em ciclo.

No início de cada ciclo o *master* começa por enviar uma *trigger message* e bloquear à espera de resposta do *slave*. Á recepção de um pacote é verificado se este é do tipo FTT; se não for, este volta ao estado bloqueante, esperando por um novo pacote; se for, então incrementa o contador de pacotes e verifica se foi atingida a condição de paragem de ciclo, ou seja, pedido de último pacote.

Caso esta condição ainda não tenha sido atingida, este reinicia o ciclo enviando nova *trigger message*. Caso a condição seja atingida, este mostra as estatísticas ao utilizador e termina a sua execução.

#### 5.5.1.2 Slave

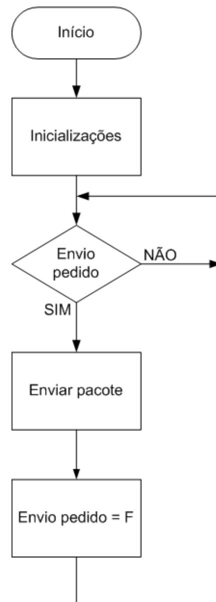


**Figura 5.12 – Diagrama de *software* do *slave* para a experiência 1**

Na Figura 5.12 podemos observar o funcionamento do nó *slave* implementado no programa 1. Este, à luz do que acontece com o *master*, começa por efectuar as inicializações. Posto isto, o nó fica à espera da recepção de uma *trigger message* para despoletar o envio de um pacote. Quando esta é recebida, o nó envia o pacote e retoma o estado de espera por nova *trigger message*.

### 5.5.2 Experiência 2: Integridade dos dados

Com a finalidade de verificar a correcção do armazenamento dos dados em memória e posterior envio, foi realizada uma experiência usando um mecanismo básico de *checksum* [37]. Entende-se por *checksum* como sendo "um bloco de dados de tamanho fixo adicionados com o propósito de detectar erros de transmissão ou armazenamento". Embora existam mecanismos mais elaborados, para garantir que os dados chegam correctamente após a sua transmissão, escolheu-se um checksum por ser simples e suficiente para o cenário em causa.

A cada bloco de dados a enviar foi acrescentado um bloco de 4 *bytes*.

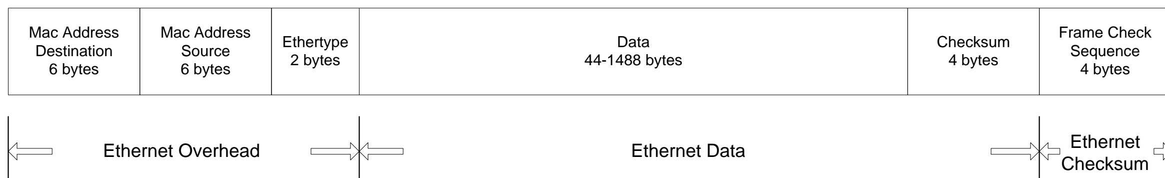


**Figura 5.13 - Estrutura do pacote Ethernet para a experiência 2**

Como se pode observar na Figura 5.13, ao pacote *Ethernet standard* foi acrescentado um campo de *checksum* dentro da zona reservada para os dados.

Tomemos como exemplo, o envio dos seguintes dados:

(os valores estão representados na base hexadecimal e são apresentados separados para comodidade do leitor)

Endereço MAC destino: 00 E0 22 FE B3 FF
Endereço MAC origem: 00 11 2F EC D8 6A
Ethertype: 8F F0
Dados: 01 23 45 67 89 AB CD EF FE DC BA 98

O *checksum* será calculado da seguinte forma:

checksum = 0
checksum = checksum + 0x01234567 = 0x01234567
checksum = checksum + 0x89ABCDEF = 0x8ACF1356
checksum = checksum + 0xFEDCBA98 = 0x189ABCDEE

Este seria o valor esperado, mas dado que a variável interna de contagem é de 4 *bytes*, existe *overflow*.

checksum = 0x89ABCDEE

Em relação ao *frame check sequence*, este é calculado por *hardware* e não é relevante abordar aqui essa questão.

O pacote a transmitir, ignorando a questão do tamanho mínimo do pacote *Ethernet*, seria então:

00 E0 22 FE B3 FF | 00 11 2F EC D8 6A | 8F F0 | 23 45 67 89 AB CD EF FE DC BA 98 | 89 AB CD EE | (*Frame Check Sequence*)

#### 5.5.2.1 Master

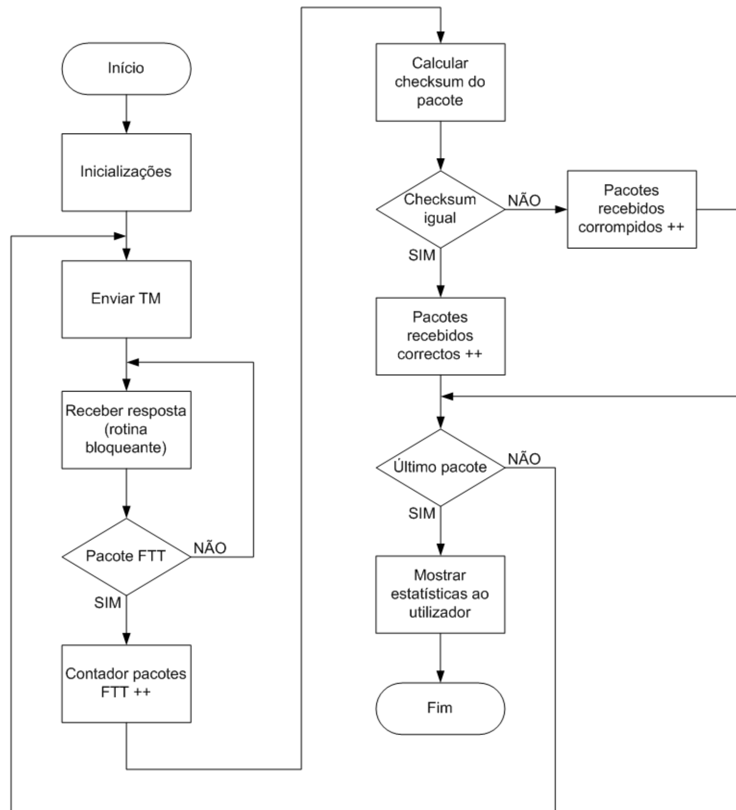


**Figura 5.14 - Diagrama de *software* do *master* para a experiência 2**

A Figura 5.14 esquematiza o funcionamento do nó *master* implementado no teste 2. À semelhança do que acontecia no teste 1, o nó efectua as inicializações e entra seguidamente em ciclo. Porém, neste caso, a adição do mecanismo de *checksum* introduz novos procedimentos.

Após receber um pacote e verificar que este é do tipo FTT, o *checksum* do pacote é calculado. Se este for igual ao proveniente do pacote é incrementado o contador de pacotes correctos, caso contrário, é incrementado o contador de pacotes corrompidos.

Após recepção do último pacote as estatísticas do teste são mostradas, finalizando assim a sua execução.

#### 5.5.2.2 Slave

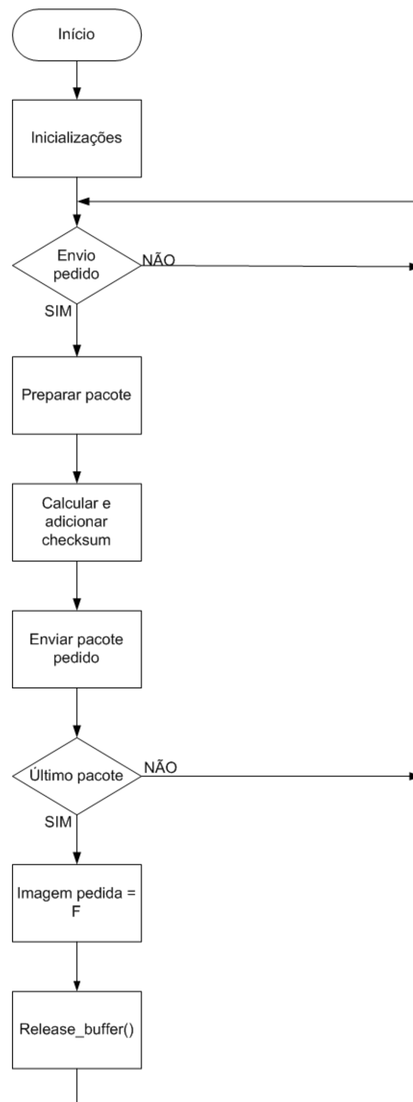


**Figura 5.15 - Diagrama de *software* do *slave* para a experiência 2**

O funcionamento do *slave* para o programa 2 é apresentado na Figura 5.15. Este começa por efectuar as inicializações necessárias e, de seguida, bloqueia à espera de uma *trigger message*. Quando esta é recebida, verifica se já tem algum *buffer* alocado e caso não tenha, trata de o alocar.

De seguida, prepara o pacote para envio, calcula e adiciona-lhe o *checksum*, e envia-o. Caso este não seja o último pacote a enviar, volta ao estado de bloqueio à espera por nova *trigger message*. Caso seja o último pacote, sinaliza uma *flag* indicando que não há pedido de imagem e liberta o *buffer*, voltando a um estado de espera por nova *trigger message*.

### 5.5.3 Experiência 3: Fragmentação e reagrupamento

Como se pode facilmente compreender, em muitas aplicações os dados gerados excedem o tamanho de um só pacote *Ethernet*. Nestes casos é preciso fragmentar os dados por vários pacotes. Após garantir, no teste anterior, que cada pacote individual é transmitido correctamente, neste teste testaram-se os mecanismos de fragmentação e reagrupamento.

Para a realização deste teste, o programa do lado do *master* foi alterado de modo a que fossem pedidos todos os fragmentos de um *buffer* (imagem) de uma forma sequencial. Foram feitos 100 pedidos de imagens, para ser possível o tratamento estatístico de bastantes ocorrências.

O software *master* implementado, além de pedir os pacotes na ordem correcta, verificava se os pacotes recebidos correspondiam ao pedido. No caso de tal não acontecer apareceria como mensagem de erro ao utilizador.

#### 5.5.3.1 Master

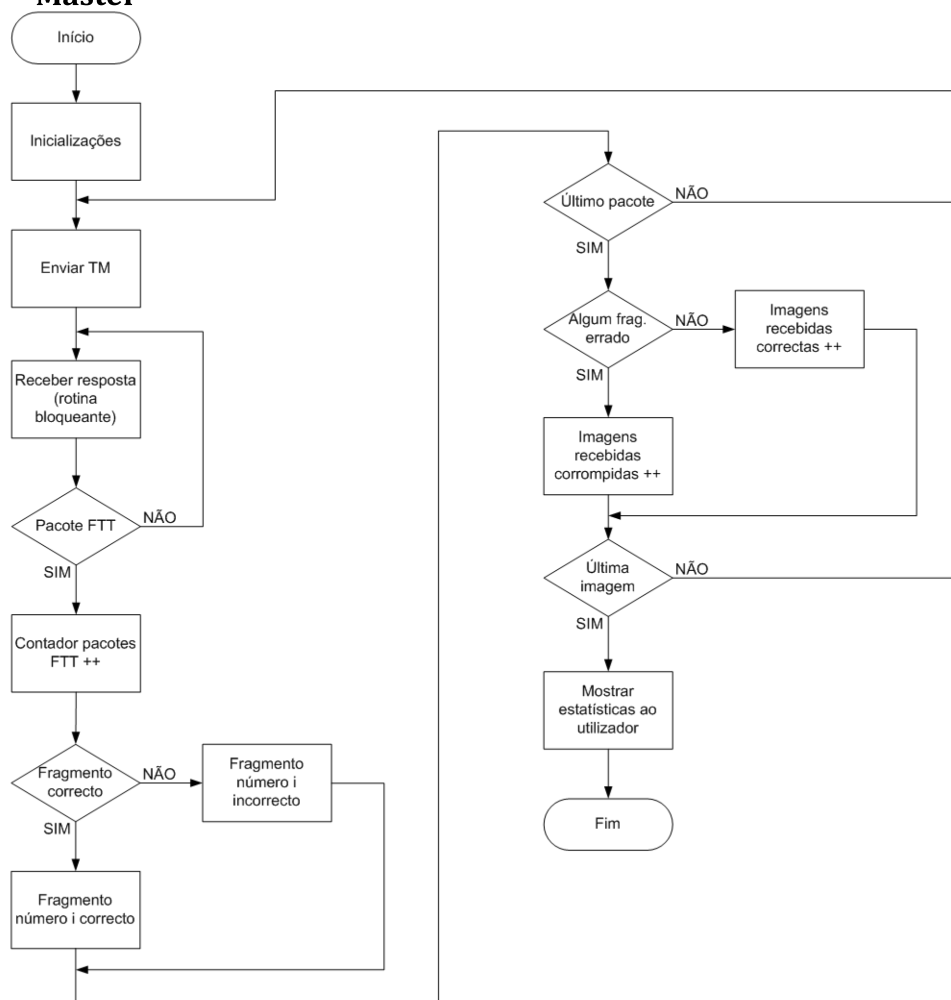


**Figura 5.16 – Diagrama de *software* do *master* para a experiência 3**

A Figura 5.16 apresenta o funcionamento do *master* relativo ao programa 3. Neste caso o nó começa por efectuar o procedimento normal de envio de *trigger message* e recepção de resposta do *slave*.

Aquando da recepção de um pacote FTT é verificado se o fragmento corresponde ao esperado e este é devidamente sinalizado. De seguida é verificado se o pacote recebido

corresponde ao último esperado. Se não for, reenvia nova *trigger message*, repetindo o ciclo.

Se for o último pacote, verifica se houve algum fragmento incorrecto. Em caso positivo é incrementado o contador de imagens recebidas corrompidas, caso contrário é incrementado o contador de imagens recebidas correctas.

Por último, este verifica se é a última imagem a receber, caso não seja, pede nova imagem enviando nova *trigger message*, se for a última, termina o programa mostrando as estatísticas ao utilizador.

#### 5.5.3.2 Slave

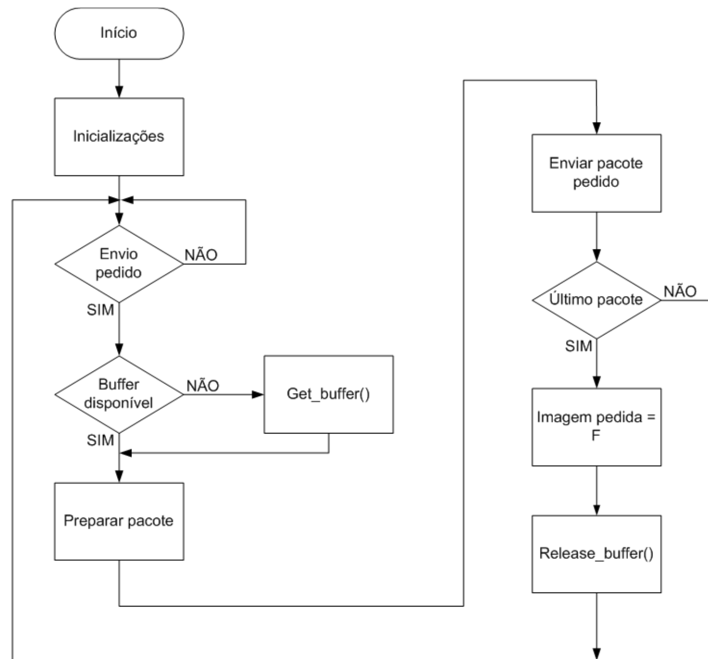


**Figura 5.17 – Diagrama de *software* do *slave* para a experiência 3**

O funcionamento do *slave* para o programa 3 é apresentado na Figura 5.17. O nó inicia a sua execução efectuando as normais inicializações e esperando pela recepção de uma *trigger message* com o pedido de envio de imagem. Ao recepcioná-la, verifica se tem *buffer* alocado e, caso não tenha, trata de o alocar.

De seguida, prepara e procede ao envio do pacote. No caso de ser o último pacote da imagem a enviar, o nó liberta o *buffer* e espera por novo pedido de imagem. Caso não seja o último pacote, este espera por nova *trigger message* para iniciar o envio do próximo pacote.

### 5.5.4 Experiência 4: Mecanismo duplo *buffer*

Como foi referido anteriormente, é necessário criar uma forma de armazenar em memória as imagens a serem transmitidas. Surge de imediato um problema de concorrência, pois não se pode alterar um *buffer* que está a ser enviado, sob pena de se corromperem os dados.

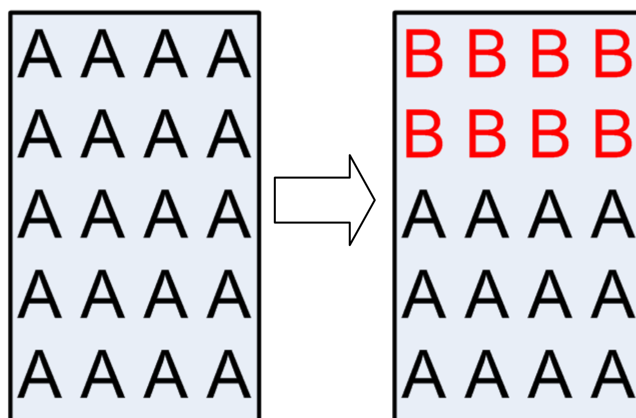


**Figura 5.18 - Exemplo do problema de um buffer sem acesso em exclusão mútua**

Observemos a Figura 5.18. O rectângulo representa um *buffer* e as letras representam os dados de uma imagem. No rectângulo à esquerda, vemos um *buffer* acabado de preencher totalmente com dados. Se no momento em que um *buffer* contém dados válidos, se começar a escrever dados de uma nova imagem nesse mesmo *buffer*, existe corrupção dos dados, conforme está representado na imagem à direita. Parte dos dados pertencem à nova imagem, outra parte à imagem anterior.

Para resolver este problema, foi implementado um mecanismo de dois *buffers*. Existe um árbitro que gere os dois *buffers*, para que estes sejam fornecidos em exclusão mútua e que impeça a corrupção de uma determinada imagem. Desta forma, é possível ler e escrever simultaneamente (ou dar essa impressão). Não é preciso esperar que a imagem em memória seja enviada pela rede para que possamos capturar uma nova imagem e vice-versa.

Neste teste, do lado do *slave*, o programa foi alterado para que além da informação contida nos dados, fosse disponibilizado o número do buffer utilizado para o armazenamento da imagem enviada.

Já do lado do *master* continua o pedido de 100 imagens. Apenas existe uma leve diferença, o programa disponibiliza no final de todos os pedidos, se os *buffers* utilizados foram sempre alternados ou não, e em quais houve a eventual falha, para facilitar o *debug*.

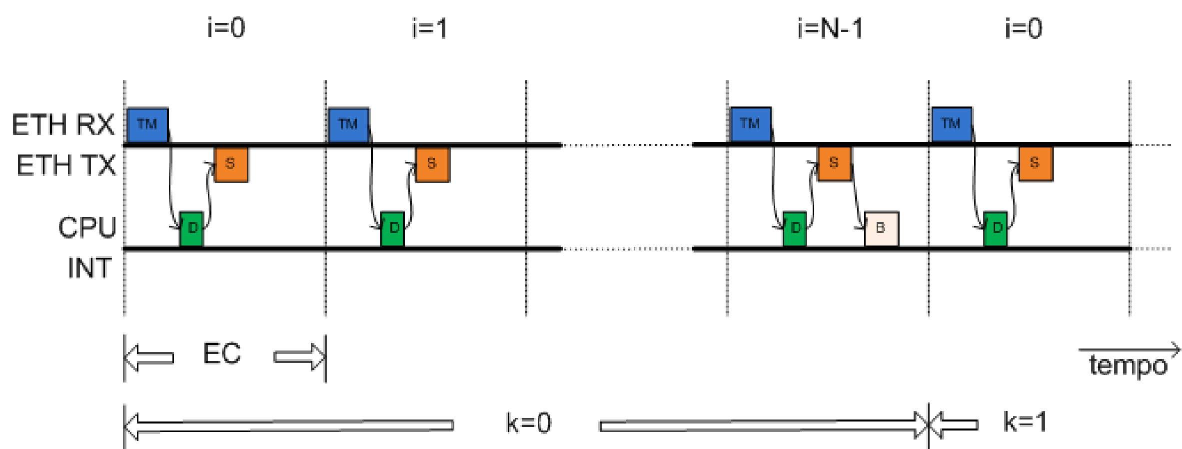

**Figura 5.19 - Diagrama temporal do teste 4: Mecanismo sequencial de troca de *buffers***

Na Figura 5.19, apresenta-se o diagrama temporal deste teste. Primeiro segue uma breve explicação da notação utilizada, e depois a explicação do teste propriamente dito.

A letra k representa o número da imagem a ser enviada, a letra i o número de fragmento dessa mesma imagem e N o número de pacotes necessários para enviar uma imagem completa.

ETH RX e ETH TX representam a actividade *Ethernet* do *slave*, quer de recepção, quer de transmissão, respectivamente.

CPU (*Central Processing Unit*) representa a actividade do CPU do *slave*, e INT representa o processamento de interrupções do mesmo.

EC significa *elementary cycle* (ciclo elementar do protocolo FTT-SE) e a letra B de *buffer*, simboliza o processamento necessário para adquirir uma nova imagem para que no próximo ciclo haja informação "fresca" pronta a ser enviada.

TM significa *Trigger Message* e D significa *decode*, representando genericamente o processamento que o *slave* tem de executar para descodificar a informação contida na TM. A letra S significa *send*, ou seja, representa a resposta do *slave* ao pedido feito na TM enviada pelo *master*.

Como se pode observar na Figura 5.19, o teste consiste em pedir sequencialmente todos os fragmentos de uma imagem. Quando se chega ao último fragmento, é preparada uma nova imagem para ser enviada.

Este teste foi concebido para averiguar se funcionava ou não o mecanismo de troca de *buffer* e em que medida tal afectava o desempenho como *slave* FTT-SE.

### 5.5.4.1 Master

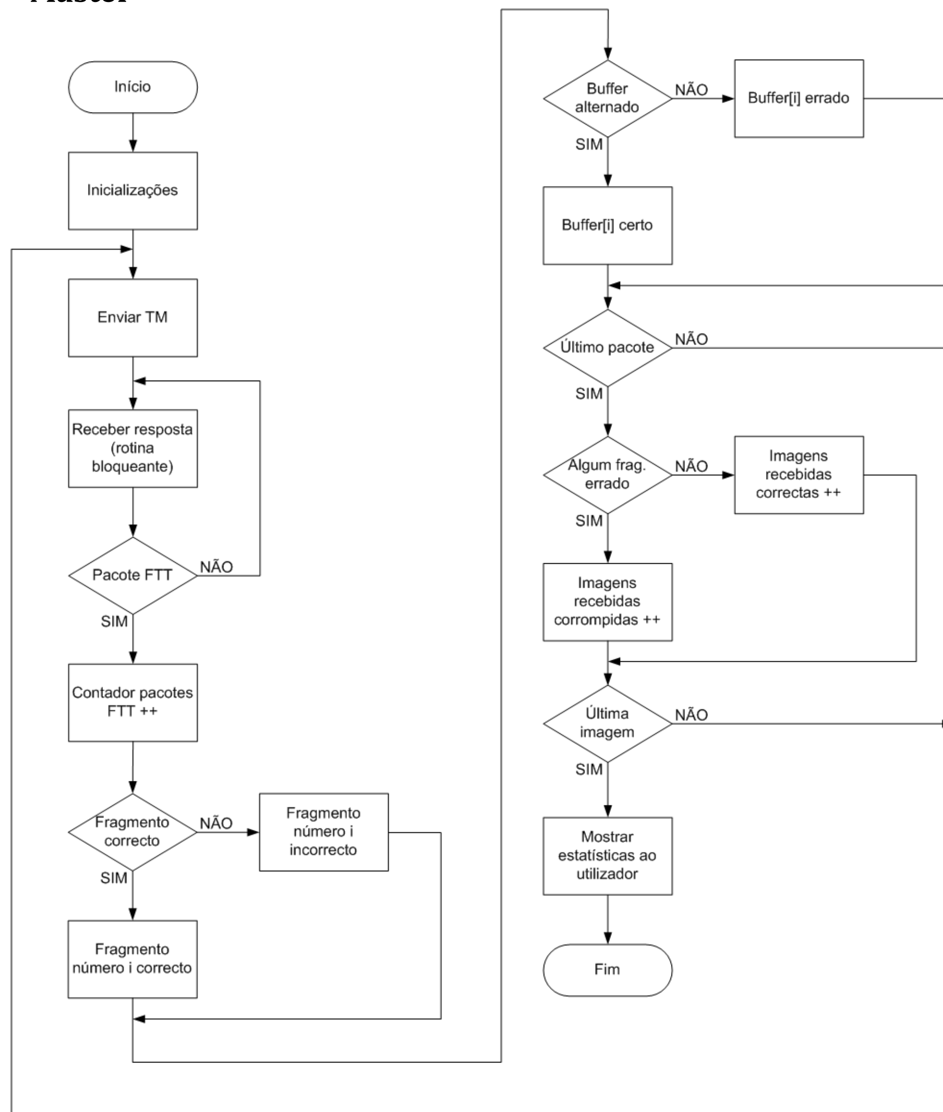


**Figura 5.20 - Diagrama de *software* do *master* para a experiência 4**

A Figura 5.20 apresenta o funcionamento do nó *master* para o programa 4. Ao iniciar a sua execução, este efectua as inicializações entrando de seguida num ciclo de pedido de imagens.

Cada ciclo é iniciado, começando por enviar uma *trigger message*. A cada pacote recebido é verificado se este corresponde ao fragmento de imagem correcto e este é devidamente sinalizado. Seguidamente sinaliza se houve a correcta alternância do *buffer*.

Na chegada do último pacote, o *master* verifica se a imagem tem algum fragmento errado e, se tal acontecer, incrementa o contador de imagens corrompidas. Caso contrário, incrementa o contador de imagens correctas.

Chegando à última imagem, mostra as estatísticas ao utilizador e termina a sua execução.

#### 5.5.4.2 Slave

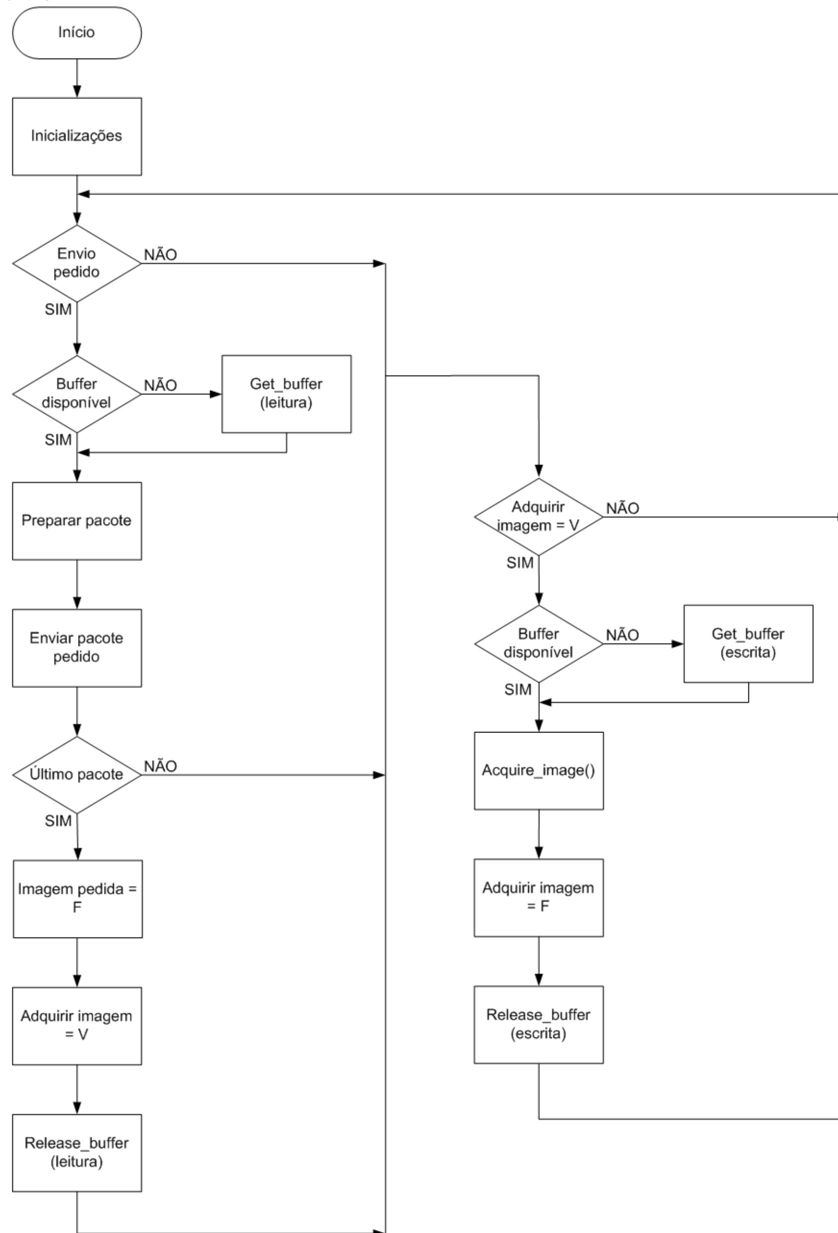


**Figura 5.21 - Diagrama de *software* do *slave* para a experiência 4**

A Figura 5.21 explica o funcionamento do nó *slave* para o programa 4. Ao longo da sua execução este vai respondendo às *trigger messages* enviadas pelo nó *master*.

Aquando da chegada desta, o nó verifica se tem *buffer* alocado e, caso não tenha trata de o alocar.

De seguida, trata da preparação e envio do pacote. Ao envio do último pacote o nó sinaliza que já não há pedido de imagem a tratar e assinala a necessidade de aquisição de nova imagem libertando seguidamente o *buffer*. Posteriormente, adquire nova imagem e volta a esperar por novo pedido.

### 5.5.5 Experiência 5: Mecanismo de troca de *buffers* com interferência em N/2

O teste anterior apenas garantia que, em caso de sucesso, o sistema alternava entre os dois *buffers* sequencialmente. Quando acabasse a operação de leitura ou escrita sobre o *buffer* 1, o próximo *buffer* a ser utilizado seria o 2 e vice-versa, salvo situações especiais. Não existia o problema de concorrência propriamente dito, apenas estava a ser testado o mecanismo de troca.

Neste teste, o sistema iria ser posto à prova a uma concorrência controlada. Ou seja, a necessidade de leitura e de escrita não iria ocorrer aleatoriamente, mas em tempos bem determinados, de forma a ter algum espírito crítico perante os resultados obtidos.

No teste anterior, uma nova imagem era preparada após "soltar" o *buffer* previamente usado. Apesar dos resultados apresentados mostrarem que os *buffers* eram usados de forma alternada, poder-se-ia dar o caso de o mecanismo não funcionar bem quando um *buffer* estava a ser usado.

É precisamente o intuito deste teste: aceder a um *buffer* em modo de escrita enquanto um *buffer* está a ser utilizado em modo de leitura/envio. N/2 significa que a interferência é criada a meio de o envio de um *buffer*, sendo N o número de pacotes necessários para o envio do mesmo.

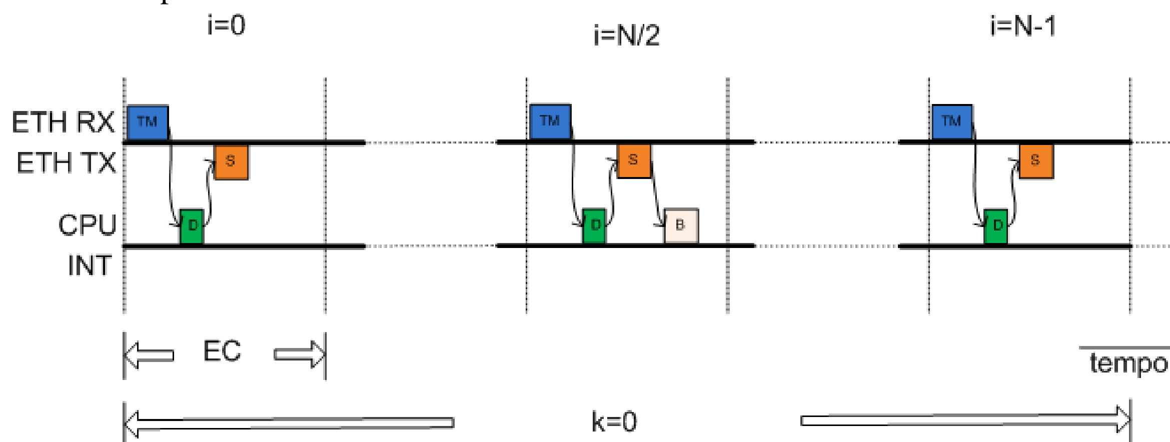


**Figura 5.22 - Diagrama temporal do teste 5: Mecanismo de troca de *buffers* com interferência em N/2**

Como se pode observar na Figura 5.22, o acesso ao *buffer* para escrita, vai acontecer precisamente a meio do envio de uma imagem existente.

#### 5.5.5.1 Master

O funcionamento do nó *master* para a experiência 5 é em tudo idêntico ao do nó *master* utilizado na experiência 4.

#### 5.5.5.2 Slave

A Figura 5.23 demonstra o funcionamento do nó *slave* na experiência 5.

Aquando a recepção de uma *trigger message* este verifica se tem *buffer* alocado e, caso não tenha trata de o alocar.

De seguida trata da preparação e envio do pacote. Caso o pacote seja o número N/2 então sinaliza a necessidade de aquisição de nova imagem. Foi escolhido N/2, para se forçar a concorrência entre a escrita de uma nova imagem num *buffer* e o envio de um

outro *buffer* pela rede, criando assim um caso de utilização de um *buffer* para escrita e outro para leitura simultaneamente.

Quando é efectuado o envio do último pacote, a *flag* de imagem pedida é posta a falso e o *buffer* é libertado. No fim do envio de cada pacote é verificada a necessidade de aquisição de nova imagem e, esta é efectuada em caso afirmativo.

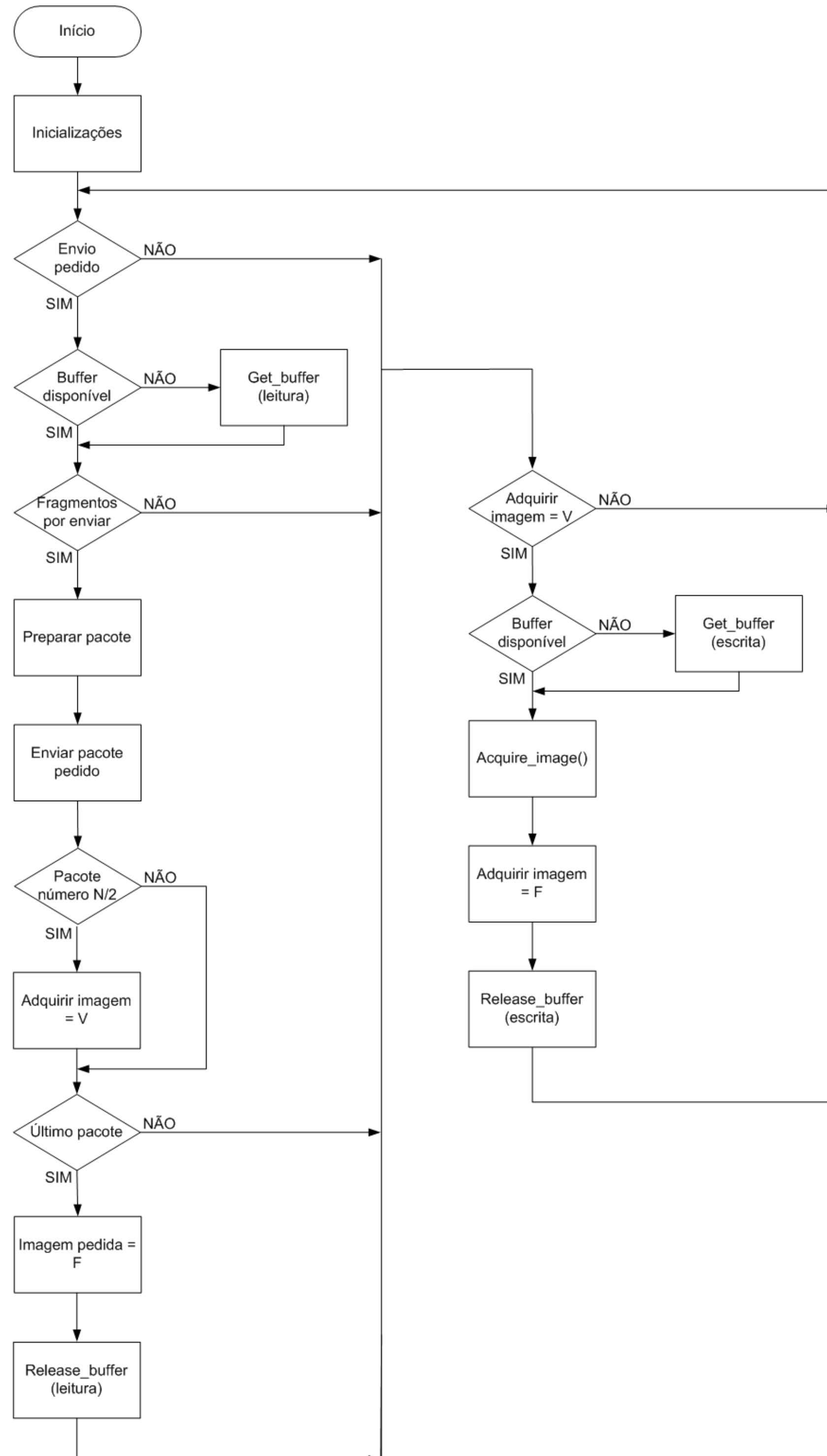


**Figura 5.23 - Diagrama de *software* do *slave* para a experiência 5**

### 5.5.6 Experiência 6: Mecanismo de troca de *buffers* com interferência de *timer*

Em todos os testes anteriores, o acesso a um novo *buffer* para escrita acontece sempre após o envio de uma mensagem, e sempre contido no seu *elementary cycle*.

Nesta secção põe-se à prova o sistema no caso em que a necessidade de escrita ocorra durante uma tarefa de leitura.

Após o envio do último fragmento, é armado um *timer* com um valor bem determinado. Após esse tempo, na rotina disparada pelo *timer*, é activada uma *flag* que sinaliza a necessidade de um novo *buffer*.

Repetindo este processo para vários valores, numa gama de 1ms, consegue-se varrer um ciclo elementar completo, podendo assim caracterizar o comportamento do sistema.

Dado a existência de duas linhas de execução, e devido também às várias possibilidades inerentes dos valores escolhidos para o *timer* uma de duas situações:

a) A activação do *timer* e o acesso ao buffer ocorrem dentro do próprio ciclo elementar;
b) A activação do *timer*, parte ou a totalidade do acesso ao buffer ocorrem num ciclo elementar seguinte.

Analisemos então em maior detalhe a situação a), o caso em que o processo responsável por novo *buffer* está completamente contido no próprio ciclo elementar. Pode então acontecer:

1. O tempo relativo que o sistema necessita para processar as suas tarefas somado ao tempo do *timer* é inferior a 1ms, e como tal o acesso ao *buffer* é feito nesse mesmo ciclo elementar (Figura 5.24);

2. O tempo relativo que o sistema necessita para processar as suas tarefas somado ao tempo do *timer* é ligeiramente superior a 1ms, ou seja, começa o acesso ao *buffer* mas este conjunto de instruções é interrompido (a chegada de um *frame* na rede é atendido em contexto de interrupções e com maior prioridade) e retomado logo após a conclusão das instruções que interromperam (Figura 5.25);

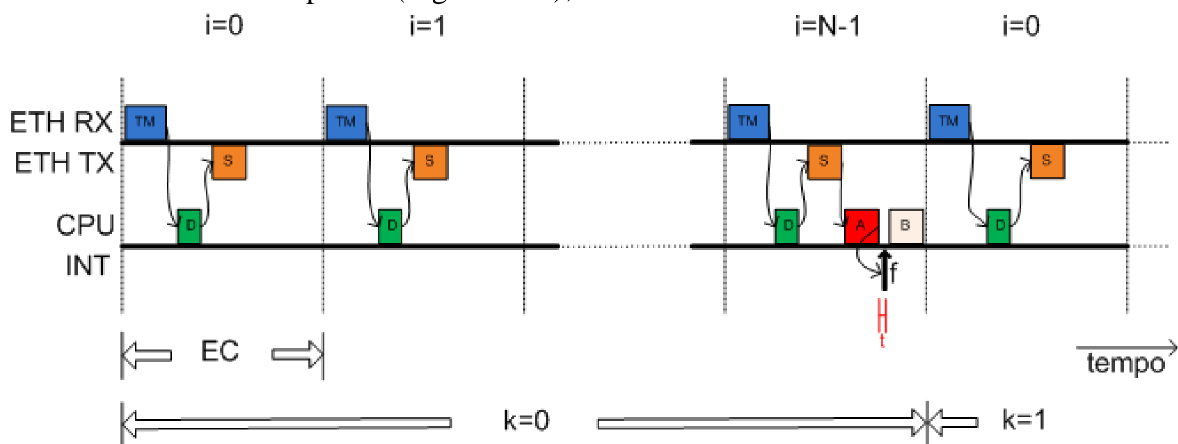


**Figura 5.24 - Diagrama temporal do teste 6: Mecanismo de troca de *buffers* com interferência de *timer* (1.)**

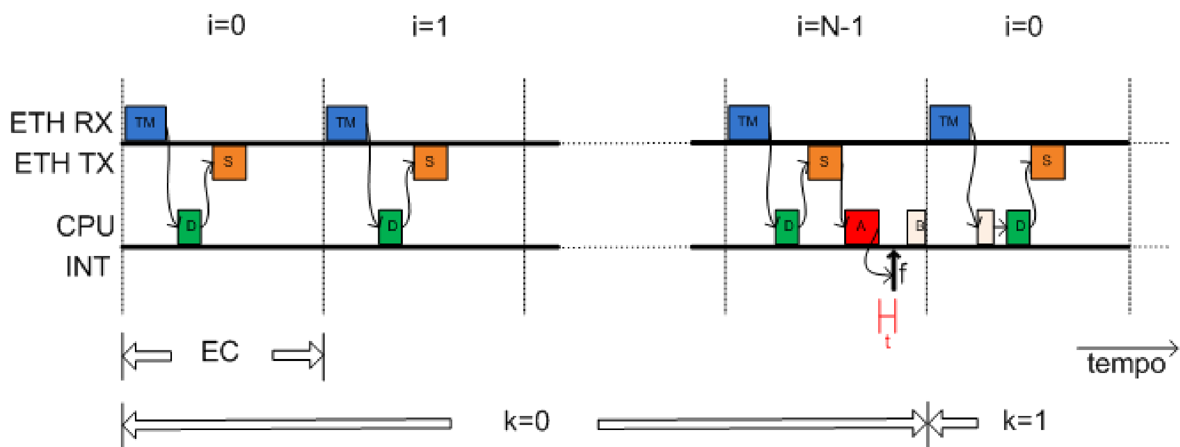

**Figura 5.25 - Diagrama temporal do teste 6: Mecanismo de troca de *buffers* com interferência de *timer* (2.)**

Na situação 1., a latência do ciclo elementar após o acesso ao buffer não é afectada. Já na situação 2. a latência é afectada. Dependendo do tempo de execução da tarefa, este tempo pode ou não ser significativo.

No caso da situação b), podem ocorrer as seguintes situações:

3. A activação da rotina do *timer* ser imediatamente a seguir à recepção da TM; a linha de execução principal pode ter mudado de contexto no momento em que se preparava para testar a eventual necessidade de requisição de um buffer para escrita (situação representada na Figura 5.26) ;

4. O mesmo que 3., mas aquando da mudança de contexto o programa estava na iminência de enviar um fragmento (Figura 5.27);

   (Nota: existem ainda as situações variantes à 3. e 4., em que a rotina do *timer* pode ter sido disparada imediatamente antes da recepção da TM, ou ocorrer durante a recepção da TM, e como esta é executada com as interrupções desligadas, ser adiada para o fim da recepção da TM. O resultado temporal iria ser o mesmo.)

5. Após ter sido processada a TM, e se ter começado a reagir à mesma, é activado a rotina de atendimento ao *timer*, o acesso ao buffer ocorre rápido o suficiente antes de se começar o envio da mensagem de resposta. Ou até mesmo com a activação da rotina do *timer* logo após o bloco de *decode* (Figura 5.28);

6. Situação semelhante à anterior, mas a execução da parte responsável pelo acesso ao buffer não é suficientemente rápida, e é interrompida pelo envio da reposta (Figura 5.29);

7. A activação da rotina de atendimento ao *timer* ocorre após o envio da mensagem (Figura 5.30).

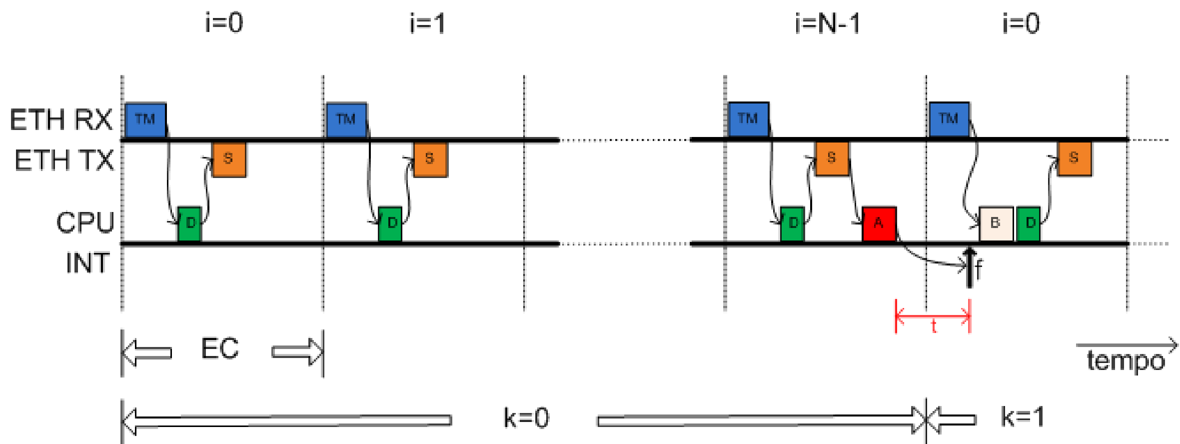


**Figura 5.26 - Diagrama temporal do teste 6: Mecanismo de troca de *buffers* com interferência de *timer* (3.)**

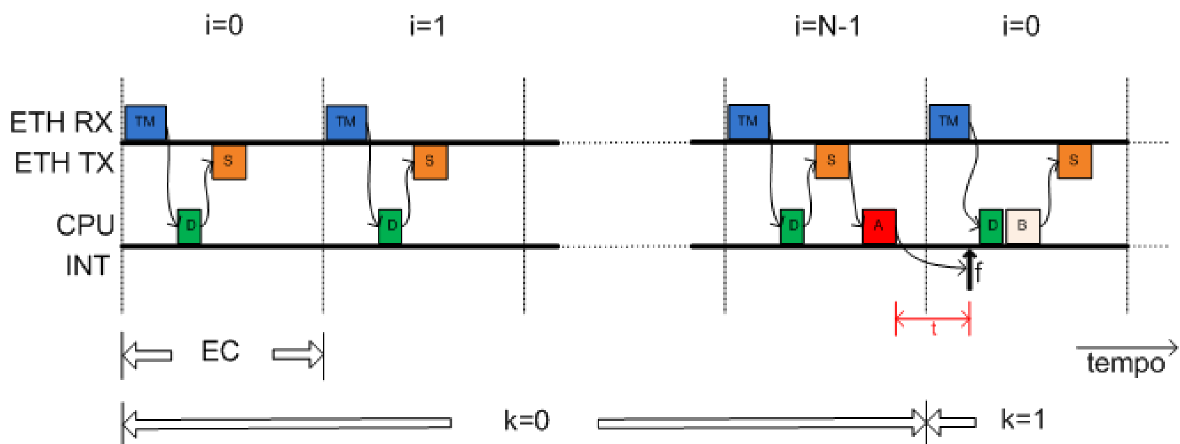


**Figura 5.27 - Diagrama temporal do teste 6: Mecanismo de troca de *buffers* com interferência de *timer* (4.)**

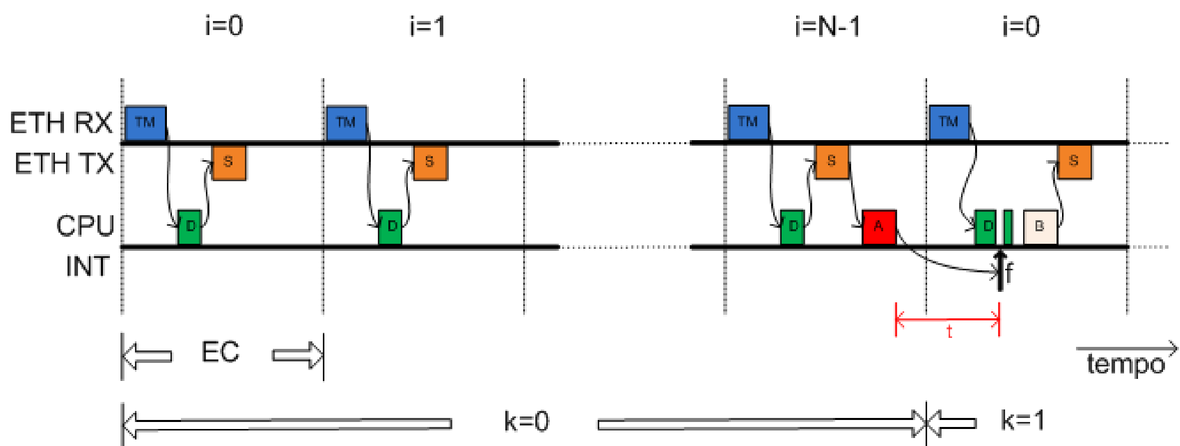


**Figura 5.28 - Diagrama temporal do teste 6: Mecanismo de troca de *buffers* com interferência de *timer* (5.)**

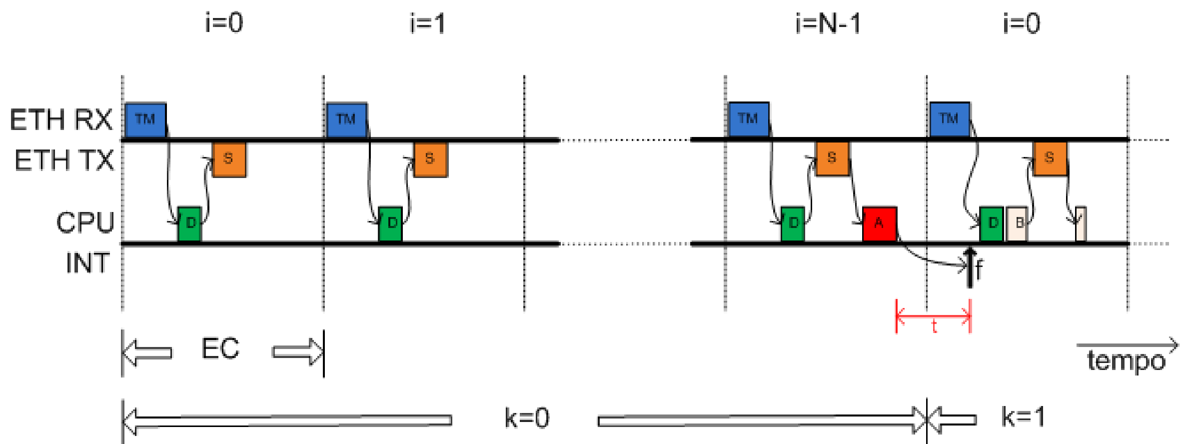

Figura 5.29 - Diagrama temporal do teste 6: Mecanismo de troca de *buffers* com interferência de *timer* (6.)

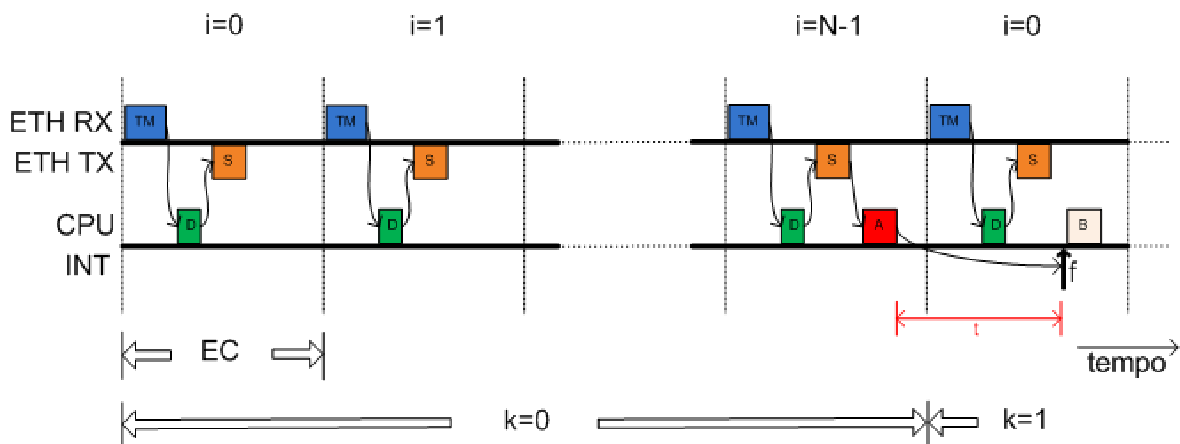

Figura 5.30 - Diagrama temporal do teste 6: Mecanismo de troca de *buffers* com interferência de *timer* (7.)

Como facilmente se pode observar, há situações em que a latência sai afectada, e outras não. Nas situações 1 e 7, a latência não sai afectada. O acesso ao *buffer* para escrita está suficientemente bem isolado e não faz com que haja atraso no envio da mensagem.

Já nas restantes situações, há um aumento de latência, pelo facto de um bloco de código ser executado pelo meio do que seria de esperar à partida. É precisamente por causa deste fenómeno que foi criado este teste, para ver se o sistema lida bem com a interferência criada pelo acesso de leitura e escrita no *buffer*, ainda que de forma aparente.

### 5.5.6.1 Master

O funcionamento do nó *master* para a experiência 6 é em tudo idêntico ao do nó *master* utilizado nas experiências 4 e 5.

### 5.5.6.2 Slave

O funcionamento do nó *slave* para a experiência 6 é apresentado na Figura 5.31.

A cada recepção de uma *trigger message*, o nó verifica se já tem *buffer* alocado e, em caso negativo, trata de o alocar. De seguida prepara o pacote e procede ao seu envio.

Quando envia o último pacote, sinaliza que não existe pedido de imagem a atender, arma o *timer* que despoletará a sinalização de necessidade de aquisição de imagem e liberta o *buffer*.

No fim do envio de cada pacote, é verificada a necessidade de aquisição de imagem e procede à mesma em caso afirmativo.

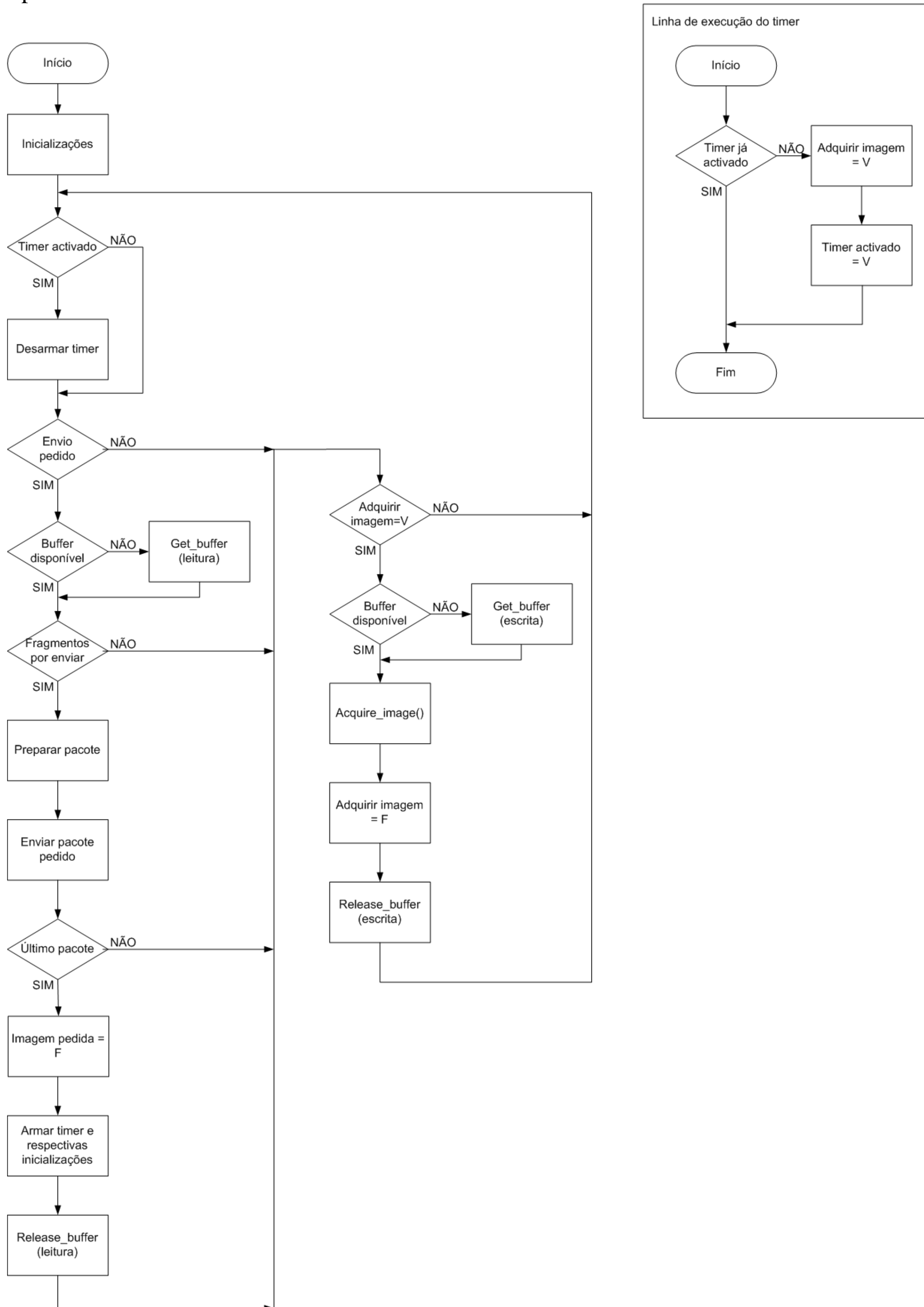


**Figura 5.31 - Diagrama de *software* do *slave* para a experiência 6**

## Capítulo 6

# 6 Resultados experimentais

Foi realizado um conjunto extensivo de testes experimentais com o objectivo de verificar a performance e a correcção da implementação da *stack* FTT-SE realizada no âmbito desta dissertação. Os testes incidiram sobre cada um dos componentes que integram a *stack*. Os testes realizados e respectivos resultados são apresentados neste capítulo.

## 6.1 Experiência 1: Tempo de resposta

A experiência realizada para o teste do tempo de resposta é a descrita na secção 5.5.1.

Os dados a seguir apresentados foram baseados numa experiência em que se mediu o tempo de resposta a 20700 TMs.

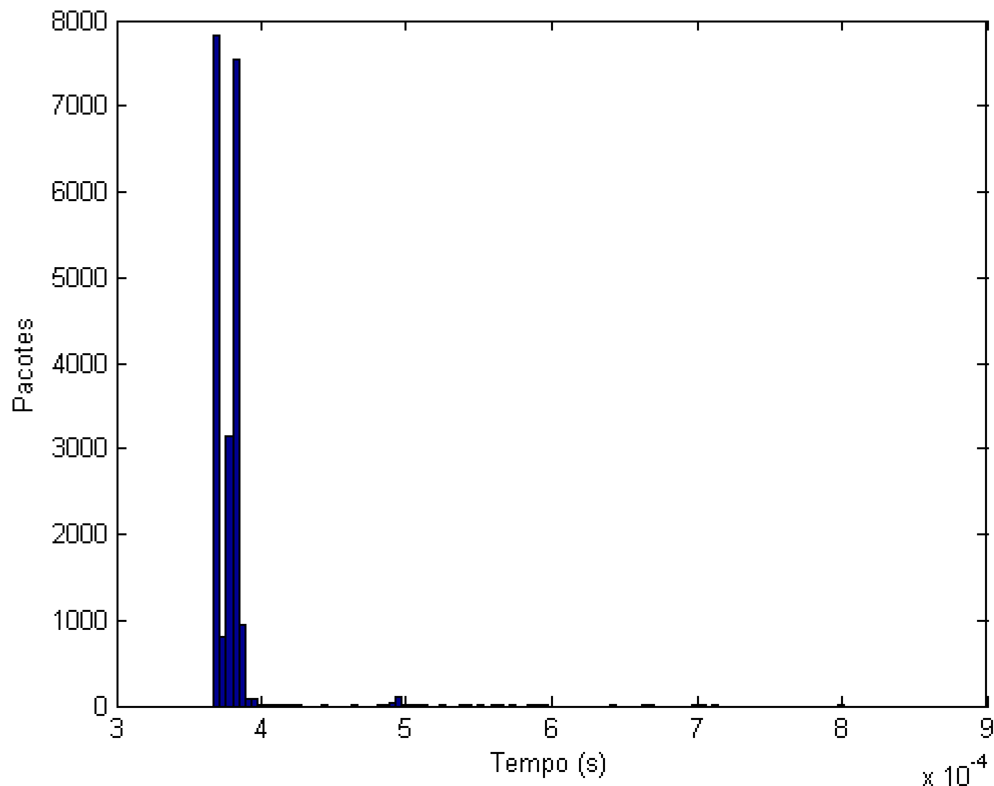


**Figura 6.1 – Latência entre a *Trigger Message* do *master* e resposta do *slave***

Como se pode observar na Figura 6.1, o tempo de resposta é em norma um pouco inferior a 400us, podendo no entanto atingir cerca 800us. Uma vez que a janela de guarda é função do tempo máximo de cada dispositivo, trata-se de um valor relativamente elevado e que limita em termos práticos a duração mínima do EC. Uma possível justificação para este valor algo elevado poderá ser o facto de a implementação actual ser baseada em *polling*. Este será um dos aspectos a melhorar numa futura evolução do trabalho realizado, dada a sua importância no funcionamento do protocolo.

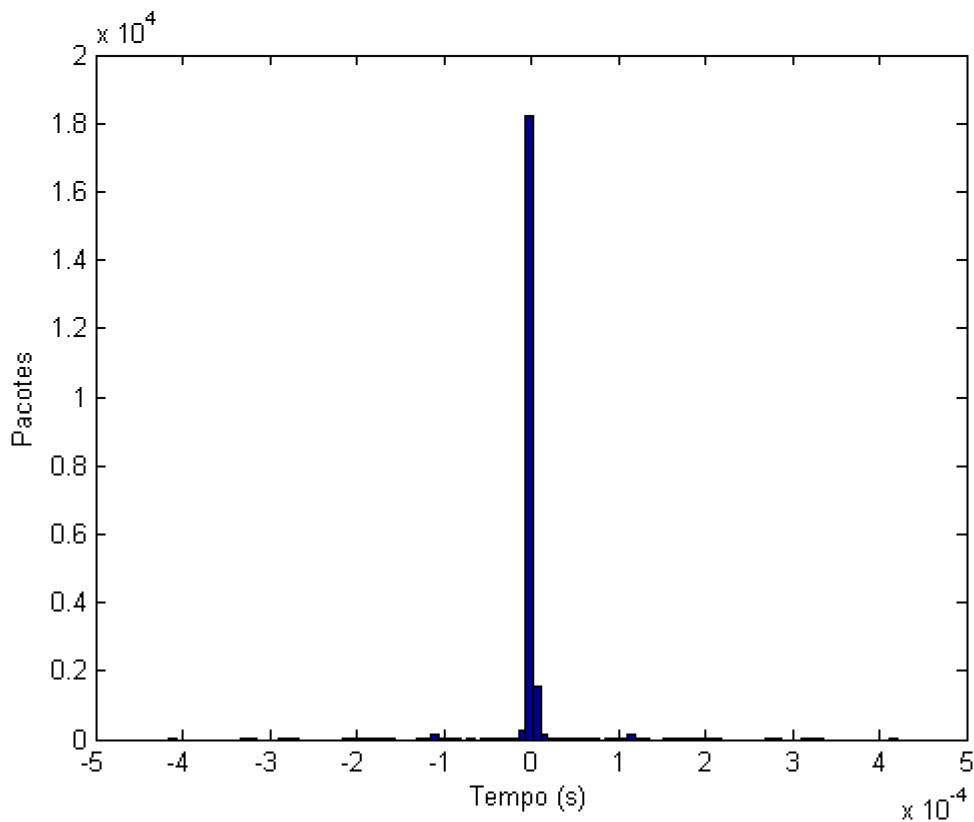


**Figura 6.2 – *Jitter* relativo entre pacotes sucessivos de resposta ao *master* (teste 1)**

A Figura 6.2 apresenta o *jitter* relativo do tempo de resposta. Como se antevia na Figura 6.1, o *jitter* é em média baixo, mas ocasionalmente apresenta valores elevados, da ordem de 400us.

| | *Mínimo* | *Média* | *Máximo* | *Desvio Padrão* |
|---|---|---|---|---|
| Latência (s) | $3.67 \text{ x } 10^{-4}$ | $3.78 \text{ x } 10^{-4}$ | $8.01 \text{ x } 10^{-4}$ | $1.50 \text{ x } 10^{-5}$ |
| *Jitter* relativo (s) | 0 | $4.19 \text{ x } 10^{-6}$ | $4.20 \text{ x } 10^{-4}$ | $1.86 \text{ x } 10^{-5}$ |

**Tabela 6.1 - Valores de latência entre o pedido do *master* e a resposta do *slave* (experiência 1)**

Pode concluir-se que, na sua actual implementação, a *stack* apresenta um tempo de resposta e de *jitter* elevados, que condicionam a duração mínima do EC.

## 6.2 Experiência 2: Integridade dos dados

A experiência realizada para o teste da integridade dos dados é a descrita na secção 5.5.2.

Foram enviadas 20700 TMs (o equivalente a 100 imagens completas), tendo-se sempre verificado a ausência de pacotes com *checksum* errado. Assim, não há nenhum indício da presença de erros na implementação desta parte da *stack*.

## 6.3 Experiência 3: Fragmentação e reagrupamento

A experiência realizada para o teste de fragmentação e reagrupamento é a descrita na secção 5.5.3.

Foram enviadas 20700 TMs (o equivalente a 100 imagens completas), sendo os resultados apresentados de seguida.

|  | *Mínimo* | *Média* | *Máximo* | *Desvio Padrão* |
|---|---|---|---|---|
| Latência (s) | $2.42 \times 10^{-4}$ | $4.69 \times 10^{-4}$ | $8.88 \times 10^{-4}$ | $2.20 \times 10^{-5}$ |

**Tabela 6.2 - Valores de latência entre o pedido do *master* e a resposta do *slave* (experiência 3)**

*Output* da consola do programa do *master*:

```
Ftt packets     : 20700
Wrong fragments :     0


Received images
Received        :   100
Correct images  :   100
Incorrect images:     0

End program, received 100 images successfully.
```

Não houve nenhum pacote fora de ordem. Tal implica que as 100 imagens de teste foram enviadas e recebidas com sucesso.

Analisando a Tabela 6.2 pode verificar-se que o *overhead* adicional devido à fragmentação é de 91us em média, logo não degrada de uma forma significativa o desempenho do sistema.

## 6.4 Experiência 4: Mecanismo duplo *buffer*

A experiência realizada para o teste do mecanismo de duplo *buffer* é a descrita na secção 5.5.4.

Tal como em testes anteriores, foram enviadas 20700 TMs (o equivalente a 100 imagens completas), sendo os resultados apresentados de seguida.

|  | *Mínimo* | *Média* | *Máximo* | *Desvio Padrão* |
|---|---|---|---|---|
| Latência (s) | $2.43 \times 10^{-4}$ | $4.73 \times 10^{-4}$ | $8.72 \times 10^{-4}$ | $2.78 \times 10^{-5}$ |

**Tabela 6.3 - Valores de latência entre o pedido do *master* e a resposta do *slave* (experiência 4)**

Como se pode constatar na Tabela 6.3, o mecanismo de troca de *buffers* não compromete o desempenho, e como tal, é uma solução válida para resolver o problema de acesso a um *buffer* em exclusão mútua.

*Output* da consola do programa do *master*:

```
Pacotes recebidos:
Total    :    0
Ftt-only :20700 Correctos  :20700 Com Erro:    0
Fragmentos Errados:    0


Imagens recebidas:
Recebidas:  100 Correctas:  100 Incorrectas:    0
```

O *output* do programa comprova que não houve problemas no envio e recepção dos dados.

Além do *output*-tipo dos testes anteriores, foi acrescentado um teste, que permite saber se os *buffers* foram usados alternadamente ou não. É precisamente o que significa a linha "0 erros de *buffers* não-alternados": todas as imagens usaram sempre um dos *buffers* alternadamente.

### 6.5 Experiência 5: Mecanismo de troca de *buffers* com interferência em N/2

A experiência realizada para o teste de troca de *buffers* com interferência em N/2 é a descrita na secção 5.5.5.

Seguindo o mesmo raciocínio de experiências anteriores, foram enviadas 20700 TMs (o equivalente a 100 imagens completas), sendo os resultados apresentados de seguida.

| | *Mínimo* | *Média* | *Máximo* | *Desvio Padrão* |
|---|---|---|---|---|
| Latência (s) | 2.42 x $10^{-4}$ | 4.73 x $10^{-4}$ | 8.06 x $10^{-4}$ | 2.73 x $10^{-5}$ |

**Tabela 6.4 - Valores de latência entre o pedido do *master* e a resposta do *slave* (experiência 5)**

*Output* da consola do programa do *master*:

```
Pacotes recebidos:
Total    :    0
Ftt-only :20700 Correctos  :20700 Com Erro:    0
Fragmentos Errados:    0

Imagens recebidas:
Recebidas:  100 Correctas:  100 Incorrectas:    0
```

Conforme se pode observar no *output*, não ocorreram problemas alterando o instante em que se escreve num *buffer*.

### 6.6 Experiência 6: Mecanismo de troca de *buffers* com interferência de *timer*

A experiência realizada para o teste de troca de *buffers* com interferência de *timer* é a descrita na secção 5.5.6.

Foram enviadas 20700 TMs (o equivalente a 100 imagens completas), sendo os resultados apresentados de seguida.

Na Tabela 6.5, apresentam-se os resultados da latência de resposta das mensagens por parte do *slave* aos pedidos do *master*. Foi feito um teste de cada vez, para cada valor temporal do *timer*. Em cada teste, foram pedidas 100 imagens completas. De notar que começando o valor em 0.5 ms, estas situações não se vão deparar nos fragmentos i=N-1 da imagem k=0 e i=0 da imagem k=1, mas sim nos fragmentos 0 e 1 da imagens k=1. Mas as situações possíveis mantêm-se, com as devidas alterações.

| *Timer (ms)* | *Mínimo* | *Média* | *Máximo* | *Desvio Padrão* |
|---|---|---|---|---|
| 0.5 | 2.43 x $10^{-4}$ | 4.74 x $10^{-4}$ | 8.46 x $10^{-4}$ | 2.77 x $10^{-5}$ |
| 0.6 | 2.43 x $10^{-4}$ | 4.74 x $10^{-4}$ | 8.85 x $10^{-4}$ | 2.77 x $10^{-5}$ |
| 0.7 | 2.43 x $10^{-4}$ | 4.74 x $10^{-4}$ | 8.82 x $10^{-4}$ | 2.78 x $10^{-5}$ |
| 0.8 | 2.41 x $10^{-4}$ | 4.73 x $10^{-4}$ | 7.80 x $10^{-4}$ | 2.74 x $10^{-5}$ |
| 0.9 | 2.42 x $10^{-4}$ | 4.73 x $10^{-4}$ | 7.32 x $10^{-4}$ | 2.73 x $10^{-5}$ |
| 1.0 | 2.42 x $10^{-4}$ | 4.73 x $10^{-4}$ | 8.68 x $10^{-4}$ | 2.78 x $10^{-5}$ |

| | | | | |
|---|---|---|---|---|
| 1.1 | 2.42 x $10^{-4}$ | 4.73 x $10^{-4}$ | 8.50 x $10^{-4}$ | 2.74 x $10^{-5}$ |
| 1.2 | 2.43 x $10^{-4}$ | 4.73 x $10^{-4}$ | 8.39 x $10^{-4}$ | 2.75 x $10^{-5}$ |
| 1.3 | 2.42 x $10^{-4}$ | 4.74 x $10^{-4}$ | 8.15 x $10^{-4}$ | 2.79 x $10^{-5}$ |
| 1.4 | 2.43 x $10^{-4}$ | 4.74 x $10^{-4}$ | 8.64 x $10^{-4}$ | 2.77 x $10^{-5}$ |
| 1.5 | 2.43 x $10^{-4}$ | 4.74 x $10^{-4}$ | 8.19 x $10^{-4}$ | 2.93 x $10^{-5}$ |

**Tabela 6.5 - Valores de latência entre o pedido do *master* e a resposta do *slave* (experiência 6), varrendo um ciclo elementar**

*Output* da consola do programa do *master*:

```
Pacotes recebidos:
Total    :    0
Ftt-only :20700 Correctos  :20700 Com Erro:    0
Fragmentos Errados:    0

Imagens recebidas:
Recebidas:  100 Correctas:  100 Incorrectas:    0
```

Como se pode observar existem valores para os quais os valores de latência são menos que a maioria. Serão casos que se pensa que sejam situações *best-case scenario* (1. Ou 7.). Mas na maior parte dos casos, os valores são parecidos, e são a maioria, o que condiz com o esperado, já que a maior parte das situações saíam afectadas na latência.

Mas a conclusão mais importante, é que mesmo no pior dos casos, a latência está contida na janela de 1ms, e bastante previsível (desvio padrão muito baixo).

# Capítulo 7

# 7 Conclusões

Neste capítulo é feita uma análise dos resultados obtidos e análise dos objectivos a que esta dissertação se propunha.

No final são indicadas sugestões de trabalho futuro, como forma de continuação desta dissertação.

## 7.1 Análise de resultados

Os resultados obtidos no âmbito desta dissertação são satisfatórios para conseguir tirar algumas conclusões, porém existem aspectos que poderiam ser melhorados, tendo em vista uma melhor implementação da *stack*.

Um exemplo, seria o de passar o código responsável pela aquisição de uma nova imagem para uma rotina em contexto de interrupção, de forma a aumentar o rigor dos instantes de activação, minimizando assim possivelmente o *jitter* de transmissão de informação.

## 7.2 Análise de objectivos

No que aos objectivos propostos diz respeito, o estudo dos conceitos básicos de tempo-real, da rede *Ethernet* e do protocolo FTT-SE dão-se como concluídos com sucesso.

No que toca à implementação e teste da *stack* FTT-SE, foram também concluídos com sucesso.

Os objectivos de estudo de *software* e *hardware* das câmaras Riavision e implementação de um mecanismo de gestão de QoS nas mesmas, não foram concluídos. Tal deveu-se ao facto de estar acordada uma parceria com a empresa Riamolde - Engenharia e Sistemas S.A., que se comprometeu a fornecer o *hardware* (placa de desenvolvimento *Analog Blackfin* e mais tarde, câmara com *interface Ethernet*), bibliotecas de programação base para a câmara, e ainda suporte às mesmas caso fosse necessário. Infelizmente a parceria não correu da melhor forma, apenas foi fornecida a placa de desenvolvimento, e por esse facto, não foram possíveis de concluir estes objectivos.

O objectivo final de integração no demonstrador, dependiam dos objectivos em falta, e como tal, não foram também realizados.

## 7.3 Trabalho Futuro

Sem o *hardware* e *software* referidos, dificilmente este trabalho poderá ter continuação directa. No entanto, existem algumas componentes que se podem aproveitar para a sua continuação.

O *Framework* de comunicação foi concluído com sucesso, e como tal pode ser utilizado em trabalho futuro. Pode-se eventualmente tentar arranjar uma solução alternativa, uma outra câmara com *interface Ethernet* com bibliotecas de programação disponíveis e compatíveis com o modelo de programação, e continuar esta dissertação no mesmo modelo de uma câmara *Ethernet* compatível com o protocolo. Se assim for, existem várias coisas que se podem fazer, dependendo do tempo disponível:

- Um estudo entre várias câmaras disponíveis no mercado (o desempenho que cada uma delas apresenta e escolher a melhor para a situação em causa);

- Comparar na prática o desempenho do protocolo FTT-SE *versus* outros protocolos existentes;
- Montagem de uma rede com vários nós (câmaras ou não), tendo em vista testar o protocolo FTT-SE em condições de maior exigência de desempenho;
- O mesmo acima, mas comparando o FTT-SE e outros protocolos.

# Bibliografia

# Bibliografia


1. Marau, R.R.D., *Comunicações de tempo-real em switched Ethernet suportando gestão dinâmica de QdS*, in *Departamento de Electrónica, Telecomunicações e Informática*. 2009, Universidade de Aveiro: Aveiro.
2. Pedreiras, P.B.R., *Suporte de Comunicações de Tempo-Real Flexíveis em Sistemas Distribuídos*, in *Departamento de Electrónica, Telecomunicações e Informática*. 2003, Universidade de Aveiro: Aveiro.
3. Luís Almeida, J.F., and Pedro Fonseca, *Flexible Time-Triggered communication on a controller area network*, in *Departamento de Electrónica, Telecomunicações e Informática*. 1998, Universidade de Aveiro: Aveiro.
4. Bosch, *CAN Specification Version 2.0*. 1991.
5. Figueiredo, N.M.d.M., *Controlo Distribuído de Plataformas para Experiências de Mecatrónica*, in *Departamento de Electrónica, Telecomunicações e Informática*. 2008, Universidade de Aveiro: Aveiro.
6. *Wikipedia*. "System" [acedido em 2010/04/19]; Link: http://en.wikipedia.org/wiki/System.
7. *Wikipedia*. "Control system" [acedido em 2010/04/19]; Link: http://en.wikipedia.org/wiki/Control_system.
8. Buttazzo, G., *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. 1st ed. Real-Time Systems Series. 1995: Springer. 379.
9. *Wikipedia*. "Sistema de alta disponibilidade" [acedido em 2010/04/27]; Link: http://pt.wikipedia.org/wiki/Sistema_de_alta_disponibilidade.
10. *Wikipedia*. "Single Point Of Failure" [acedido em 2010/04/27]; Link: http://en.wikipedia.org/wiki/SPOF.
11. Spurgeon, C.E., *Ethernet: the definitive guide*. 2000: O'Reilly. pp. 10-20.
12. *IEEE*. [acedido em 2010/06/10]; Link: http://www.ieee.org/.
13. *Standard Ethernet*. "Drive by wire" [acedido em 2010/08/10]; Link: http://standards.ieee.org/getieee802/802.3.html.
14. *Wikipedia*. "Standard IEEE 802.3" [acedido em 2010/06/11]; Link: http://en.wikipedia.org/wiki/IEEE_802.3.
15. Raimond Pigan, M.M., *Automating with PROFINET, 2nd Edition*. 2008: Publicis Publishing, Erlangen.
16. *PROFIBUS & PROFINET International Home Page*. "System" [acedido em 2010/05/19]; Link: http://www.profibus.com/.
17. *PROFINET: The industrial Ethernet Standard for Automation (presentation)*.
18. *Wikipedia*. "Ethernet Powerlink" [acedido em 2010/08/10]; Link: http://en.wikipedia.org/wiki/Ethernet_Powerlink.
19. *IXXAT POWERLINK Introduction*. [acedido em 2010/08/10]; Link: http://www.ixxat.com/powerlink_technologie_en.html.
20. *What's The Power Behind ETHERNET Powerlink*. [acedido em 2010/08/10]; Link: http://www.motioncontrolonline.org/i4a/pages/Index.cfm?pageID=3458.
21. *POWERLINK Basics brochure (en)*. [acedido em 2010/08/10]; Link: http://www.ethernet-

powerlink.org/index.php?id=12&tx_abdownloads_pi1[action]=getviewclickeddownload&tx_abdownloads_pi1[uid]=203&no_cache=1.

22. *POWERLINK basics (en)*. [acedido em 2010/08/10]; Link: http://www.ethernet-powerlink.org/index.php?id=12&tx_abdownloads_pi1[action]=getviewclickeddownload&tx_abdownloads_pi1[uid]=7&no_cache=1.
23. *POWERLINK Top Ten Facts (en)*. [acedido em 2010/08/10]; Link: http://www.ethernet-powerlink.org/index.php?id=12&tx_abdownloads_pi1[action]=getviewclickeddownload&tx_abdownloads_pi1[uid]=37&no_cache=1.
24. *B&R*. [acedido em 2010/08/10]; Link: http://www.br-automation.com/cps/rde/xchg/br-automation_com/hs.xsl/index_ENG_HTML.htm.
25. *Wikipedia*. "TDMA" [acedido em 2010/08/10]; Link: http://en.wikipedia.org/wiki/Time_division_multiple_access.
26. *TTTech Computertechnik AG*. [acedido em 2010/08/10]; Link: http://www.tttech.com/.
27. *TTEthernet White Paper*. [acedido em 2010/08/10]; Link: http://www.tttech.com/fileadmin/content/white/TTEthernet/TTEthernet_Article.pdf.
28. *Wikipedia*. "Drive by wire" [acedido em 2010/08/10]; Link: http://en.wikipedia.org/wiki/Drive_by_wire.
29. *Analog Devices Homepage*. [acedido em 2010-05-20]; Link: http://www.analog.com/.
30. *EZ-KIT Lite® for the ADSP-BF533, ADSP-BF532, and ADSP-BF531 Blackfin® Processors* [acedido em 2010/05/20]; Link: http://www.analog.com/en/embedded-processing-dsp/blackfin/BF533-HARDWARE/processors/product.html.
31. *ADZS-BF533-EZLITE image*. [acedido em 2010/05/20]; Link: http://www.analog.com/static/imported-files/images/Product_Descriptions/38092608585777137873875bf533_hardware.jpg.
32. *Blackfin USB-LAN EZ-Extender*. [acedido em 2010/05/20]; Link: http://www.analog.com/en/embedded-processing-dsp/blackfin/BF-EXTENDERUL/processors/product.html.
33. *ADZS-USBLAN-EZEXT image*. [acedido em 2010/05/20]; Link: http://www.analog.com/static/imported-files/images/Product_Descriptions/BlackfinUSBLANEZ_Extender_revised.jpg.
34. *USB-Based Emulator and High Performance USB-Based Emulator* [acedido em 2010/05/20]; Link: http://www.analog.com/en/embedded-processing-dsp/blackfin/USB-EMULATOR/products/product.html.
35. *ADZS-USB-ICE image*. [acedido em 2010/05/20]; Link: http://www.analog.com/static/imported-files/images/Product_Descriptions/5754930502391903402849118416824438334027445629891175OOusb_emulator.jpg.
36. *Wikipedia*. "Joint Test Action Group" [acedido em 2010/04/27]; Link: http://en.wikipedia.org/wiki/Joint_Test_Action_Group.
37. *Wikipedia*. "Checksum" [acedido em 2010/07/22]; Link: http://en.wikipedia.org/wiki/Checksum.