FTT-SE: Desenvolvimento de um *dissector* para um protocolo de tempo real

César Gomes [1]
Universidade de Aveiro

Orientador: Paulo Pedreiras[2]
Universidade de Aveiro

30 de Outubro de 2010

[1]cesargomes@ua.pt
[2]pbrp@ua.pt

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

Este trabalho surgiu no âmbito do projecto ***Bolsa de Integração na Investigação - 2009*** financiado pela **FCT**[1] e tem por objectivo o desenvolvimento de um *protocol dissector* em *Wireshark* para a família de protocolos ***FTT-SE***[2] sobre *Ethernet*.

1.1 Motivação

Existe hoje em dia uma classe de aplicações de tempo real que incluem actividades críticas que impõem requisitos de pontualidade e previsibilidade.Estas aplicações, normalmente implementadas em arquitecturas distribuídas levaram ao desenvolvimento de protocolos de comunicação específicos, denominados protocolos de tempo real, que permitem oferecer garantias determinísticas de previsibilidade e latência.

Têm-se observado um crescimento no desenvolvimento destes protocolos assentes na plataforma *Ethernet* que se prendem essencialmente por elevada capacidade, disponibilidade de recursos técnicos e humanos, e baixo custo.

Existem já no mercado varias ferramentas de capazes de observar o tráfego que circula numa rede do tipo referido, vulgarmente conhecidas por *sniffers*, mas possuem apenas capacidade de descodificar os protocolos de comunicações mais genéricos, por este motivo, não é comum encontrar ferramentas deste tipo para redes de *tempo-real*.

Daí que seja essencial desenvolver uma ferramenta capaz de analisar o tráfego destas redes para garantir o bom funcionamento destas.

O objectivo deste trabalho está definido então como o desenvolvimento de um *plug-in* para o *Wireshark* para este tipo de redes.

[1] Fundação para a Ciência e Tecnologia
[2] *Flexible Time Triggered*

Capítulo 2

O sistema *Ethernet*

2.1 História

O sistema *Ethernet* foi inventado em 1973 por Bob Metcalfe na *Palo Alto Research Center* na California. Àquela data estava descrito como um sistema de rede para ligar estações de trabalho e impressoras a laser de alta velocidade.

Desde então o sistema *Ethernet* evoluiu bastante na sua velocidade de transmissão, 2.94 Mbps até 10 Gbps, e também no seu meio físico, desde cabo coaxial passando por cabos *twisted pair* e mais recentemente fibra optica.

O sistema *Ethernet*, é uma tecnologia que define uma norma de comunicação em redes locais, *local area network (LAN)*, nela estão defenidos: o suporte físico de transmissão, a maneira como são transmitidos os impulsos digitais, o *hardware* suportado etc., todas as especificações necessárias para se conseguir montar uma rede área local entre quaisquer computadores.

Hoje em dia o *Ethernet* é largamente utilizado para montar *LAN's* sendo das tecnologias mais utilizadas em todo o mundo, os factores que o tornaram popular foram o baixo custo, a escabilidade, a fiabilidade, e o largo acesso a ferramentas de gestão [2].

2.2 Os quatro elementos básicos

Uma *LAN Ethernet* é feita de *hardware* e *software* que em conjunto entregam dados digitais entre computadores. Para conseguir isto quatro elementos precisam de ser combinados para fazerem um sistema *Ethernet*.

Os elementos são:

- A *frame* - Um conjunto de bytes normalizado usado para transportar dados pela rede;
- O *Medium Accsess Control Protocol* - Um conjunto de regras embutidas em cada interface *Ethernet* que permite que vários computadores

acedam a um canal partilhado;

- Componentes de Sinalização - componentes electrónicos que recebem e enviam sinais através do canal;

- Meios Físicos - cabos e outro *hardware* usados para transportar os sinais digitais aos computadores ligados à rede;

[2]

2.2.1 *Ethernet Frame*

Figura 2.1: A *frame* Ethernet mais comum [1]

A *frame* é o componente mais importante de um sistema *Ethernet*, o resto dos componentes existem para moverem *frames* de um lado para o outro. A sua estrutura é mostrada na figura 2.1.

- O *preâmbulo* é um conjunto de bits usado para sinalizar o inicio e sincronismo de uma frame;

- Os *Destination Address* e o *Source Address* são respectivamente o endereço de destino e o endereço de fonte da frame. Tal como nas nossas casas temos um endereço único para o qual nos mandam a nossa correspondência, no caso das redes *Ethernet* cada computador possui um endereço para o qual as mensagens são enviadas;

- O campo *Ethertype* pode ser usado de duas maneiras numa delas este contém o tipo de mensagem que está a ser enviada, i.e, identifica o protocolo que está presente na camada acima na mensagem, ou noutro caso pode identificar o tamanho do campo *DATA*;

- No campo *DATA* encontramos os dados em si que possuem a informação a ser transmitida;

- O campo FCS[1] é usado para deteção de erros na recepção/transmissão da *frame*;

[1]Frame Check Sequency

2.3 Topologias de Ligação

2.3.1 *Shared Ehternet - half duplex*

A arquitectura original do *Ethernet* é partilhada e *half-duplex*, i.e, os dispositivos estão ligados ao mesmo canal físico, podendo haver apenas *hub s* repetidores no caminho, e é *half-duplex* no sentido em que cada dispositivo pode estar apenas a enviar ou a receber num determinado momento, nunca em simultâneo.

Cada dispositivo é independente dos restantes, não existe um dispositivo central a controlar os outros, é então dada a mesma oportunidade para ocupar o canal a cada dispositivo.

O *Ethernet* usa um mecanismo *broacast delivery* no sentido em que todas as mensagens são ouvidas por todos os dispositivos ligados à rede, existe depois um circuito de *address matching* do lado do receptor que compara o endereço de destino de cada *frame* com o seu próprio para que este apenas processe a informação para ele endereçada.

Do lado da transmissão podemos notar o problema que se dá quando mais que um dispositivo querem enviar informação ao mesmo tempo, quando isto acontece dizemos que temos uma colisão, para resolver este o problema está defeniu-se o mecanismo *CSMA/CD*, *Carrier Sense Multiple Access with Colision Detection*

2.3.2 *Switched Ethernet - full duplex*

Nesta arquitectura os nós estão ligados a um dispositivo central, *switching hub*, efectuando canais independentes, segmentos de rede, para cada dispositivo.

Esta arquitectura elimina então o problema das colisões, visto que os nós da rede não competem pela mesmo canal, e permite a operação em *full-duplex*, i.e. cada dispositivo pode enviar e receber dadosn em simultâneo.

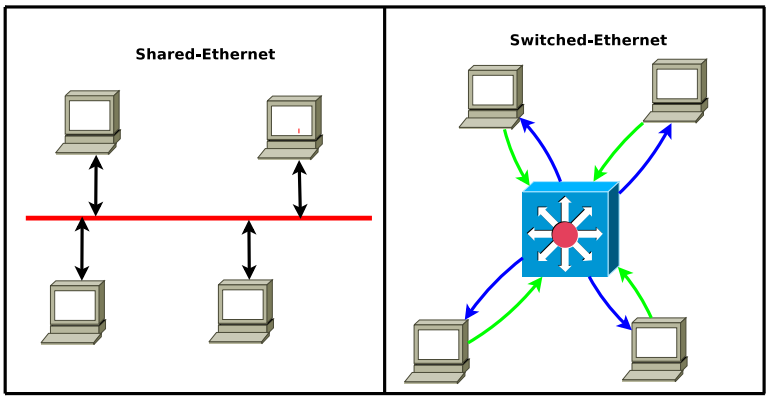

Figura 2.2: *Shared vs Switched Ethernet*

Capítulo 3

Wireshark

3.1 O que é o *Wireshark*

O *Wireshark* é um programa de *análise de redes* que se insere numa categoria de programas chamados *sniffers*. Estes programas caracterizam-se por monitorizar o tráfego que viaja numa rede.

A análise da rede pelo *Wireshark* caracteriza-se pela leitura e descodificação de pacotes e a sua apresentação numa forma legível de forma a determinar o que está a acontecer na rede. O grande espectro de protocolos que consegue analisar e o facto de ser um programa open-source que é activamente suportado, faz deste um dos programas mais utilizados para a análise de redes.

3.2 Funcionamento

O *Wireshark* obtém os pacotes a analisar através de um processo designado por *capture*. Se a rede estiver ligada através de um *ethernet hub* o *Wireshark* pode configurar a placa de rede (NIC) em *promiscuous mode* e obter não só pacotes endereçados para o computador em que o programa está instalado, mas todos os pacotes a viajar na rede. Posteriormente podemos analisar a informação descodificada pelo *Wireshark* através das suas funcionalidades (Figura 3.1):

3.2.1 GUI(Graphical User Interface)

O GUI do *Wireshark* é configurável e fácil de usar, a informação sobre os pacotes é mostrada em três áreas da janela (figura 3.2):

summary -no topo, mostra: o número do pacote, o tempo, o endereço de origem, o endereço de destino e o nome e informação sobre a camada mais alta do protocolo;

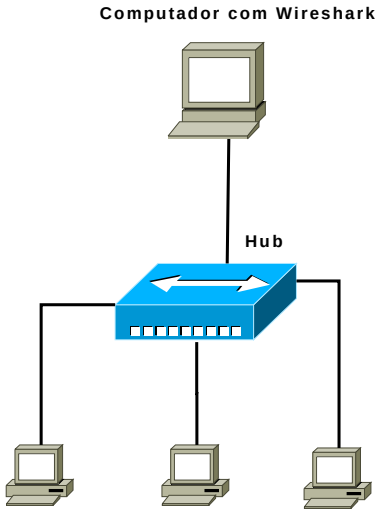

Figura 3.1: possível topologia de ligação do *Wireshark* numa rede

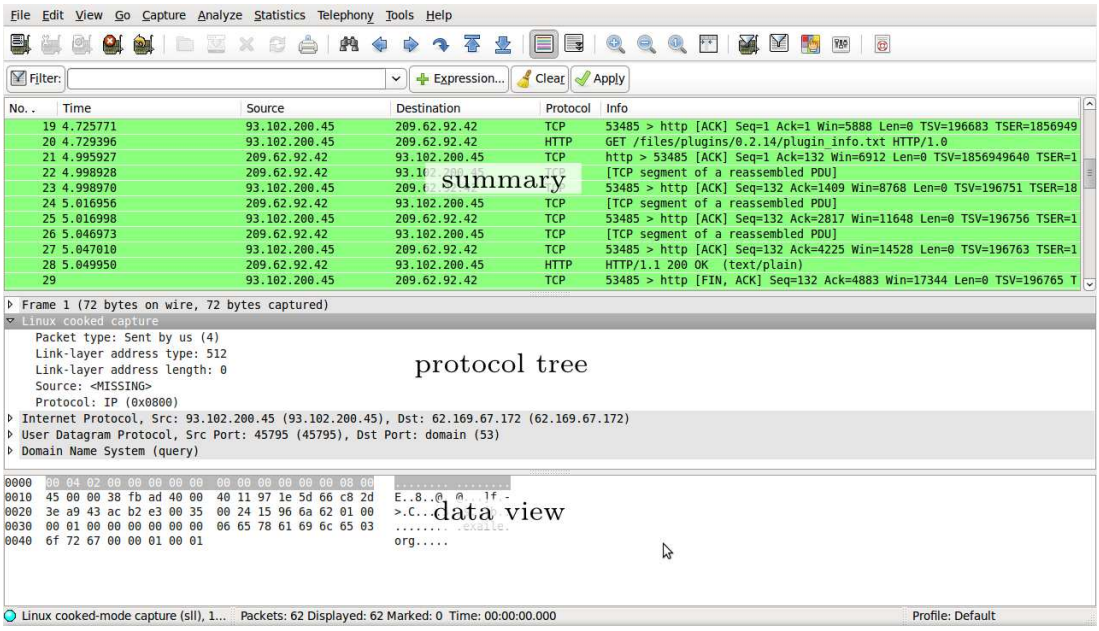

Figura 3.2: GUI do *Wireshark*

protocol tree -no meio, mostra detalhes sobre cada camada contida no pacote capturado (desde a camada mais alta (ex. HTML) até a mais baixa (ex. Ethernet));

data view -em baixo, mostra os dados capturados em formato *raw* em **HEX** e **ASCII**;

3.2.2 Filtros

O *Wireshark* tem a capacidade de usar filtros de captura(*capture filters*) e filtros de apresentação(*display filters*). Os *display filters* possuem uma sintaxe própria,que permite fazerem-se funções para seguir conversações entre computadores(p.ex: *"folow TCP stream"*, que é já uma função *built in* mas é baseada na sintaxe dos filtros), a sintaxe destes possui operações lógicas e identificação dos vários campos dos protocolos (ex. *source address, destination address, time, segmentation, flags, crc*).

Embora tenham funções em comum os *capture filters* possuem uma sintaxe menos poderosa que os *display filters*.

3.2.3 Estatística

O *Wireshark* possui algumas ferramentas estatísticas, para analisar o tráfego da rede, que mostram por exemplo:

- informações sobre os protocolos usados
- informações sobre conversações
- gráficos de *throughput* e tempo-sequência

3.3 Packet Dissection

Os componentes do *Wireshark* que descodificam os pacotes são chamados *protocol dissectors*. Estes componentes são módulos de código fonte individuais que dão instruções à main do *Wireshark* em como descodificar um tipo de protocolo específico.

3.3.1 Funcionamento

Cada *dissector* descodifica a sua parte do protocolo, e depois entrega descodificação a *dissectors* subsequentes para algum protocolo *encapsulado*. A cada fase detalhes do pacote são descodificados e mostrados.

O processo de *"Dissection"*

Quando o *Wireshark* lê um pacote realiza os seguintes passos:

1. Os dados da *frame*[1] são passados pela função *epan_dissect_run()*[2].

[1]Uma *frame* é a estrutura básica de comunicação de dados(conjunto de bits que constituem um pacote

[2]Função que desencadeia o processo de *dissection de um pacote*

2. A função *epan_dissect_run()* define os ponteiros para a *frame*, *column* e *data* e chama a função *dissect_packet*.

3. A função *dissect_packet* cria o "primeiro"*tvbuff*[3] e depois chama a função *dissect_frame()*.

4. A função *dissect_frame()* descodifica os dados da *frame* e mostra a na janela *Decode* do *GUI*.

5. A função *dissect_frame()* chama a função *dissector_try_port()* para ver se existem alguns *dissectors* para descodificar a próxima parte do *tvbuff*, caso existam, é chamado o próximo *dissector* .

6. O *dissector* descodifica a sua parte do protocolo e passa o *payload* restante para o *tvbuff* e é chamada de novo a função *dissector_try_port()*.

7. Este processo de descodificar cada camada do pacote continua até não existir mais dados ou não existir nenhum dissector registado para os dados restantes. Uma vez isto a próxima *frame* é lida.

3.4 Programação

No contexto do projecto de investigação é apropriado desenvolver um *protocol dissector* como *plug-in*.

O *protocol dissector* é normalmente escrito em **ANSI-C** de modo a ser o mais portável possível dado a que o *Wireshark* corre em múltiplas plataformas.

Um dissector pode ser desenvolvido para integrar o código fonte, ou pode ser implementado como *plug-in*, a vantagem de um *plug-in* é que não é necessário a recompilação completa do *Wireshark* durante o desenvolvimento do código.

[3] O *tvbuff* é um *buffer* usado para passar o *payload* restante a ser descodificado para algum *dissector* adicional, este *buffer* deve ser actualizado por cada *dissector*

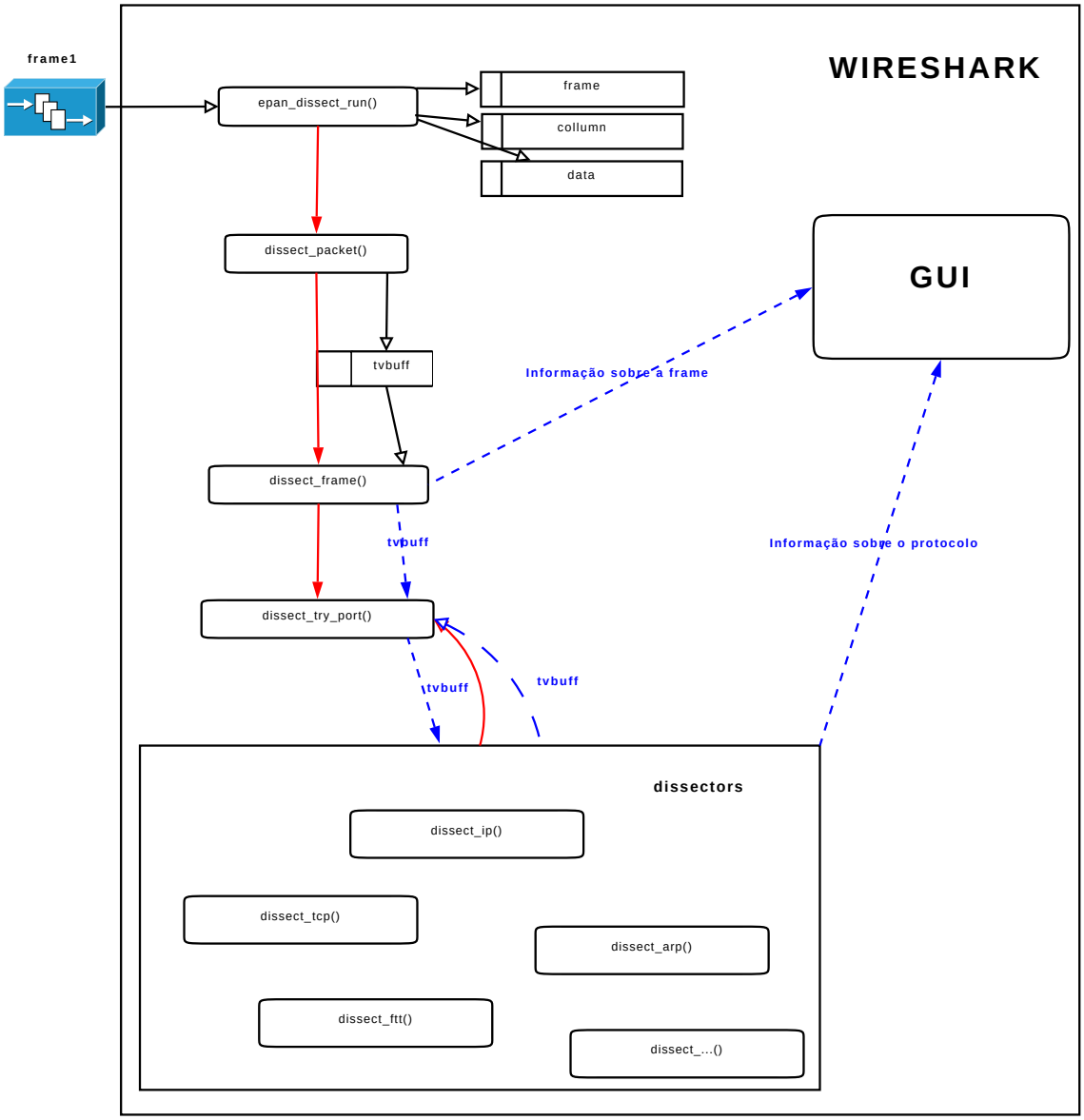

Figura 3.3: processo de *packet dissection*

Capítulo 4

Desenvolvimento de um *dissector* como *plug-in*

4.1 Introdução

O *Wireshark* é software *open source* distribuído sob a licença *General Public License* (*GPL*). Neste documento não irei descrever os mecanismos de compilação e empacotamento dos *dissectors* do *Wireshark* .

4.2 Pré-requisitos

Para desenvolvermos um dissector temos de obter o *código fonte* do *Wireshark* na versão *developer*. É também recomendável que se obtenha também os documentos *Wireshark Developer's guide* e *Wireshark User's guide*.

Antes de podermos modificar ou adicionar dissectors ao *Wireshark* temos de der capazes de compilar a aplicação apartir do *código fonte*.Para compilarmos do código fonte precisamos de ferramentas adicionais que podem não estar incluídas por defeito no sistema em que pretendemos desenvolver o dissector. As instruções de instalação do *Wireshark* estão bem descritas no ficheiro `INSTALL` no directório raiz do código fonte do *Wireshark* .

A maior parte do código do *Wireshark* que nos interessa está escrito em *ANSI-C*[1], é utilizado *ANSI-C* devido à necessidade de portabilidade do código entre os múltiplos sistemas operativos em que *Wireshark* funciona. Quando estivermos a escrever em C temos de ter em especial atenção em usar apenas funções que estejam disponíveis nos vários sistemas operativos.

[1] *American National Standards Institute-C*

4.3 Recursos

Para podermos desenvolver o *dissector* temos de conhecer a API do *Wireshark* para tal existem vários locais onde podemos aceder para obter essa informação. Embora existam mais recursos apenas referenciarei aqueles que achar relevantes.

4.3.1 Recursos na *Internet*

Existem na *Internet* vários recursos com muita informação dos quais destaco o *Wiki* do *Wireshark* , que contem muita informação, incluindo recursos de utilizadores e de programadores, informação sobre protocolos e notas dos programadores, e também a *Wireshark-dev mailing list*.

4.3.2 Recursos no *código fonte*

Durante o desenvolvimento de um *dissector* é muitas vezes necessário recorrer ao código fonte do *Wireshark* para saber como utilizar as funções por ele suportadas. São bastante importantes os seguintes directórios e ficheiros.

Directório `doc`

No directório *doc* estão documentos de texto que nos assistem durante o processo de desenvolvimento de um *dissector* .

`README.developer` - O documento principal que nos dá suporte no desenvolvimento de um novo *dissector* . Nele estão incluídas: metodologias a seguir, informações muito importantes sobre o *design* de um dissector, um *template* com as principais funções de um *dissector* e sua descrição, e potenciais problemas que poderam surgir aquando do desenvolvimento de um dissector.

`README.malloc` - Nele está descrita superficialmente a *API* de gestão de memória do *Wireshark* .

`README.binarytrees` - Informação sobre a utilização de *árvores binárias* num *dissector* .

`README.plugins` - Documentação sobre como utilizar a interface *plug-in* no *Wireshark* .

`README.request_response_tracking` - informação de como implementar estruturas de *tracking* de conversações num *dissector* .

Directório *epan*

O directório *Ethernet Protocol Analyzer (EPAN)* contem *protocol dissectors* e as funções mais úteis e mais usadas num *dissector* . Dou destaque ao ficheiro *packet_info.h* pois contem informação sobre uma estrutura básica com muita informação da maior importância sobre os pacote a ser analisado.

Directório *Plug-ins*

É neste directório que se encontram os *dissectors* implementados pela interface *plug-in* e é neste também que vamos colocar o *código fonte* para o nosso *dissector* .

4.4 Desenvolvimento do dissector

Antes de começar-mos a construir um *dissector* é recomendável que se analise a criação passo a passo de um *dissector* presente no *Wireshark Developer's guide*. Depois de analisado o anterior documento pode se começar um *dissector* de raiz, apartir do *template* no ficheiro *README.developer*, ou alterar um *dissector* já existente.

4.4.1 Instalação do *dissector* como *plug-in* apartir do *código fonte*

Ver ficheiro `README.plugins` em `doc/`.

4.5 Estrutura de um *dissector*

A análise desta secção do relatório deve ser acompanhada pelo *template* de um *dissector* do ficheiro `/doc/README.developer`. Pressupoem as substituições:

`PROTONAME` - Nome do Protocolo

`PROTOSHORTNAME` - Um nome abreviado para o protocolo

`PROTOABBREV` - Um nome para usar nas expressões dos filtros; deve apenas conter letras minúsculas, dígitos e hifens.

4.5.1 Preâmbulo

```
1  #ifdef HAVE_CONFIG_H
2  # include "config.h"
3  #endif
4
5  #include <stdio.h>
```

```
6     #include <stdlib.h>
7     #include <string.h>
8
9     #include <glib.h>
10
11    #include <epan/packet.h>
12    #include <epan/prefs.h>
```

Nesta secção defenimos os ficheiros que precisamos de incluir para o programa, estes são precisos para as funções globais que irão ser usadas no *dissector* . Quando é preciso usar funções mais especificas temos de adicionar a sua fonte neste local.

4.5.2 Registo de parâmetros e declarações de funções

```
1     /* Forward declaration we need below (if using
          proto_reg_handoff...
2        as a prefs callback)       */
3     void proto_reg_handoff_PROTOABBREV(void);
4
5     /* Initialize the protocol and registered fields */
6     static int proto_PROTOABBREV = -1;
7     static int hf_PROTOABBREV_FIELDABBREV = -1;
8
9     /* Global sample preference ("controls" display of numbers)
          */
10    static gboolean gPREF_HEX = FALSE;
11    /* Global sample port pref */
12    static guint gPORT_PREF = 1234;
13
14    /* Initialize the subtree pointers */
15    static gint ett_PROTOABBREV = -1;
```

Nesta secção declaramos variaveis essenciais ao registo do *dissector* no *Wireshark* , bem como o registo dos *campos* analisados com informação relevante a passar para o *Wireshark* .

Podem ainda declararar-se nesta secção, variáveis globais, estruturas e constantes necessárias ao funcionamento do *dissector* .

4.5.3 Função *proto_register_PROTOABBREV*

```
1     void
2     proto_register_PROTOABBREV(void)
3     {
4       module_t *PROTOABBREV_module;
5
6     /* Setup list of header fields  See Section 1.6.1 for
          details*/
7       static hf_register_info hf[] = {
8         { &hf_PROTOABBREV_FIELDABBREV,
```

```
9             { "FIELDNAME",              "PROTOABBREV.FIELDABBREV",
10            FIELDTYPE, FIELDBASE, FIELDCONVERT, BITMASK,
11            "FIELDDESCR", HFILL }
12          }
13        };
14
15      /* Setup protocol subtree array */
16        static gint *ett[] = {
17          &ett_PROTOABBREV
18        };
19              ...
20
21      /* Register the protocol name and description */
22        proto_PROTOABBREV = proto_register_protocol("PROTONAME",
23          "PROTOSHORTNAME", "PROTOABBREV");
24
25        PROTOABBREV_module = prefs_register_protocol(
              proto_PROTOABBREV,
26          proto_reg_handoff_PROTOABBREV);
27
28      /* Register a sample preference */
29        prefs_register_bool_preference(PROTOABBREV_module, "
              show_hex",
30          "Display numbers in Hex",
31          "Enable to display numerical values in hexadecimal.",
32          &gPREF_HEX);
33
34      /* Register a sample port preference   */
35        prefs_register_uint_preference(PROTOABBREV_module, "tcp.
              port", "PROTOABBREV TCP Port",
36          " PROTOABBREV TCP port if other than the default",
37          10, &gPORT_PREF);
38      }
```

Esta função é usada para registar o *dissector* no *Wireshark* , os *scripts* *make-reg-dotc* e *make-reg-py* passam por todos os *dissectors* e constroem a lista de modo a que sejam registados no *motor* do *Wireshark* .

Na linha 7 temos a declaração do *hf array*, este *array* é importante para tirar partido das capacidade dos *display filter*, cada elemento do *array* será um item que poderá ser filtrado no *Wireshark* (ex: `ip.src=0.0.0.255`) . Podem se definir diferentes itens em diferentes tipo de conteúdos p.e. , *Ethernet addresses*, números em hexadecimal, *strings* etc.

4.5.4 Função *proto_reg_handoff_PROTOABBREV*

```
1     void
2     proto_reg_handoff_PROTOABBREV(void)
3     {
4       dissector_handle_t PROTOABBREV_handle;
5
```

```
6     /*  Use new_create_dissector_handle() to indicate that
          dissect_PROTOABBREV()
7      *  returns the number of bytes it dissected (or 0 if it
           thinks the packet
8      *  does not belong to PROTONAME).
9      */
10      PROTOABBREV_handle = new_create_dissector_handle(
            dissect_PROTOABBREV,
11                  proto_PROTOABBREV);
12      dissector_add("PARENT_SUBFIELD", ID_VALUE,
            PROTOABBREV_handle);
13    }
```

Esta função é usada para instruir o *Wireshark* de quando deve chamar o *dissector*. A função *new_create_dissector_handle()* na linha 10, passa ao *Wireshark* qual a função que deve chamar para processar os pacotes.

A função *dissector_add()*na linha 12 passa ao *Wireshark* o elemento que despoletará o *dissector* , esse elemento é o *PARENT_SUBFIELD* (ex: "*ethertype*" no caso do elemento ser o campo *type* do protocolo *Ethernet*, "*tcp.port*" no caso do elemento ser o campo *port* do protocolo *TCP*), o valor que toma o elemento será o *ID_VALUE*.

4.5.5 Função *dissect_PROTOABBREV()*

```
1     static int
2     dissect_PROTOABBREV(tvbuff_t *tvb, packet_info *pinfo,
          proto_tree *tree)
3     {
4
5     /* Set up structures needed to add the protocol subtree and
          manage it */
6       proto_item *ti;
7       proto_tree *PROTOABBREV_tree;
8
9         ...
10
11    /* Make entries in Protocol column and Info column on
          summary display */
12      col_set_str(pinfo->cinfo, COL_PROTOCOL, "PROTOABBREV");
13
14    /* This field shows up as the "Info" column in the display;
          you should use
15       it, if possible, to summarize what's in the packet, so
             that a user looking
16       at the list of packets can tell what type of packet it is
             . See section 1.5
17       for more information.
18
19        ...
20
21      col_clear(pinfo->cinfo, COL_INFO);
```

```
22
23        col_set_str(pinfo->cinfo, COL_INFO, "XXX Request");
24
25          ...
26
27
28      /* create display subtree for the protocol */
29          ti = proto_tree_add_item(tree, proto_PROTOABBREV, tvb,
                0, -1, FALSE);
30
31          PROTOABBREV_tree = proto_item_add_subtree(ti,
                ett_PROTOABBREV);
32
33      /* add an item to the subtree, see section 1.6 for more
            information */
34          proto_tree_add_item(PROTOABBREV_tree,
35            hf_PROTOABBREV_FIELDABBREV, tvb, offset, len, FALSE);
36
37
38      /* Continue adding tree items to process the packet here */
39
40
41        }
42
43      /* If this protocol has a sub-dissector call it here, see
            section 1.8 */
44
45      /* Return the amount of data this dissector was able to
            dissect */
46        return tvb_length(tvb);
47      }
```

Esta é a função principal do *dissector* nela tem-se acesso a varias estruturas que contém informação sobre o pacote a ser analisado, elas são:

`packet_info *pinfo` - estrutura com variadas informações sobre o pacote (ex: tempo, *tcp port* ...), mais informações no ficheiro `/epan/packet_info.h`.

`tvbuff_t *tvb` - estrutura com *buffers* com a *raw data* do pacote, mais informações no ficheiro `/epan/tvbuff.h`.

Nesta função deve se começar por verificar as informações principais sobre o pacote para garantir que o pacote pertence ao protocolo que se está analisar. Após isso é recomendável que se coloque informação sobre a mensagem na *summary window*, colocando informação na *column* e *info column*, é recomendável tambem criar uma *subtree* para mostrar informação sobre o protocolo na zona *protocol tree* da *GUI* do *Wireshark* pode ainda colocar-se algum processamento de dados nesta função, mas caso o *dissector* a implementar seja muito complexo é recomendável que se divida o processamento dos dados por várias funções.

Capítulo 5

O protocolo *FTT-SE*

5.1 Protocolos de tempo real

Nas comunicações em redes de controlo e monitorização de processos industriais, existem certos requisitos que são diferentes das redes de comunicação de uso comum em casa ou no escritório. Por exemplo, a predictabilidade é favorecida em detrimento da taxa de transmissão média, e existem frequentemente relações de precedência temporal entre mensagens. Qualquer incumprimento destes requisitos pode ter impacto negativo significativo no desempenho do controlo de um processo industrial.

Para satisfazer estes requisitos, têm vindo a ser desenvolvidas redes de comunicação especificas que se adaptam às comunicações em ambiente industrial, em que é preciso garantir eficiência temporal na monitorização de sensores e no controlo de automatismos.

Existem já varias soluções para as comunicações em "*tempo real*", no entanto é interessante desenvolver uma solução que consiga garantir este tipo de comunicações assentando-se na plataforma *Ethernet*, devido à sua fácil implementação, alta disponibilidade de recursos técnicos e humanos e baixo custo. Visto que a *Ethernet* é uma arquitectura de rede uso geral, não está preparada para este tipo de comunicações.

O protocolo ***FTT-SE*** propõe então fazer com que uma rede em *Ethernet* satisfaça os requisitos de uma rede de "*tempo real*".

5.1.1 Requisitos

Os sistemas de comunicação que suportam aplicações de "tempo real" tem de cumprir alguns requisitos. Os requisitos mais comuns são predictabilidade, tratamento de tráfego periódico com diferentes períodos, tratamento de tráfego esporádico, latência limitada, eficiência no tratamento de pacotes e gestão dinâmica de *QoS*.

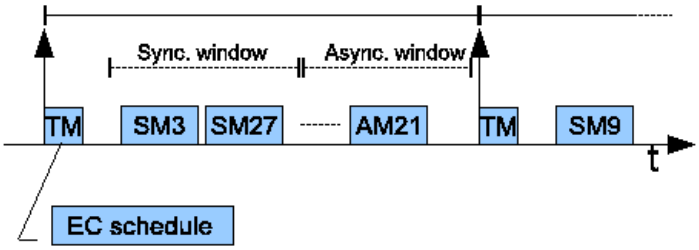

Figura 5.1: Estrutura do *Elementary Cycle*

5.2 Arquitectura

De modo a preencher os requisitos das redes de "tempo real", o protocolo ***FTT-SE*** utiliza uma arquitectura centralizada de agendamento e controlo de transmissão *master/multislave*. Dado que o controlo de transmissão e o agendamento estão localizados num só nó (*master*), é possível a implementação da gestão dinâmica de *QoS*;

O *Master* é responsável por dizer a cada *slave* quando e o que transmitir, fazendo assim com que exista uma noção coerente de tempo em cada nó da rede, e tambem consegue-se evitar colisões na rede.

5.3 Funcionamento do protocolo

5.3.1 *Elementary Cycle*

Um dos aspectos fundamentais do protocolo é o *Elementary Cycle*(EC), que é uma janela temporal de duração fixa que aloca o tráfego de mensagens no barramento. O tempo no barramento é então organizado numa sucessão infinita de ECs. Em cada EC existem ainda duas janelas temporais, a síncrona e a assíncrona, dedicadas tráfego desencadeado por tempo e trafico desencadeado por eventos, respectivamente (Figura 5.1).

Cada EC começa com o *broadcast*[1] de uma *Trigger Message*(TM) pelo *Master*, que sincroniza a rede e identifica todas as mensagens, síncronas ou assíncronas, a serem processadas naquele EC. As mensagens síncronas são de alta prioridade e são as primeiras a serem enviadas no EC em instantes temporais bem defendidos calculados apartir da informação na TM, as mensagens assíncronas são enviadas no restante tempo do EC, podendo ou não possuir requisitos de *tempo real*.

[1] envio de uma mensagem para todos os nós na rede

5.4 Tipos de mensagens

O protocolo *FTT-SE* utiliza para comunicar entre os diferentes nós diferentes tipos de mensagens que assentam no campo de dados(*Data*), do protocolo *Ethernet*, com o *Type* `0x8ff0` (Figura 5.2).

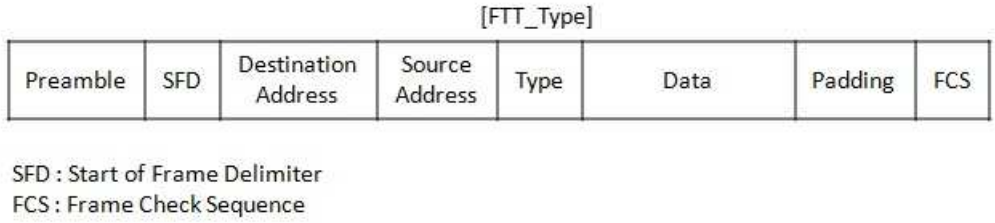

Figura 5.2: Estrutura de um pacote *Ethernet*

Na estrutura dos pacotes FTT existe um campo, de dois *bytes* que identifica o tipo de mensagem e está descrito como:

`0x0000` *Trigger Message*

`0x0001` *Syncronous Data Message*

`0x0002` *Asynchronous Data Message*

`0x0003` *Asynchronous Status Message*

`0x0004` *Idle*

`0x0005` *Plug n Play*

5.4.1 *Trigger Message*

A *Trigger Message* é enviada pelo *Master* para todos os nós(*broadcast*) e possui informações sobre todas as mensagens a serem enviadas pelos diferentes nós durante o corrente EC, a sua estrutura é (Figura 5.3):

Campo	**Tamanho(*bytes*)**	**Descrição**
FTT Type	2	Tipo de mensagem
Seq No	1	Numero de sequência
Flags	1	conjunto de *flags*
NSM	2	Número de mensagens Síncronas
NAM	2	Número de mensagens Assíncronas
MSG Index	4	Informação sobre mensagem
ID	2	Identificador da mensagem
Frag No	2	Número de Fragmentação

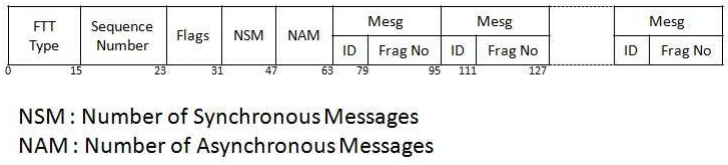

Figura 5.3: Estrutura de uma *Trigger Message*

5.4.2 *Syncronous Data Message*

A *Syncronous Data Message* transporta informações que precisam de ser transmitidas em tempo real ,a sua estrutura é (Figura 5.4):

Campo	**Tamanho(*bytes*)**	**Descrição**
FTT Type	2	Tipo de mensagem
ID	2	Identificador da mensagem
Seq No	1	Numero de sequência
Flags	1	conjunto de *flags*
Frag Seq No	2	Número de Fragmentação
Data	Variável	Dados a serem transmitidos

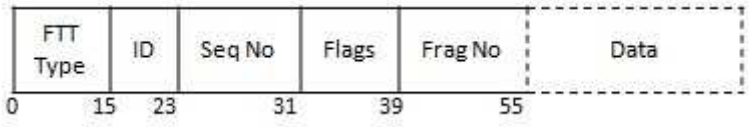

Figura 5.4: Estrutura de uma *Syncronous Data Message* ou *Asyncronous Data Message*

5.4.3 *Asyncronous Data Message*

A *Asyncronous Data Message* transporta informações "assíncronas", podendo possuir ou não requisitos de pontualidade, a sua estrutura é igual à *Syncronous Data Message*(Figura 5.4).

5.4.4 *Asynchronous Status message*

Este tipo de mensagem transporta informações de estado e configuração do sistema e dos dispositivos na rede, a sua estrutura não é relevante no âmbito deste projecto.

5.4.5 *Idle*

Sem relevância no âmbito do projecto.

5.4.6 *Plug N Play*

Sem relevância no âmbito do projecto.

Capítulo 6

Funcionalidades do dissector *FTT-SE*

6.1 Identificação e Processamento das mensagens

O *dissector* está preparado para identificar todos os tipos de mensagens pertencentes ao protocolo *FTT-SE*, no entanto só é capaz de processar e interrelacionar as mensagens dos tipos: *Trigger Message*, *Synchronous Data Message* e *Asyncronous Data Message*.

6.1.1 *Summary*

Na janela summary são mostradas as informações que sumarizam o pacote analisado. Em primeiro lugar é identificado o protocolo na coluna *Protocol*. , na coluna *Info* é mostrado o tipo de mensagem e em seguida informações especificas de cada tipo de mensagem:

Trigger Message

- ***Sync Msgs*** - Identificação do número de *Synchronous Data Messages* a serem enviadas no presente *EC*;
- ***Async Msgs*** - Identificação do número de *Asynchronous Data Messages* a serem enviadas no presente *EC*;

Synchronous Data Message

- ***Id*** - Identificação do *Id*;
- ***Frag no*** - Identificação do número de Fragmentação;

Asynchronous Data Message

- ***Id*** - Identificação do *Id*;
- ***Frag no*** - Identificação do número de Fragmentação;

6.1.2 Informações na *Protocol Tree*

Na *Protocol Tree* são mostradas informações detalhadas sobre a mensagem a ser analisada e sobre a relação entre mensagens. As informações são asseguir descritas por tipo de mensagem.

Trigger Message

- Identificação do ***Sequence Number***;
- Identificação do campo ***flags***;
- Identificação das ***Synchronous Messages***;

  Identificação da quantidade;

  Identificação do ***ID*** e do ***Fragmentation Number*** de cada mensagem a ser enviada no ***EC***;
- Identificação das ***Asynchronous Messages***;

  Identificação da quantidade;

  Identificação do ***ID*** e do ***Fragmentation Number*** de cada mensagem a ser enviada no ***EC***;

Synchronous Data Message

- Identificação do ***ID***;
- Identificação do ***Sequence Number***;
- Identificação do ***Fragmentation Number***;
- Identificação da ***Frame*** da *Trigger Message* que inicia o *EC* em que a mensagem se insere;
- Calculo do tempo passado desde o inicio do *EC*;

Asynchronous Data Message

- Identificação do ***ID***;
- Identificação do ***Sequence Number***;
- Identificação do ***Fragmentation Number***;

- Identificação da ***Frame*** da *Trigger Message* que inicia o *EC* em que a mensagem se insere;
- Calculo do tempo passado desde o inicio do *EC*;

6.2 Detecção de Erros

Um aspecto importante num *dissector* é a analise funcional do protocolo, dai que seja interessante detectar erros para estudar o funcionamento dos nós.

Os erros mostrados são:

Mensagem não incluída na *Trigger Message* - Este erro dá-se quando uma mensagem é transmitida com um ***ID*** que não existe na *Trigger Message* correspondente ao *EC*;

Mensagens Síncronas por enviar - Este erro dá-se quando se detecta o inicio de um novo ***EC*** (chegada de uma nova *Trigger Message*) sem que tenham sidas detectadas todas as ***Synchronous Data Messages*** correspondentes ao *EC* anterior;

Mensagens Assincronias por enviar - Este erro dá-se quando se detecta o inicio de um novo ***EC*** (chegada de uma nova *Trigger Message*) sem que tenham sidas detectadas todas as ***Asynchronous Data Messages*** correspondentes ao *EC* anterior;

Erro no número de sequência da *Trigger Message* - Este erro dá-se quando se detecta a chegada de uma ***Trigger Message*** com o ***Sequence Number*** diferente do anterior incrementado de um;

Para acedermos a uma lista detalhada de todos os erros numa captura podemos aceder a *Expert Info* ou *Expert Info Composite* em *Analyze* na barra de menus.

Expert Info - Lista sequencial de todos os erros (Figura 6.1);

Expert Info Composite - Lista de erros agrupados por tipo de erro (Figura 6.2)

6.3 Filtragem

Uma das ferramentas do *Wireshark* é a interface de *Display Filters* que tem a capacidade de filtrar os pacotes mostrados no *GUI* por parâmetros embebidos no dissector.

Os parâmetros implementados neste *dissector* estão a seguir descritos:

Figura 6.1: Expert Info

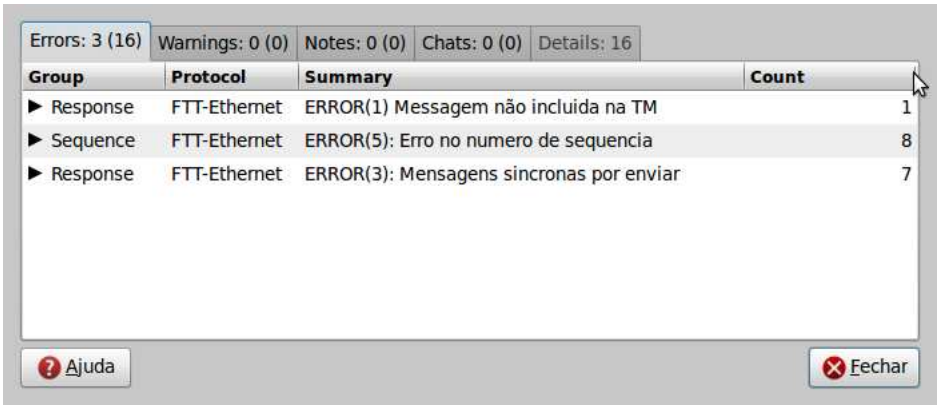

Figura 6.2: Expert Info Composite

6.3.1 Tipo

Aplicável a todas as mensagens do protocolo.

Sintaxe	Formato	Base	Intervalo de Dados
`ftt.type`	UINT16	*Value String*	0 - "*Trigger Message* 1 - "*Synchronous Data Message*" 2 - "*Asynchronous Data Message*" 3 - "*Asynchronous Signaling Message*" 4 - "*Idle Message*" 5 - "*Plug n Play Message*"

6.3.2 *Sequence Number*

Sintaxe	Formato	Base	Intervalo de Dados
`ftt.seq_no`	UINT8	decimal	0 a 255

6.3.3 *Flags*

Aplicável a *Trigger Message*, *Synchronous Data Message* e *Asynchronous Data Message*.

Sintaxe	Formato	Base	Intervalo de Dados
`ftt.flag_s`	UINT8	hexadecimal	0x00 a 0xff

6.3.4 *Número de Mensagens Síncronas*

Aplicável a *Trigger Message*.

Sintaxe	Formato	Base	Intervalo de Dados
`ftt.nsm`	UINT16	decimal	0 a 65535

6.3.5 *Número de Mensagens Assíncronas*

Aplicável a *Trigger Message*.

Sintaxe	Formato	Base	Intervalo de Dados
`ftt.nam`	UINT16	decimal	0 a 65535

6.3.6 *Frame* correspondente

Aplicável a *Synchronous Data Message* e *Asynchronous Data Message*.

Sintaxe	Formato	Base	Intervalo de Dados
`ftt.TMframe`	FRAME_NUM	decimal	0 a ...

6.3.7 Erro

Aplicável a *Trigger Message*, *Synchronous Data Message* e *Asynchronous Data Message*.

Sintaxe	Formato	Base	Intervalo de Dados
`ftt.error`	UINT16	*Value String*	0 - "*Comprimento Mínimo*"[1] 1 - "*Mensagem não incluída na TM*" 2 - "*Mensagem duplicada*"[1] 3 - "*Mensagens síncronas por enviar*" 4 - "*Mensagens assincronias por enviar*" 5 - "*Erro no numero de sequência*"

[1]

6.3.8 *Número de Mensagens Síncronas por enviar*

Aplicável a *Trigger Message*.

Sintaxe	Formato	Base	Intervalo de Dados
`ftt.sleft`	UINT16	decimal	0 a 65535

6.3.9 *Número de Mensagens Assíncronas por enviar*

Aplicável a *Trigger Message*.

Sintaxe	Formato	Base	Intervalo de Dados
`ftt.asleft`	UINT16	decimal	0 a 65535

6.3.10 *Latência desde o inico do EC*

Aplicável a *Synchronous Data Message* e *Assynchronous Data Message*.

Sintaxe	Formato	Base	Intervalo de Dados
`ftt.TMtime`	Tempo Relativo	Tempo em segundos	Não aplicavel

6.3.11 *Id* da Mensagem

Aplicável a *Synchronous Data Message* e *Assynchronous Data Message*.

Sintaxe	Formato	Base	Intervalo de Dados
`ftt.id`	UINT16	hexadecimal	0x0000 a 0xFFFF

6.3.12 *Id* de *Asynchronous Data Message* na *Trigger Message*

Aplicável a *Trigger Message*, identifica se existe o *Id* no *EC*.

Sintaxe	Formato	Base	Intervalo de Dados
`ftt.AMsgId`	UINT16	hexadecimal	0x0000 a 0xFFFF

6.3.13 *Id* de *Synchronous Data Message* na *Trigger Message*

Aplicável a *Trigger Message*, identifica se existe o *Id* no *EC*.

Sintaxe	Formato	Base	Intervalo de Dados
`ftt.SMsgId`	UINT16	hexadecimal	0x0000 a 0xFFFF

[1]Não Implementado

6.3.14 *Número da Trigger Message na captura*

Aplicável a *Trigger Message*, este campo exprime numa ordem sequencial a ordem da *Trigger Message* na captura.

Sintaxe	Formato	Base	Intervalo de Dados
`ftt.TMno`	UINT16	Decimal	0 a 65535

6.3.15 Utilização dos filtros

Para aplicarmos filtros à captura que está a ser analizada está disponível na barra de ferramentas, em cima, uma interface que permite que se escrevam *Display Filters.*(No *input* dos filtros existe um um mecanismo de *auto-completion*)(Figura 6.3).

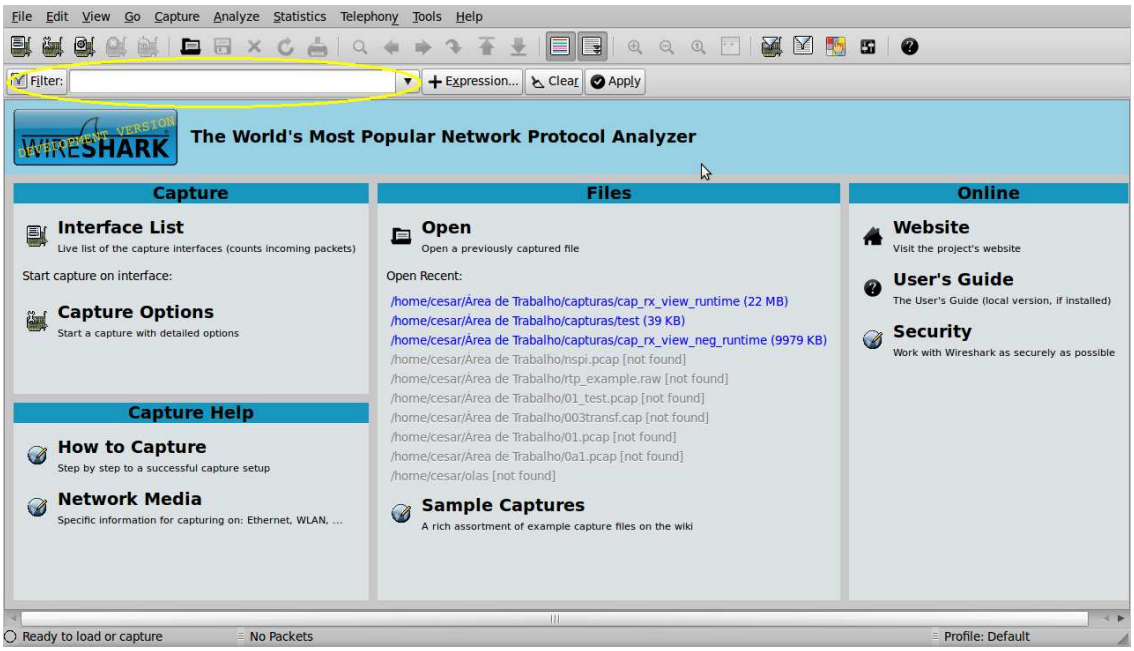

Figura 6.3: Interface de *input* de filtros

Podemos verificar a existência de um campo do protocolo na *frame* ao colocarmos apenas o campo no filtro, p.e se quisermos filtrar as mensagens com erro, podemos colocar no filtro apenas `ftt.error`(Figura 6.4).

Podemos utilizar os operadores seguintes em conjunto com os campos previamente descritos, de acordo com o formato e base da variável associada:

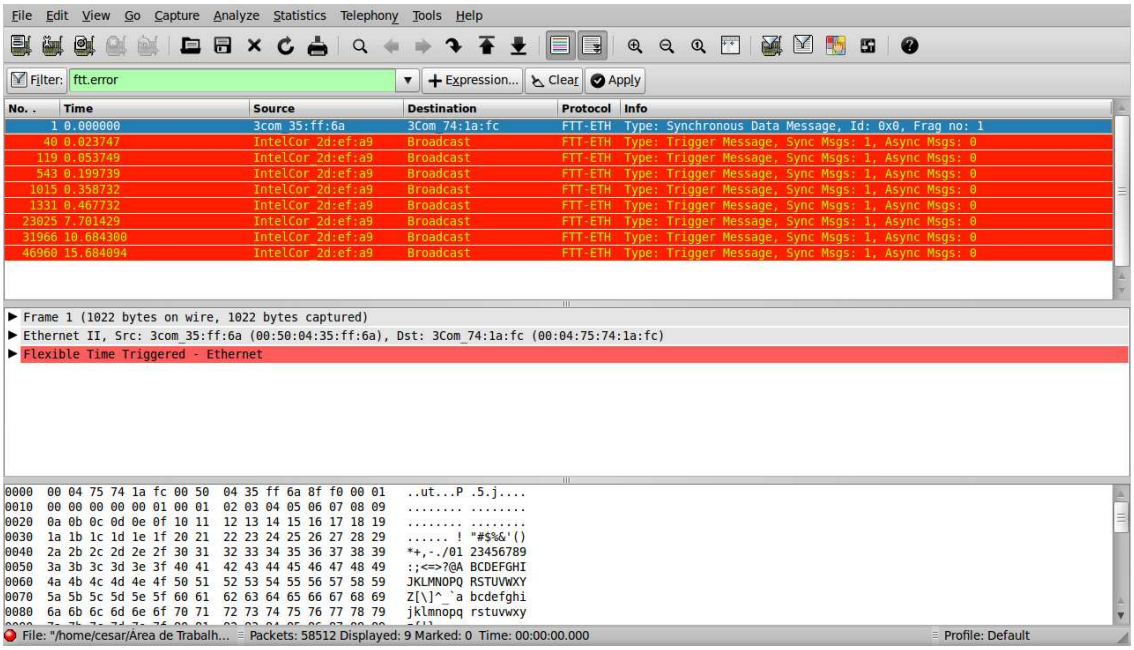

Figura 6.4: Filtro de existência de erros

Operadores	Significado
> ou *gt*	Maior Que
>= ou *ge*	Maior ou Igual
< ou *lt*	Menor Que
<= le *le*	Menor ou Igual
== ou *eq*	Igual a
!= ou *nq*	Não Igual a
contains	Uma *string* ou é encontrada dentro de outra
matches	Uma expressão corresponde a uma *string*
& ou *bitwise_and*	*"Bitwise and"* para testar *bits* especificos

Várias relações podem ser combinadas com os operadores lógicos *or*(||) e *and*(&&)

6.3.16 Exemplos de utilização de filtros

Exemplo 1

Se se pretender filtrar a captura pelas mensagens com o *Id* igual a `0x0000` deve-se colocar na caixa de dialogo a expressão: `ftt.id == 0x0000`(Figura 6.5);

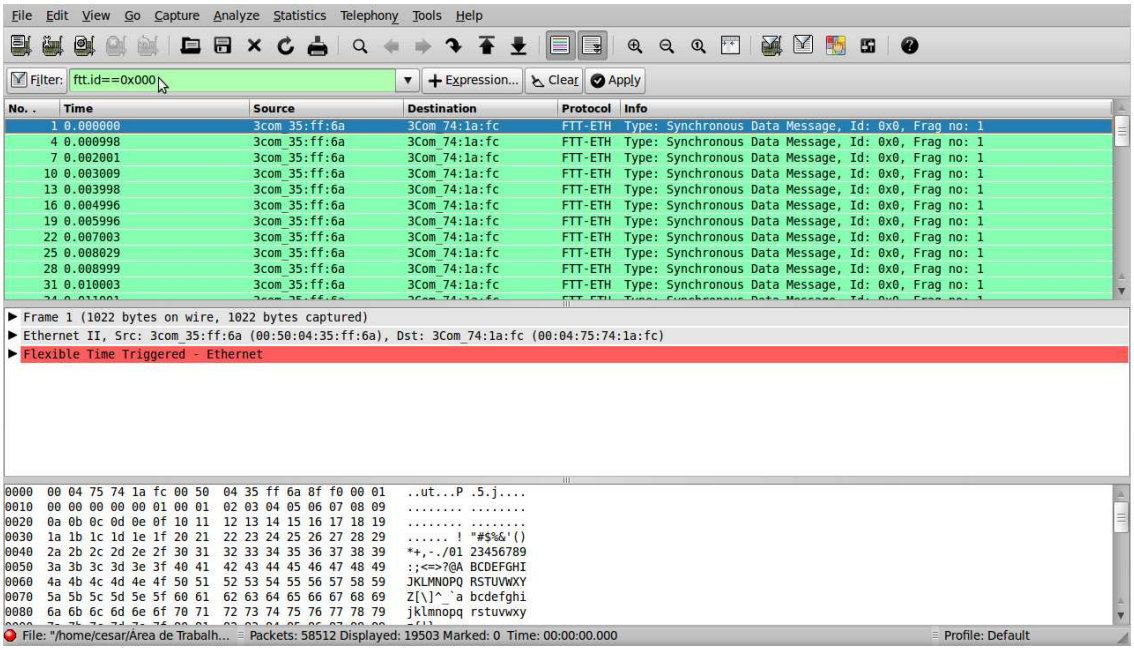

Figura 6.5: Exemplo de filtragem por *Id*

Exemplo 2

Se se pretender filtrar a captura pelas mensagens do tipo *Synchronous Data Message* deve-se colocar na caixa de dialogo a expressão: `ftt.type == 1` ou `ftt.type == "Synchronous Data Message"`(Figura 6.6);

Exemplo 3

Se se pretender filtrar a captura pelas *Trigger Messages* com referência a mais que duas *Asynchronous Data Message* deve-se colocar na caixa de dialogo a expressão: `ftt.nam > 2` (Figura 6.7);

Exemplo 4

Se se pretender filtrar a captura pelas mensagens com latência menor do que 0.00002 segundos e do tipo *Synchronous Data Message* deve-se colocar na caixa de dialogo a expressão: `ftt.TMtime < 0.00002 && ftt.type == 1`(Figura 6.8);

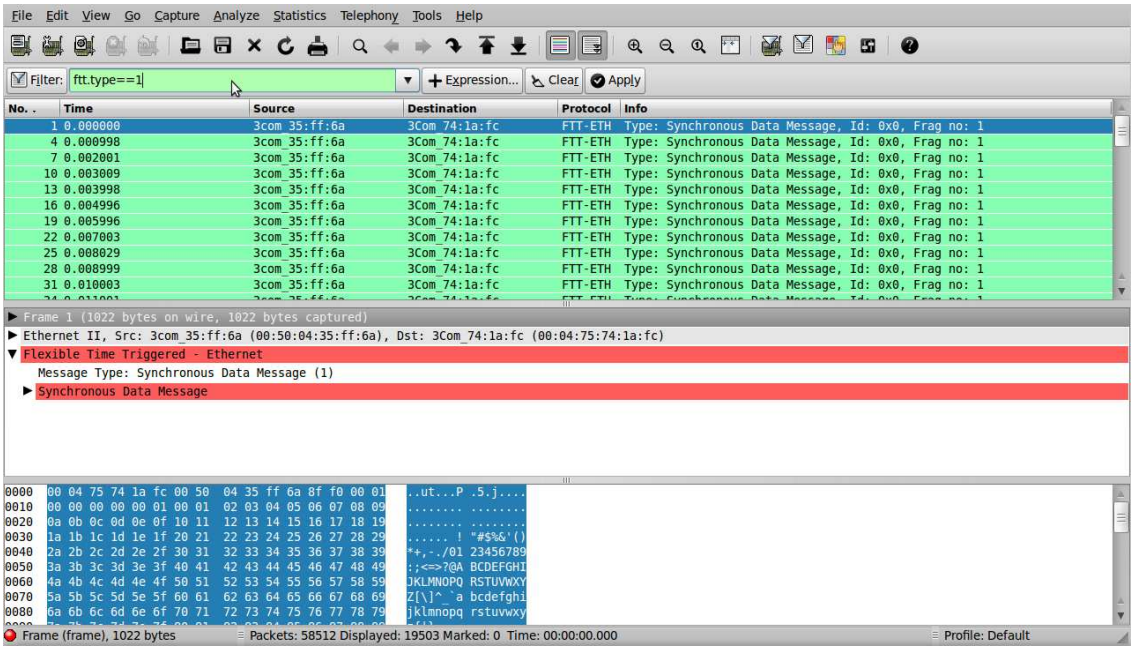

Figura 6.6: Exemplo de filtragem por tipo de mensagem

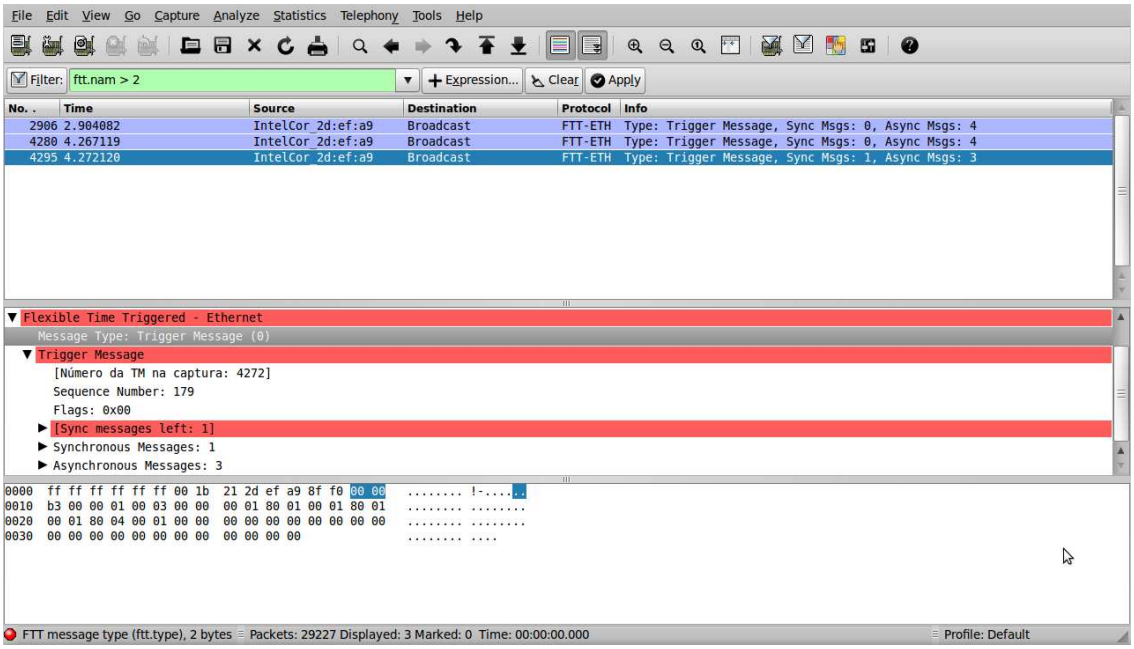

Figura 6.7: Exemplo de filtragem por número de mensagens Assíncronas

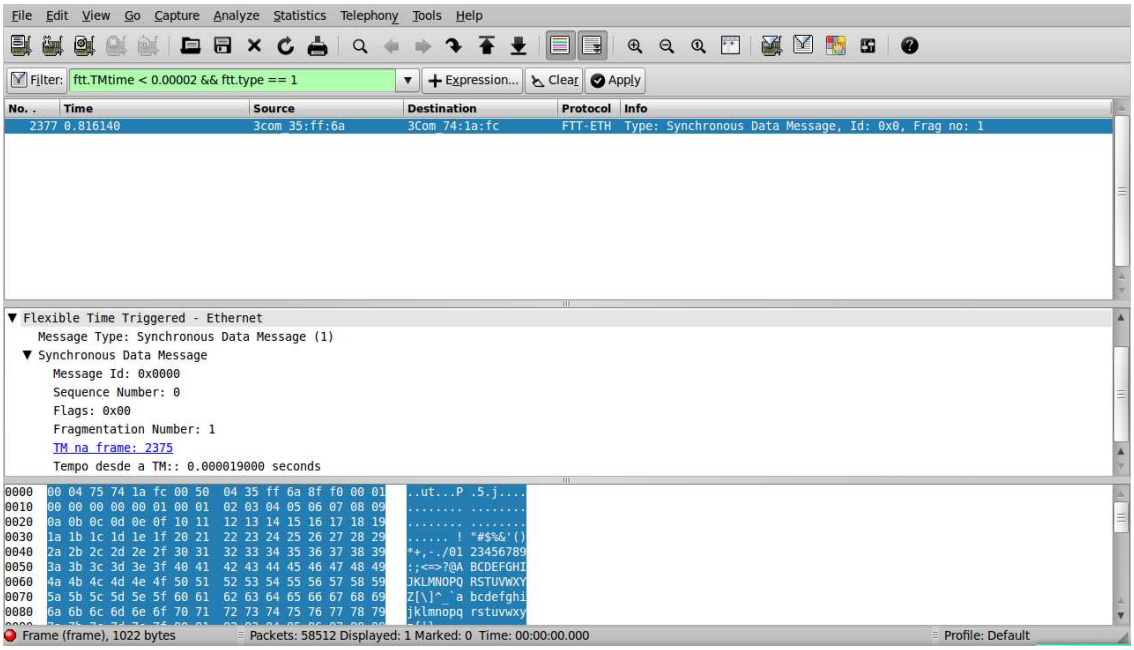

Figura 6.8: Exemplo de filtragem por tempo e por tipo de mensagem

Capítulo 7

Descrição Funcional do código do *dissector* para *FTT-SE*

A análise deste capitulo deve ser feita acompanhando o código fonte do *dissector*

7.1 Função *proto_register_ftt()*

Esta função é caracterizada por registar o *dissector* no "*motor*" do *Wireshark* , daí que sejam nela declaradas informações importantes para o funcionamento correcto do *plug-in*.

O *array* `hf_register_info hf[]` é responsável pela definição e registo de todos os itens que serão capazes de ser filtrados, bem como a maneira como são mostrados na *protocol tree*.

O *array* `gint *ett[]` é responsável pela definição das variáveis que possuem o estado dos ramos mostrados na *protocol tree*.

Por fim são executadas funções que são responsáveis por registar as características do *dissector* no *Wireshark* .

7.2 Função *proto_reg_handoff*

Esta função regista no *Wireshark* o ponto de entrada quando se detecta uma mensagem do protocolo, função `dissect_ftt`, bem como o elemento que despoleta o *dissector* , o campo *type* do protocolo *Ethernet* ser igual a `0x8FF0`. É tambem registada a função `ftt_init_routine()` que é executada sempre que se inicia uma nova captura.

7.3 Função *ftt_init_routine()*

Esa função é chamada sempre que se inicia uma nova captura. Esta é responsável por definir os valores iniciais de variáveis de estado e alocar o *array* de estruturas que vão armazenar as informações de cada *EC*.

7.4 Função *dissect_ftt()*

Esta função é chamada sempre que o *Wireshark* detecta uma mensagem com o *ethertype* igual a `0x8FF0`, esta tem acesso às estruturas *tvb* e *pinfo*, já anteriormente descritas, e que contém toda a informação necessária à analise do protocolo.

A função começa por colocar informações na janela *summary*, o *FTT-ETH* na coluna *Protocol*; asseguir coloca informação sobre o tipo de mensagem na coluna *Info*.

O próximo passo é colocar informação na janela *Protocol tree*, portanto, cria-se então a *tree* para o protocolo *FTT*, e depois é colocada na *tree* a informação sobre o tipo de mensagem.

Identificado o tipo , o processamento da mensagem é enviado para uma função especifica.

7.5 Função *dissect_TM()*

Esta função é chamada pela `dissect_FTT()` sempre que esta detecta uma mensagem do tipo *Trigger Message* e tem acesso às mesmas estruturas que a anterior, mais o acesso à *tree* do protocolo.

A função começa por recolher a informação estrutural da mensagem, o *sequence number*, as *flags*, o número de *Synchronous Data Messages* e o número de *Asynchronous Data Messages*. Estas informações juntamente com outras informações da mensagem são colocadas num *array* de estruturas em que cada posição representa um *EC*. Cada posição do contem as seguintes informações:

- Número da *frame* da *Trigger Message*;
- *Tempo* da mensagem;
- *Sequence Number*;
- Número de *Synchronous Data Messages* por enviar(para efeito de detecção de erros);
- Número de *Asynchronous Data Messages* por enviar(para efeito de detecção de erros);

- Árvore binária com as informações de cada *Synchronous Data Message*;
- Árvore binária com as informações de cada *Asynchronous Data Message*;

Cada posição da árvore binária é indexada pelo *Id* da mensagem e possui as seguintes informações:

- Numero de fragmentos;
- (liberdade para colocar mais informação);

Consoante o número de cada tipo de mensagem a ser transmitido são recolhidas as informações de cada mensagem: o *Id* e o *Fragmentation Number* e são guardadas em estruturas adequadas.

O próximo passo é discriminar em *summary* , na coluna *Info*, a quantidade de mensagens de cada tipo que deverão ser enviadas no currente *EC*.

É feito então algumas verificações para detectar erros, e depois são colocadas na *Protocol Tree* as informações já descritas para *Trigger Messages*.

7.6 Função *dissect_SDM()*

Esta função é chamada pela `dissect_FTT()` sempre que esta detecta uma mensagem do tipo *Synchronous Data Message* e tem acesso às mesmas estruturas que a função *dissect_FTT()* , mais o acesso à *tree* do protocolo.

É feito então a identificação dos conteúdos presentes na mensagem, e colocado na coluna *Info* da janela *Summary* o *Id* e o *Fragmentation Number*.

De seguida é verificado através do tempo da mensagem , a que *EC* esta pertence, para depois ser verificada a sua existência na árvore binária correspondente.

A seguir são processadas e mostradas na *protocol tree* as informações já descritas para este tipo de pacote.

7.7 Função *dissect_ADM*

Função funcionalmente equivalente à `dissect_SDM` para *Asynchronous Data Messages*.

7.8 Função *checkMem()*

Esta função é chamada na função `dissect_TM()` pois é nela que se guardam informações na memória. A função verifica se a memória alocada já está toda ocupada e caso se verifique, esta realocará os ponteiros de modo a que seja possível guardar mais informações na memória.

7.9 Função *findTM()*

Esta é uma função auxiliar que é responsável por encontrar o índice adequado do *array* de estruturas associadas às *Trigger Messages*, através do tempo da mensagem que lhe é passada.

Capítulo 8

O sistema TAP

8.1 O que é

O *Wireshark* tem em si implementado o sistema *TAP* que permite que sejam executadas notificações de informação, geradas a partir de eventos de alguns pacotes de alguns protocolos.

É implementado no *dissector* um mecanismo que passa informações para a interface *TAP* que é depois recolhida por um *tap listener*.

8.2 O *tap-fttstat*

Com base no mecanismo *TAP* foi desenvolvida uma ferramenta que produz informações estatísticas, sobre o protocolo *FTT*. A utilização desta ferramenta está apenas disponivel no *tshark*.

8.2.1 Descrição das informações fornecidas pela ferramenta

Quando a ferramenta é utilizada, esta produz informações importantes que são asseguir descritas.

8.2.2 Latência nos *ECs*

Por cada *EC* que é iniciado e indexado por uma *Trigger Message* é impressa uma linha com a informação do número da *Trigger Message* na captura e a latência desta ao primeiro pacote recebido do correspondente *EC*.

8.2.3 Estatísticas Sumárias

Latency Stats

Esta secção das estatísticas mostra a o mínimo, o máximo a média e o desvio padrão das latências do primeiro pacote em cada *EC*

Btween EC Stats

Esta secção mostra o mínimo, o máximo, a média e o desvio padrão das latências entre *ECs*, os tempos entre *Trigger Messages* sucessivas.

Selected Id Stats

Esta secção mostra o mínimo, o máximo, a média e o desvio padrão das latências entre mensagens com o mesmo *Id* nos diferentes *ECs*.

8.3 Utilização da ferramenta

Para a utilizarmos esta ferramenta temos de ter um instância de terminal aberta e executar o seguinte comando:

`tshark [opções de captura] -z fttstat,0x<Id>`

no campo *Id* colocamos o *Id*(em hexadecimal) que queremos que seja analisado nas *Selected Id Stats* ou `-1` caso não seja necessário o processamento destas informações.

Capítulo 9

Resultados experimentais

9.1 Testes preliminares

9.1.1 Desenvolvimento

Durante o estudo do desenvolvimento do *Wireshark* foi implementado um *dissector* para testes para um protocolo imaginário ,"Caesar Protocol" com a estrutura:

Tipo - Tipo de mensagem do protocolo, sem sentido funcional apenas para efeito de testes.

Valor	Descrição
01	*Initialize*
02	*Terminate*
03	*Data*

Flags - *Flags* indicadoras de características do protocolo, apenas o *bit menos significativo* tem informação significativa.

LSB	Descrição
0	*Reply*
1	*Request*

Sequence Number - Numero de sequência das mensagens, as mensagens *Request* e *Reply* correspondentes têm de ter o mesmo *sequence number*.

Este protocolo é implementado directamente sobre *Ethernet* com o *type* `0x1001`.

As funcionalidades deste *dissector* são:

- Identificação do tipo de mensagem;
- Identificação e processamento das *flags*;
- Relacionamento entre *Requests e Replys* em tempo e numero de *frame*;

- Detecção de erro se existir uma *Reply* sem ter sido recebido a *Request* correspondente;

Os testes ao funcionamento deste *dissector* foram feitos manualmente com o programa ***PackETH***[1] , uma ferramenta capaz de gerar pacotes *Ethernet* e envia-las em qualquer interface de rede disponível no computador, os testes foram feitos maioritariamente através da interface *loopback*.

9.1.2 Resultados

Este *dissector* serviu para aprender a trabalhar com a *API* do *Wireshark* e a ter noções sobre como indexar estruturas no programa. Foi bastante importante para compreender como se deve e e por onde começar a desenvolver um *dissector* .

Os resultados dos testes foram satisfatórios.

9.2 Testes ao *dissector FTT-SE*

Os testes a este *dissector* foram feitos inicialmente com o recurso ao programa ***PackETH***, tendo ainda sido escrito por mim um programa simples que gerava em texto uma *Trigger Message* com os seguintes parâmetros configuráveis:

- *Sequence Number*;
- *Flags*;
- Numero de *Synchronous Data Messages*;
- Numero de *Asynchronous Data Messages*;

sendo o resto dos parâmetros gerados aleatoriamente.

Durante o desenvolvimento foram sido implementadas funcionalidades ao *dissector* e sendo testadas as possíveis situações de erro.

Foram testadas as situações:

Sequence Number - Foram testadas situações de erro de *Trigger Messages* consecutivas com *Sequence Number* não sequenciais;

Numero de Mensagens - Foram testadas situações de erros de *Trigger Messages* com variados números de mensagens Síncronas e Assíncronas em que algumas todas ou nenhumas foram respondidas;

Mensagens não Incluídas - Foram testadas situações de envio de mensagens antes de existir alguma *Trigger Message*, e de mensagens Síncronas e Assíncronas não anunciadas.

[1] Programa disponível em http://packeth.sourceforge.net/ sob a licença GNU-GPL

Para alem destes testes foram testadas situações sem erros. O resultados destes testes estiveram de acordo com as funcionalidades propostas para o dissector.

Capítulo 10

Conclusões e Trabalho Futuro

10.1 Conclusões

Visto que os protocolos de tempo real como o *FTT-SE* são usados em ambientes industriais, no controlo de sistemas de produção, estes têm de ser muito fiaveis, pois podem por em causa toda uma linha de produção. Este *dissector* vem fornecer uma ferramenta de análise e *debbuging* das redes implementadas sobre *FTT-SE*, sendo particularmente útil para detectar falhas do sistema.

10.2 Trabalho Futuro

Sendo sempre possível melhorar, como temos observado na evolução da tecnologia que nos rodeia, este trabalho tem muita margem por onde possa ser melhorado.

Em primeiro lugar é possível conferir mais robustez ao código, e adicionar mais informações detalhadas sobre os pacotes(p.e. a interpretação das *flags*) e sobre a interrelação entre eles.

Não tendo sido adicionado ao *dissector* funcionalidades para processar as *Asynchronous Status Message* e *Assynchronous Signaling Message*, é sempre possível adicioná-las.

Capítulo 11

Manual de Instalação

11.1 Ambiente de desenvolvimento

Este trabalho foi desenvolvido no sistema operativo *Ubuntu* na versão *9.04 Jaunty Jackalope* com o *kernel linux 2.6.28-11-generic*, na edição do código fonte foi utilizado o *IDE Geany* na versão *0.16 Argon*.

11.2 Instalação do *Wireshark*

Foi utilizado o *Wireshark Development Version Network Protocol Analyzer 1.3.0* compilado apartir do código fonte, que está disponivel em `http://www.wireshark.org/download/src/all-versions/wireshark-1.3.0.tar.gz`

A instalação do programa requer que se tenha o pacote do *link* anterior descompactado e que se tenha uma instância de terminal com o seu *working directory* na pasta em que temos o código fonte, a seguir temos de seguir as seguintes instruções no terminal:

1. `sudo ./autogen.sh` - este comando faz com que se criem alugumas makefiles- caso faltem pacotes instalados no sistema operativo a *script* notifica quais estão em falta;

2. `sudo ./configure -disable-warnings-as-errors` - esta script configura as opções do programa, a opção `-disable-warnings-as-errors` faz com que as *warnings* não sejam tratadas como erros e, consequentemente não parem a compilação - caso faltem pacotes instalados no sistema operativo a *script* notifica quais estão em falta;

3. `sudo make install` - este comando faz a instalação propriamente dita do *Wireshark* executando todas as makefiles e compilando todo o código fonte necessário;

11.3 Instalação do *dissector* FTT-ETH como *plug-in*

Os directórios referidos nesta secção tem como directório base do *Wireshark* instalado apartir do código fonte.

11.3.1 O directório do *plug-in* e os seus ficheiros

Deve ser criada uma pasta `plugins/ftt` onde devem ser colocados os ficheiros: `AUTHORS`, `COPYING`, `Changelog`, `Makefile.am`, `Makefile.common`, `Makefile.nmake`, `moduleinfo.h`, `moduleinfo.nmake`, `packet-ftt.c`, `plugin.rc.in` `ftt-global.h`, `packet-ftt.h`, que devem ser disponibilizados em anexo com este documento.

11.3.2 Mudanças em ficheiros já existentes do *Wireshark*

Para finalizar-se a instalação do *plug-in* tem-se de alterar os seguintes ficheiro:

- `configure.in`
- `epan/Makefile.am`
- `Makefile.am`
- `Makefile.nmake`
- `packaging/nsis/Makefile.nmake`
- `packaging/nsis/wireshark.nsi`
- `plugins/Makefile.am`
- `plugins/Makefile.nmake`

11.3.3 Mudanças no `plugins/Makefile.am`

É necessário mudar a directiva `SUBDIRS` (por ordem alfabética) de modo a reflectir a adição do plugin:

```
1  SUBDIRS = $(_CUSTOM_SUBDIRS_) \
2  ...
3  gryphon \
4  irda \
5  ...
6  ftt \
```

O directório deve ser inserido por ordem alfabética.

11.3.4 Mudanças no `plugins/Makefile.nmake`

Deve-se adicionar, por ordem alfabética, em process-plugins o seguinte:

```
1  cd ftt
2  $(MAKE) /$(MAKEFLAGS) -f Makefile.nmake $(PLUGIN_TARGET)
3  cd ..
```

A seguir à adicionamos em install-plugins:

```
1   ...
2   xcopy plugins\ftt\*.dll $(INSTALL_DIR)\plugins\$(VERSION) /d
3   cd plugins
4   if exist Custom.nmake $(MAKE) /$(MAKEFLAGS) -f Custom.nmake
        install-plugins
```

11.3.5 Mudanças na `Makefile.am`

Deve se adicionar o plugin (por ordem alfabética) em `plugin_ldadd:`

```
1 if HAVE_PLUGINS
2
3 plugin_ldadd = \
4   ...
5   -dlopen plugins/gryphon/gryphon.la      \
6   -dlopen plugins/irda/irda.la    \
7   ...
8   -dlopen plugins/ftt/ftt.la      \
9   ...
```

11.3.6 Mudanças na `configure.in`

Deve-se adicionar (por ordem alfabética) a *Makefile* do *plug-in* em `AC_OUTPUT`

```
1   AC_OUTPUT(
2     ...
3     plugins/gryphon/Makefile
4     plugins/irda/Makefile
5     ...
6     plugins/ftt/Makefile
7     ...
8     ,)
```

11.3.7 Mudanças ao `epan/Makefile.am`

Deve-se adicionar o caminho relativo para o plugin (por ordem alfabética) em plugin_src:

```
1     plugin_src = \
2         ...
3       ../plugins/gryphon/packet-gryphon.c \
4       ../plugins/irda/packet-irda.c \
5         ...
6       ../plugins/ftt/packet-ftt.c \
7         ...
```

11.3.8 Comandos para instalar o *dissector*

O próximo passo é despoletar a compilação do plug-in, a primeira compilação deve ser feita apartir da raiz, verificando todas as *makefiles* do *Wireshark* por mudanças, deve se então abrir uma instancia de terminal e executar:

```
sudo make install
```

As compilações subsequentes em caso de alteração do *dissector* podem ser feitas em `plugins/ftt`, sendo esta compilação mais rápida que a anterior, deve-se então executar de novo:

```
sudo make install
```

11.3.9 Instalação dos filtros de cores

Ainda nos ficheiros disponibilizados juntamente com este documento está um ficheiro com o nome `color`, que contém as regras de cores para o protocolo FTT. A sua instalação pressupõe a execução do *Wireshark* em modo de *super utilizador*.

Depois de iniciado o programa deve se aceder na barra de menus a `View->Coloring Rules`, na janela que irá aparecer deve se escolher a função `Import` e seleccionar o ficheiro `color`.

11.4 Instalação *protocol tap fttstat*

Para instalar o *tap* tem-se de alterar o ficheiro `MAKEFILE` e executar os seguintes passos:

- adicionar em `am__objects_8` (por ordem alfabética) a seguinte linha: `tshark-tap-fttstat.$(OBJEXT)`
- adicionar em `TSHARK_TAP_SRC` (por ordem alfabética) a seguinte linha: `tap-fttstat.c\`
- a seguir a em `distclean-compile:` `-rm -f *.tab.c` (por ordem alfabética) a seguinte linha: `include ./$(DEPDIR)/tshark-tap-fttstat.Po`
- adicionar na secção a seguir à ultima o código presente no ficheiro `addMkf` fornecido com este documento;

para concluir a instalação executasse o comando `sudo make install` na pasta raiz do *Wireshark* .

Bibliografia

[1] "Ethernet - Wikipedia, the free encyclopedia."

[2] C. E. Spurgeon, *Ethernet (The Definitive Guide)*. O'Reilly, 2000.

[3] P. Pedreiras, L. Almeida, and J. A. Fonseca, *The Quest for Real-Time Behavior in Ethernet*. CRC Press, 2004.

[4] P. Pedreiras, P. Gai, L. Almeida, and G. C. Buttazzo, "Ftt-ethernet: A flexible real-time communication protocol that supports dynamic qos management on ethernet-based systems," *IEEE Transactions on Industrial Informatics*, vol. vol. 1, n.* 3, pp. –162, Ago. 2005.

[5] A. Orebaugh, G. Ramirez, J. Burke, and L. Pesce, *Wireshark & Ethereal Network Protocol Analyzer Toolkit (Jay Beale's Open Source Security)*. Syngress Publishing, 2006.

[6] U. Lamping and E. Sharpe, Richard Warnicke, *Wireshark User's Guide 29805 for Wireshark 1.2.0*, 2008.

[7] U. Lamping, *Wireshark Developer's Guide 29805 for Wireshark 1.2.0*, 2008.