Easy, Smart and Modern...

# Manual de Software Desenvolvido para o Miavita

Frederico Gonçalves
MEIC-T, 57843
frederico.lopes.goncalves@gmail.com

20 de Junho de 2012

# Conteúdo

**1 Preparação do Sistema** **2**
1.1 Cross-compilation . . . 2
1.2 Código do Projecto . . . 3

**2 Código** **4**
2.1 Kernel Sender e Programa Servidor . . . 4
2.2 RT2501 - Driver Modificado . . . 6
2.3 Kernel Modificado . . . 7

**3 Especificação do Pacote de Dados** **8**
3.1 Binário . . . 8
3.2 ASCII . . . 9

**4 Como Usar os Componentes** **10**

**5 Framework de Interceptores** **12**
5.1 Interceptor de agregação . . . 12
5.1.1 Nível Aplicação . . . 12
5.1.2 Nível Rede . . . 13
5.2 Interceptor de desagregação . . . 13

# Capítulo 1

# Preparação do Sistema

## 1.1 Cross-compilation

Esta secção demonstra como compilar programas para o ARM a partir de uma outra plataforma (cross-compiling). As vantagens são óbvias. O processo de compilação é muito mais rápido e por vezes não é de todo possível compilar código no ARM, como é o caso dos módulos para o kernel. Isto deve-se ao facto de a *source* do kernel não estar presente no ARM, pois é demasiado grande.

De modo a poder compilar código para o ARM noutra plataforma é necessário uma versão do *gcc* preparada para tal. Esta pode ser obtida no site *ftp* da TS - ftp://ftp.embeddedarm.com/ts-arm-sbc/ts-7500-linux/cross-toolchains/. O pacote não traz apenas uma versão diferente do *gcc*, traz também uma versão diferente da *libc*, denominada *uclibc* (micro libc) e ainda um conjunto de ferramentas associadas ao processo de compilação. A *uclibc* destingue-se da *libc* por ser muito mais pequena e estar desenhada para correr em sistemas embebidos. É perfeitamente normal que funções presentes na *libc* não se encontrem na *uclibc*. Por esta razão é extremamente recomendável usar a *cross-toolchains* mais recente possível, de modo a trazer a versão mais actual da *uclibc*.

Uma vez feito o download basta desarquivar o ficheiro para um local onde este fique permanentemente. Por exemplo:

```
tar xvzf crosstool-linux-gcc.tar.gz -C ~
```

Os binários para compilar programas encontram-se em:

```
~/arm-unknown-linux-gnu/bin/
```

Embora seja possível usar o compilador especificando o caminho todo, é bastante mais produtivo adicioná-lo à *PATH*, de modo a poder ser chamado de qualquer lado. Para tornar o processo automático é também recomendável inserir o próximo comando no ficheiro `~/.bashrc`.

```
export PATH=$PATH:/home/fred/arm-unknown-linux-gnu/bin/
```

Se tudo correr bem, o seguinte comando deve produzir a versão do *gcc* para o ARM.

```
arm-unknown-linux-gnu-gcc -version
```

Estes são todos os passos necessários para preparar o sistema para compilar programas para o ARM. Para usar o compilador basta fazê-lo da mesma forma que usaríamos o *gcc*. Como por exemplo:

```
arm-unknown-linux-gnu-gcc -o exec main.c
```

Para compilar o kernel também é necessário usar este novo compilador, mas neste caso é necessário alterar a Makefile do kernel e substituir a variável *CROSS_COMPILE* por *CROSS_COMPILE=arm-unknown-linux-gnu-* (Note-se que não é suposto ter *gcc* no final).

## 1.2 Código do Projecto

Esta secção explica como obter o código do projecto. Todo o código desenvolvido para o projecto encontra-se no *github*, precisamente em https://github.com/cnm/mia_vita.

Uma vez feito o *clone* do repositório é possível observar as seguintes directorias principais:

```
kernel_sender/
interruption/
rt2501/
rt3070/
fred_framework/
```

Estas directorias são as que contém a maior parte do código desenvolvido e/ou modificado e é nelas que o resto do manual se foca.

De notar que embora o comando `git branch -a` mostre vários *branches* remotos, o código principal encontra-se no *master*.

# Capítulo 2

# Código

Esta secção explica todo o código desenvolvido. Mais precisamente, descreve quais as funções de cada componente, como compilá-los e correr.

## 2.1 Kernel Sender e Programa Servidor

A directoria `kernel_sender/` contém o código que prepara os pacotes e envia-os para o nó *sink*. Este por sua vez deverá ter uma instância do programa que recebe os dados a correr. Este programa encontra-se em `kernel_sender/user`.

#### Kernel Sender

O programa que envia os dados (*sender_kthread.ko*) opera em *kernel land*. A principal razão para este facto prende-se com a forma como as amostras são recolhidas do ADC. Estas são recolhidas por um módulo de kernel (*int_mod.ko* localizado em `interruption/`) criado pelo João Trindade que preenche um *buffer* com tais amostras e as temporiza. Seria possível ler este *buffer* a partir de um programa em *user land*, contudo isto traria um *overhead* indesejável, pelo que optou-se por fazer o programa cliente também em *kernel land*.

Como seria de esperar, o módulo que envia os dados depende do módulo que recolhe as amostras. Como tal, este último tem de ser compilado primeiro. Na directoria `kernel_sender/` encontra-se um *script* com o nome *compile.bash*. Este *script* trata de compilar tanto o módulo *int_mod.ko* como o módulo *sender_kthread.ko*, resolvendo todas as dependências.

Quando o módulo *sender_kthread.ko* foi desenvolvido, o protocol de sincronização estava em fase de testes. O objectivo dos testes era comparar o atraso dado por dispositivos GPS, face ao atraso medido pelo protocolo. Por esta razão, várias zonas do código encontram-se circunscritas por `#ifdef __GPS__ ... #endif`. Para que o código seja compilado para os testes com GPS basta passar ao compilador a *flag* `-D__GPS__`. Como isto trata-se de um módulo de kernel, é preciso abrir a Makefile e adicionar a linha:

```
EXTRA_CFLAGS+=-D__GPS__
```

Note-se que ao compilar este módulo para correr os testes com GPS, é necessário também compilar o código do driver (Secção 2.2) e o programa que recebe os dados com a mesma *flag*.

É extremamente importante perceber que esta *flag* não faz com que o valor de criação do pacote passe a ser medido pelo GPS. Apenas prepara o código para os testes com GPS. O valor de criação do pacote continua a ser medido pelo protocolo de sincronização. Para alterar o código de modo a usar o valor do GPS como tempo de criação do pacote é necessário alterar as zonas marcadas com comentários no código. (TODO Fred Marca com comentários e faz listagem dos ficheiros sff)

É ainda possível compilar o módulo com informação de *debug* que é emitida para `/var/log/syslog`. Neste caso é necessária a *flag* `-DDBG`.

O módulo *sender_kthread.ko* tem uma funcionalidade muito simples. Limita-se a criar uma *thread*, que por sua vez abre um socket e de X em X tempo vai lendo N amostras do *buffer* de amostras em *int_mod.ko* (ficheiro source: proc_entry.c). Por cada amostra é criado um pacote e os dados são enviados para o *sink*.

É preciso especificar pelo menos o IP do *sink* e o do próprio nó. Isto pode ser feito em *runtime*. Para saber todos os parâmetros de um módulo basta usar o comando:

```
modinfo sender_kthread.ko
```

```
filename:  ./sender_kthread.ko
description:  This module spawns a thread which reads the buffer exported by João and
sends samples accross the network.
author:  Frederico Gonçalves, [frederico.lopes.goncalves@gmail.com]
license:  GPL v2
depends:  int_mod
vermagic:  2.6.24.4 mod_unload ARMv4
parm:  bind_ip:This is the ip which the kernel thread will bind to.  Default is
localhost.  (charp)
parm:  sink_ip:This is the sink ip.  Default is localhost.  (charp)
parm:  sport:This is the UDP port which the sender thread will bind to.  Default is
57843.  (ushort)
parm:  sink_port:This is the sink UDP port.  Default is 57843.  (ushort)
parm:  node_id:This is the identifier of the node running this thread.  Defaults to 0.
(ushort)
parm:  read_t:The sleep time for reading the buffer.  (uint)
```

Especificar parametros para um módulo é bastante simples. Como exemplo:

```
insmod sender_kthread.ko bind_ip="192.168.2.123" sink_ip="192.168.2.1"
```

Por fim é preciso ter em conta a especificação do pacote de dados (Secção 3). Todos os campos são enviados em *network byte order*, que é *Big Endian*. Os processadores ARM podem funcionar tanto em *Little Endian*, como em *Big Endian*. Infelizmente, os processadores das placas TS-7500 funcionam em *Little Endian*, pelo que os dados têm de ser convertidos antes de serem enviados. Para tipos de dados alinhados, isto é, inteiros de 16, 32 e 64 bits o kernel já fornece funções que fazem a conversão. Contudo, cada amostra tem 24 bits, pelo que não existe nenhuma função que faça a conversão por nós. Esta foi então implementada na função *read_nsamples* localizada no ficheiro *interruption/proc_entry.c*. O modo como foi implementada foi pensado para ser o mais rápido possível, evitando ciclos. Contudo é preciso ter especial cuidado com o seguinte. Esta conversão depende de dois grande factores:

1. Assume que o ARM trata os dados como *Little Endian*. Se por alguma razão o *hardware* mudar, é preciso verificar se esta conversão está a ser feita correctamente.
2. Assume que o código no ficheiro *interruption/fpga.c* lê as amostras de uma forma especifica. Se este código mudar, é necessário verificar se a conversão contínua a ser bem feita. Por outras palavras, o código da função *read_nsamples* é extremamente dependente do código do ficheiro *interruption/fpga.c*.

### Programa Servidor

Na directoria `kernel_sender/user` encontra-se o programa que recebe os dados do módulo `sender_kthread.ko`. Para compilar o programa basta usar a Makefile dentro da directoria. A Makefile define a variável `CC`, que é usada para determinar qual o compilador a usar. Para compilar o programa para o ARM, basta alterar esta variável para reflectir o caminho para o *cross-compiler*. Por exemplo:

```
make CC=arm-unknown-linux-gnu-gcc
```

Tal como os módulos do kernel, este código também pode ser compilado para ser usado nos testes com GPS. Neste caso é preciso adicionar à *flag* `CFLAGS` a opção `-D__GPS__`, dentro da Makefile. O *output* do programa são dois ficheiros, um em formato binário e outro em formato json (ver Secção 3). É preciso ter em conta que quando são feitos os testes com GPS, o campo *timestamp* reflecte o atraso medido pelo protocolo

de sincronização e um campo adicional (*gps_us*) reflecte o atraso medido pelo GPS. O primeiro encontra-se em nanosegundos e o último em microsegundos.

Os pacotes são escritos para o ficheiro binário tal e qual como chegam ao socket. Contudo, antes de serem escritos para o formato json, são convertidos para a *byte order* do CPU. Tanto o ficheiro binário, como o ficheiro json são rotativos. Os pacotes são escritos para um ficheiro A. Na próxima escrita, o ficheiro A é movido para um ficheiro B e rescrito com os novos pacotes. Todos os nomes e tamanhos do ficheiro são configuráveis em *runtime*:

./main -h

Usage: ./main [-i <interface>] [-p <listen_on_port>] [-b <output_binary_file>] [-j <output_json_file>] [-o <moved_file_prefix>]
-i Interface name on which the program will listen. Default is eth0
-p UDP port on which the program will listen. Default is 57843
-b Name of the binary file to where the data is going to be written. Default is miavita.bin
-j Name of the json file to where the data is going to be written. Default is miavita.json
-t Test the program against GPS time. Make sure to compile this program with -D__GPS__.
-o Output file prefix when the file is moved by log rotation. Default is miavita_old.
-c Buffer capacity expressed in terms of number of packets. Default is 100.

Na altura da escrita deste manual, a interface gráfica para o utilizador necessitava de ler os ficheiros json com os pacotes ordenados por *timestamp*. Como tal, o programa que recebe os dados faz *buffering* de X pacotes e antes de os escrever ordena-os. O mecanismo de *buffering* e rotação dos ficheiros encontra-se implementado no ficheiro `list.c`. Embora todos os X pacotes sejam ordenados, apenas $\frac{X}{2}$ são escritos para os ficheiros. Isto serve para evitar ao máximo que um pacote atrasado fique desordenado nos ficheiros.

## 2.2 RT2501 - Driver Modificado

O driver fornecido pela Ralink foi modificado para sincronizar os pacotes de acordo com o algoritmo descrito na minha tese. A principal diferença é que o código actual faz tudo ao nível do driver e não necessita da *framework* de interceptores.

A ideia é conseguir interceptar os pacotes enviados no último momento possível. Neste caso, trata-se da função que submete o pacote ao controlador de USB (função `RTUSBBulkOutDataPacket` no ficheiro `rtusb_bulk.c`). Por outro lado, também é necessário interceptar os pacotes recebidos o mais cedo possível. Neste caso, trata-se da *callback* chamada pelo controlador de USB (função `REPORT_ETHERNET_FRAME_TO_LLC` no ficheiro `rtusb_data.c`).

As funções que sincronizam os pacotes enviados e recebidos encontram-se no ficheiro `sync_proto.c`. É preciso notar que as funções que submetem pacotes ao controlador de USB e recebem pacotes do mesmo, são chamadas para todos os pacotes enviados e recebidos. Por esta razão é preciso impedir que o *driver* tente sincronizar pacotes que nada têm haver com o módulo *sender_kthread.ko*. Como por exemplo pacotes ARP. Para este efeito pensou-se no conceito de filtros, muito semelhante ao modo como funcionam as *iptables*. Neste caso, são definidos um conjunto de filtros que indicam qual o tráfego a sincronizar. Estes filtros encontram-se implementados no ficheiro `filter_chains.c`. Quando nenhum filtro é especificado, nenhum pacote é sincronizado. Contudo, é possível definir filtros para sincronizar, por exemplo, todo o tráfego gerado na porta 57843 pelo IP 192.168.0.123. O driver assume que quando um pacote passa nas especificações de um filtro tem o formato especificado no capitulo 3.

Uma vez mais, se todos os outros componentes foram compilados para os testes com GPS, este também o deverá ser (Adicionar a *flag* `EXTRA_CFLAGS+=-D__GPS__` à Makefile).

Os filtros podem ser definidos em *runtime*. Para tal, o driver implementa uma entrada *proc* (`/proc/synch_filters`) que recebe dados num dado formato e cria os filtros apropriados. Na directoria `rt2501/sources/Module/user` encontram-se dois programas que tratam de criar e remover filtros. É necessário compilar os programas para o ARM, pelo que a Makefile deve reflectir o compilador apropriado. Opcionalmente pode-se ainda correr o comando `make install` para que os comandos de filtros fiquem instalados no sistema. De modo a saber como usar os comandos, basta corrê-los sem argumentos.

O comando que remove filtros recebe o identificador do filtro a remover. É possível saber este identificador através do comando:

```
cat /proc/synch_filters
```

Este comando imprime também todas as especificações dos filtros criados. É preciso ter atenção que todos os nós por onde o pacote passa têm de ter os mesmos filtros criados de modo a que o protocolo funcione.

## 2.3 Kernel Modificado

O kernel fornecido pela TS foi modificado de modo a conter mais duas *system calls*. Uma é usada para fazer *set* ao tempo do GPS e outro é usada para obter este tempo. O ficheiro `README.rst` no repositório do miavita, explica como adicionar *system calls* ao ARM. Este kernel pode ser obtido fazendo:

```
git clone https://github.com/cnm/ts7500_kernel
```

O tempo do GPS é mantido em duas variáveis dentro do kernel (_miavita_elapsed_secs, _miavita_elapsed_usecs). Uma mantém os segundos dados pelo GPS a outra mantém os microsegundos dados pelo ARM. Não é possível obter menos do que um segundo do GPS, pelo que os microsegundos do ARM são usados. Assume-se que dentro de um segundo o *drift* do cristal do ARM não é significativo.

Ambas as variáveis têm 64 bits e são *signed*. Encontram-se dentro da directoria `ipc` dentro das *sources* do kernel, no ficheiro `miavita_syscall.c`. A função `pulse_miavita_xtime` é chamada pelo módulo *int_ mod.c* (Secção 2.1) a cada interrupção dada pelo PPS do GPS. Isto garante que a variável que mantém os segundos tem a mesma precisão que o PPS do GPS, que é cerca de 50 nanosegundos.

O ficheiro `miavita_syscall.c` implementa não só as *system calls* criadas, mas também o modo como outros módulos podem interagir com as variáveis que mantém o tempo do GPS. Para que outros módulos usem tais funções é necessário que incluam o ficheiro `miavita_xtime.h` (`#include <linux/miavita_xtime.h>`). Dentro do repositório do miavita, existe uma directoria denominada `interruption/`, onde está um ficheiro com o nome `fpga.c`. Este ficheiro contém exemplos de como estas variáveis podem ser usadas.

# Capítulo 3

# Especificação do Pacote de Dados

## 3.1 Binário

Esta secção mostra como os dados são mantidos no ficheiro binário. A figura 3.1 mostra a estrutura de cada pacote.

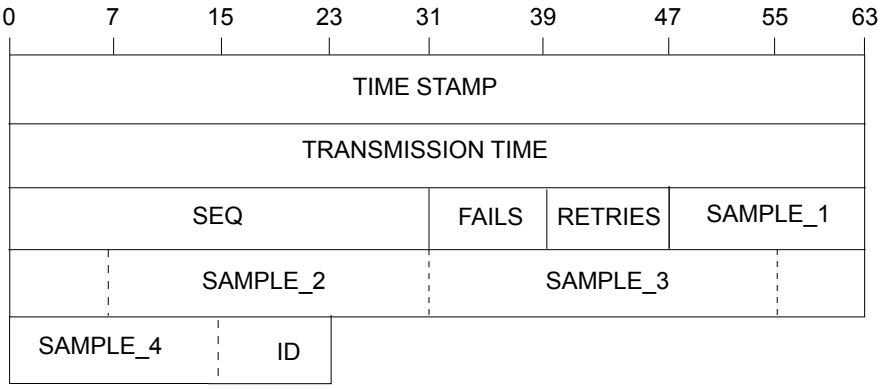


Figura 3.1: Pacote de dados.

Cada campo do pacote é descrito da seguinte forma, quando transmitido na rede:

**Time stamp**
O primeiro campo do pacote é usado para um valor de tempo, com 64 bits, *signed* e transmitido em *Big Endian*. Este campo representa o tempo de criação do pacote, ou o tempo que passou desde que o pacote foi criado.

**Air time estimation**
O segundo campo no pacote é usado para um valor de tempo, com 64 bits, *signed* e transmitido em *Big Endian*. Este campo representa o tempo de transmissão estimado pelo driver.

**Sequence number**
O terceiro campo no pacote é usado para um valor de um número de sequência, com 32 bits, *unsigned* e transmitido em *Big Endian*.

**Fails**
O quarto campo no pacote é usado para fornecer à aplicação o número de falhas que ocorreram desde o último pacote recebido. Este campo pode incluir falhas de todos os pacotes, não só aqueles que pertencem à aplicação.

**Retries**
O quinto campo no pacote é usado para fornecer à aplicação o número de retransmissões que ocorreram

desde o último pacote recebido. Este campo pode incluir retransmissões de todos os pacotes, não só aqueles que pertencem à aplicação.

**Samples**
O sexto campo é usado para guardar 4 amostras retiradas do Geophone, com 24 bits, *signed* e transmitidas em *Big Endian*.

**Node Identifier**
O último campo identifica o nó da rede.

## 3.2 ASCII

Esta secção fornece informação sobre o formato ascii usado para representar os pacotes. O ficheiro de *output* tem um formato JSON, onde as chaves são compostas por `<identificador do nó>:<número de sequência do pacote>`. O seguinte é um exemplo de um ficheiro com o formato descrito.

```
{
        "12:0" : {"timestamp" : 12345, "air_time" : 12344,
                        "sequence" : 0, "fails" : 0,
                        "retries" : 0, "sample_1" : 1234,
                        "sample_2" : 1234, "sample_3" : 12345,
                        "node_id": 12},
        "13:5" : {"timestamp" : 12345, "air_time" : 12344,
                        "sequence" : 5, "fails" : 0,
                        "retries" : 0, "sample_1" : 1234,
                        "sample_2" : 1234, "sample_3" : 12345,
                        "node_id": 13}
}
```

# Capítulo 4

# Como Usar os Componentes

Esta secção demonstra como correr e instalar cada componente de modo a ter um caso de uso do miavita. Por agora o cartão tem de poder ser escrito. Em primeiro lugar é necessário ter o kernel descrito na secção 2.3 preparado para *flashar* na placa ARM. Este é o procedimento que utilizei:

1. No meu computador dentro da pasta que contém o kernel executo `make modules modules_install zImage`.
2. Insiro o cartão do ARM no meu computador.
3. Apago a pasta `/lib/modules/2.6.24.4/` do cartão, executando `rm -r /media/<particao4>/lib/modules/2.6.24.4/`.
4. Copio os novos módulos para dentro do cartão, executando `cp -r /lib/modules/2.6.24.4/ /media/<particao4>/lib/modules/`.
5. Ponho o novo kernel na partição dois do cartão SD, executando `dd if=arch/arm/boot/zImage of=/dev/sdb2` (Atenção que o caminho do *output* pode variar).
6. Como normalmente quero que o kernel na *flash* seja o mesmo do cartão, faço logo uma cópia para dentro do cartão SD de modo a poder *flashá-lo* mais tarde. Executo `dd if=/dev/sdb2 of=/media/<particao4>/root/zImage.dd`
7. Não esquecer de desmontar o cartão - `umount /media/*`
8. Ponho o cartão numa placa ARM e em geral uso uma *board* de desenvolvimento para concluir o procedimento.
9. Ponho a *board* a *bootar* pelo cartão SD.
10. Em primeiro lugar, faço o *flash* do novo kernel:

    ```
    spiflashctl -W 4095 -z 512 -k part1 -i zImage.dd
    ```

11. Após concluído, é preciso criar as dependências dos módulos, executando `depmod`.
12. Por fim, é preciso fazer *reboot* à placa.

Em segundo lugar, é preciso pôr o módulo rt73.ko na placa. Este passo pode ser intercalado com o anterior, mas assim fica mais explícito o que é necessário fazer:

1. Copio o módulo normalmente para a pasta `/lib/modules/2.6.24.4/kernel/drivers/net/wireless/rt2xx0/`.
2. Após concluído, é preciso criar as dependências dos módulos outra vez, executando `depmod`.

3. Por fim, é preciso fazer *reboot* à placa.

Terceiro passo consiste em copiar os programas e módulos para dentro do ARM de modo a poder usá-los.

1. No meu computador, compilo todos os programas e módulos com o *cross-compiler* para o ARM. Isto inclui, programa utilizador que recebe os dados, programas que criam e removem os filtros, o módulo *int_mod.ko* e o módulo *sender_kthread.ko*.

2. Copio todos os componentes para dentro do cartão SD. Em específico, copio os programas `mkfilter` e `rmfilter` para dentro da pasta `/usr/bin/`. De notar que não vale de muito ter o módulo *sender_kthread.ko* e o programa que recebe os dados na mesma placa.

**A partir daqui o sistema está pronto para ser usado. É necessário colocar o cartão em *read-only*.**

Para iniciar um caso de uso do miavita, é necessário matar alguns processos. O seguinte comando deve ser efectuado a cada *boot* da placa:

```
kill $(pgrep xuartctl); kill $(pgrep daqctl); kill $(pgrep dioctl); kill -9 $(pgrep
logsave); kill $(pgrep ts7500ctl); sleep 2; ts7500ctl -autofeed 3; sleep 5; kill $(pgrep
ts7500ctl);
```

De seguida é necessário inserir os módulos para dar início ao programa que recolhe e envia amostras para o sink. De notar que o driver rt2501 modificado já foi inserido automaticamente quando se inseriu a pen wifi.

1. Inserir o módulo que recolhe amostras: `insmod int_mod.ko`

2. Inserir o módulo que envia as amostras: `insmod sender_kthread.ko bind-ip="<IP da placa wifi>" sink-ip="<ip da placa wifi do sink>"`

Por fim, basta colocar no sink o programa que recebe os dados a correr:

```
./main -i rausbwifi
```

A partir deste ponto as placas estarão a enviar dados para o sink e este a recebê-los. Caso seja necessário testar o protocolo com GPS é preciso compilar os componentes com a *flag* `-D__GPS__` e antes de inserir qualquer módulo é preciso inicializar as variáveis no kernel (Secção 2.3). Para tal é preciso usar o programa `init_counter` dentro da directoria `kernel_sender/user/`. Compila-se com:

```
make CC=arm-unknown-linux-gnu-gcc init_counter
```

A seguir é preciso correr o programa antes de inserir qualquer módulo. É preciso ter em conta que o programa inicializa o programa `xuartctl`, que tem de ser terminado após o programa `init_counter` terminar. É ainda preciso ter atenção que o programa tenta abrir o *device* do GPS mais comum que as placas criam (`/dev/pts/1`). É possível especificar outro *device* com a *flag* `-d`. Contudo, o nome do *device* só é conhecido após o programa `xuartctl` ter inicializado. Por esta razão, se o *device* for outro é preciso matar o `xuartctl` antes de correr outra vez o programa `init_counter`.

# Capítulo 5

# Framework de Interceptores

Dentro da directoria `fred_framework/` encontra-se o código correspondenete à framework de interceptores. A ideia de tal framework é criar uma estrutura base onde outros módulos do kernel (Interceptores) se possam registar e interceptar o tráfego em cada nó. Por exemplo, a agregação de pacotes realizado no projecto miavita é feita através de dois interceptores. Um é responsável por interceptar o tráfego e agregá-lo, enquanto que outro é responsável por desagregá-lo.

A framework trabalha por cima da API de Netfilters do kernel. Assim que o módulo da framework é inserido no kernel, são registados 5 hooks (Um para cada ponto de intercepção fornecidos pelo Netfilters - Post routing, local in, etc.). Cada interceptor regista-se na framework para poder interceptar o tráfego desejado. Para que cada interceptor só esteja ciente do tráfego que realmente precisa de interceptar, a framework fornece um mecanismo que permite filtrar tráfego não desejado. Isto é atingido através de umas estruturas denominadas **regras**. As regras são simplesmente um conjunto de **filtros**. Os filtros são estruturas que possuem um conjunto de especificações e um ponto de interacção. Por exemplo, uma regra para agregar tráfego ao nível da rede e aplicação, com destino à porta 57843 e ip 192.168.0.1 iria criar uma regra com dois filtros. Um filtro iria actuar no ponto de intercepção *local out* e o outro no ponto de intercepção *post routing*. Ambos os filtros iriam conter uma especificação que iria informar a framework que todo o tráfego com aquele destino deve ser interceptado pelo interceptor de agregação.

Em suma, a framework de interceptores permite o registo de um ou mais interceptores que definem regras para que possam interceptar um subconjunto dos pacotes que passam no nó.

Ao nível da implementação, as regras são criadas através de um comando user level que escreve para um ficheiro proc. Este comando denomina-se *mkrule*. De forma semelhante existe um comando para eleminar regras - *rmrule*. No ficheiro `README.rst` na pasta `fred_framework` está descrito como usar estes comandos. Existe ainda a explicação de como usar o comando `emsg` para determinar a mensagem de erro associada a um código de erro.

## 5.1 Interceptor de agregação

Como o nome indica, este interceptor é responsável por agregar o tráfego que sai de um dado nó. No ficheiro `README.rst` na pasta `fred_framework` descreve como usar este interceptor. Este documento explica como o protocolo funciona.

### 5.1.1 Nível Aplicação

Na realidade, este tipo de agregação deveria chamar-se agregação ao nível transporte, pois os pacotes são agregados de acordo com informação no header UDP. Contudo, por razões "históricas" continou-se a chamar agregação ao nível da aplicação.

Quando um pacote a ser agregado é interceptado pelo protocolo de agregação este procederá da seguinte forma:

1. Se não existir um buffer associado ao destino desse, o protocolo cria um.

2. O pacote é inserido no buffer.

3. Se o buffer ficar cheio, um pacote agregado é construído e enviado para o destino.

Neste tipo de agregação os pacotes são agregados da seguinte forma:
`|IP header | UDP N | UDP N - 1| ... | UDP 0 |`
Isto significa que os pacotes são enviados por ordem inversa. O header IP é criado pelo protocolo de agregação na altura do seu envio. Este header leva informação no seu campo de protocolo a informar que este é um pacote agregado ao nível da aplicação.
É preciso ter em atenção que o protocolo cria uma thread que de X em X tempos irá fazer *flush* a todos os buffers de agregação com pacotes pendentes.

### 5.1.2 Nível Rede

Este tipo de agregação segue o mesmo processo do que aquele descrito na agregação ao nível da aplicação. Contudo, o pacote agregado passa a ter este aspecto:
`|IP header | IP N | IP N - 1| ... | IP 0 |`
Isto significa que os pacotes agregados são agora pacotes IP completos e podem portanto vir de origens diferentes. A mesma thread que faz *flush* aos pacotes agregados ao nível aplicacional, também o faz aos pacotes agregados ao nível da rede. O primeiro header IP é criado pelo protocolo de agregação, mas deste vez o campo do protocolo indica que este pacote é agregado ao nível da rede.

## 5.2 Interceptor de desagregação

O interceptor de desagregação limita-se a desagregar os pacotes, simplesmente olhando para o tipo de protocolo que vem no header IP e procedendo à desagregação ao nível rede ou aplicacional.
Contudo, devido ao facto de os pacotes virem por ordem inversa é necessário um pequeno passo, para que estes sejam entregues pela ordem certa. Para isto, existe um buffer global com tamanho suficiente para conter o máximo de pacotes possível dentro de uma MTU, onde os pacotes são temporariamente postos. Este buffer denomina-se *scatter_ buffer*.
A ideia é a seguinte.

1. O pacote agregado é percorrido apenas uma vez.

2. Apontadores para cada pacote são guardados dentro deste buffer, do fim para o princípio.

3. No final, o buffer é percorrido e os pacotes são entregues pela ordem correcta.