



**Universidade Nova de Lisboa**  
Faculdade de Ciências e Tecnologia  
*Departamento de Informática*

Dissertação de Mestrado

*Mestrado em Engenharia Informática*

# **Captura de tráfego de rede de um processo com base no PCAP**

Nuno Miguel Galvão Martins (29603)

Lisboa  
(2011)





**Universidade Nova de Lisboa**  
Faculdade de Ciências e Tecnologia  
*Departamento de Informática*

Dissertação de Mestrado

# **Captura de tráfego de rede de um processo com base no PCAP**

Nuno Miguel Galvão Martins (29603)

Orientador: Prof. Doutor Vítor Manuel Alves Duarte

*Trabalho apresentado no âmbito do Mestrado em  
Engenharia Informática, como requisito parcial  
para obtenção do grau de Mestre em Engenharia  
Informática.*

Lisboa  
(2011)



# Resumo

---

A monitorização de aplicações permite analisar e compreender o seu comportamento, sendo possível monitorizar durante execuções reais os seus recursos computacionais, nomeadamente a utilização do *cpu*, da memória, dos dispositivos de *IO* (incluindo os dispositivos de rede), etc.

As ferramentas de monitorização de rede, em geral, utilizam a biblioteca *PCap*, de modo a capturar os fluxos das interfaces de rede. Como esta biblioteca é genérica, permite abstrair o modo como o sistema de operação lida com a captura dos fluxos, sendo que no *Linux*, o suporte é garantido pelo sistema de captura e filtragem de pacotes de rede *Linux Socket Filter*.

De um modo geral, o volume de tráfego obtido pela monitorização é elevado, sendo necessário filtrá-lo, de forma a que apenas os fluxos de dados relevantes sejam transmitidos para o monitor, em nível utilizador. Quer na biblioteca *PCap* quer no suporte dos sistemas não existe suporte para a captura baseada nas interações dos processos. Esta funcionalidade é bastante útil para os utilizadores e com vantagens na sobrecarga do sistema e desempenho.

A solução proposta pretende resolver esta problemática, através da Monitorização de Rede Orientada ao Processo (*MRoP*), estendendo o sistema de monitorização de rede do *Linux*, por meio da introdução de um módulo no núcleo, permitindo capturar as interações das aplicações e filtrando os fluxos de rede da aplicação alvo. Esta solução foi avaliada funcionalmente, verificando-se que apenas os fluxos de dados pretendidos existiam na captura. Para além desta avaliação, foi igualmente realizada uma outra de desempenho, que dependeu do conjunto de pacotes capturados, relativos ao processo alvo.

**Palavras-chave:** Monitorização, rede, aplicação, biblioteca PCap, captura de pacotes, instrumentação do núcleo de sistema, filtro

---

# Abstract

---

The monitorization of the applications allow us to understand and analyse their behaviour, during real executions, their computation resources, namely the cpu, memory, IO devices (which includes the network devices), etc.

Network monitoring tools, in general, use the PCap library, in order to capture the flows of network interfaces. Since this is a generic library, it allows to abstract the way the operating system deals with captured and filtering network packets flows. In Linux, this support is guaranteed by Linux Socket Filter. Overall, the traffic's amount obtained by monitoring is high, being necessary to filter it, so that only relevant data streams can be transmitted to the monitor, at user level. Either in libpcap, or on systems' support, there is no capture support based on processes interactions. This is a very useful and advantageous feature for users, such as in the system overhead and performance.

The presented solution aims to solve this problematic through *Monitorização de Rede orientado ao Processo* (Network Monitoring oriented Process), extending the Linux network monitoring system, through the inclusion of a kernel module, allowing to capture the applications' interactions and filtering network flows of the target application. This solution was functionally assessed, checking that only the desired data streams could be captured. Besides this evaluation, performance was also assessed, which depended on the number of captured packets, regarding the target process.

**Keywords:** Monitoring, network, application, libpcap, packet capture, kernel instrumentation, filter, filtering

---





# Conteúdo

|          |   |          |
|----------|---|----------|
| <b>1</b> | <b>Introdução</b>   | <b>1</b> |
| 1.1      | Contexto . . . . .  | 1        |
| 1.2      | Principais contribuições . . . . .                                  | 2        |
| 1.3      | Organização do Documento . . . . .                                  | 3        |
| <b>2</b> | <b>Monitorização de processos</b>                                   | <b>5</b> |
| 2.1      | Obtenção de informação . . . . .                                    | 6        |
| 2.2      | Monitorização de Rede . . . . .                                     | 8        |
| 2.3      | Biblioteca <i>PCap</i> e <i>Linux Socket Filtering</i> . . . . .    | 9        |
| 2.3.1    | Arquitetura . . . . .   | 10       |
| 2.3.2    | Biblioteca . . . . .  | 11       |
| 2.3.3    | Linux Socket Filtering . . . . .                                    | 12       |
| 2.3.4    | Limitações e optimizações . . . . .                                 | 13       |
| 2.4      | Optimizações . . . . .  | 15       |
| 2.5      | Sistemas de monitorização no núcleo do <i>Linux</i> . . . . .       | 16       |
| 2.5.1    | Eventos pré-definidos . . . . .                                     | 17       |
| 2.5.1.1  | Linux Trace ToolKit . . . . .                                       | 17       |
| 2.5.1.2  | OProfile . . . . .  | 18       |
| 2.5.2    | Suporte à monitorização . . . . .                                   | 19       |
| 2.5.2.1  | KProbes . . . . .   | 20       |
| 2.5.2.2  | Linux Kernel State Tracer . . . . .                                 | 23       |
| 2.5.3    | Comparação entre os diferentes sistemas de instrumentação . . . . . | 23       |
| 2.6      | Transferência de dados . . . . .                                    | 24       |
| 2.6.1    | Técnicas de transferência de dados . . . . .                        | 24       |
| 2.6.2    | Interface do sistema com os processos . . . . .                     | 25       |
| 2.6.3    | Comparação entre sistemas de transferência de dados . . . . .       | 26       |

|          |  |           |
|----------|--|-----------|
| 2.7      | Captura de tráfego de um processo específico . . . . .       | 27        |
| 2.8      | Conclusão . . . . .  | 29        |
| <b>3</b> | <b>Sistema Proposto</b>                                      | <b>31</b> |
| 3.1      | Estrutura de um processo . . . . .                           | 31        |
| 3.2      | Arquitetura de rede em <i>Linux</i> . . . . .                | 33        |
| 3.2.1    | Sockets e as suas famílias . . . . .                         | 33        |
| 3.2.1.1  | AF_UNIX . . . . .  | 34        |
| 3.2.1.2  | AF_NETLINK . . . . .   | 34        |
| 3.2.1.3  | AF_INET . . . . .  | 35        |
| 3.2.1.4  | AF_PACKET . . . . .  | 36        |
| 3.2.2    | NetFilter . . . . .  | 37        |
| 3.2.3    | Traffic Control . . . . .                                    | 37        |
| 3.2.4    | Interfaces de rede . . . . .                                 | 38        |
| 3.2.4.1  | Estrutura de um <i>sk_buff</i> . . . . .                     | 38        |
| 3.2.5    | Captura dos fluxos de dados das interfaces de rede . . . . . | 38        |
| 3.3      | Arquitetura do MRoP . . . . .                                | 39        |
| 3.3.1    | Instrumentação das chamadas ao sistema de rede . . . . .     | 40        |
| 3.3.2    | Estado do processo . . . . .                                 | 41        |
| 3.3.3    | Filtro de pacotes . . . . .                                  | 41        |
| 3.3.4    | Controlo e Informação . . . . .                              | 42        |
| 3.4      | Conclusão . . . . .  | 42        |
| <b>4</b> | <b>Implementação do sistema proposto</b>                     | <b>43</b> |
| 4.1      | MRoP e a sua implementação . . . . .                         | 43        |
| 4.2      | Instrumentação de funções do núcleo . . . . .                | 45        |
| 4.2.1    | Filtro de processos . . . . .                                | 46        |
| 4.3      | Estado dos <i>sockets</i> do processo . . . . .              | 46        |
| 4.3.1    | Estrutura utilizada . . . . .                                | 49        |
| 4.3.2    | API de comunicação interna do MRoP . . . . .                 | 50        |
| 4.4      | Filtro de pacotes, extensão ao LSF . . . . .                 | 51        |
| 4.5      | Informação de análise e controlo . . . . .                   | 52        |
| 4.5.1    | Informação de controlo . . . . .                             | 52        |
| 4.5.2    | Informação de análise . . . . .                              | 53        |
| <b>5</b> | <b>Avaliação</b>   | <b>55</b> |
| 5.1      | Avaliação Funcional . . . . .                                | 56        |
| 5.1.1    | Teste ao componente de controlo . . . . .                    | 56        |

|          |  |           |
|----------|--|-----------|
| 5.1.2    | Obtenção do estado dos canais do processo alvo . . . . . | 56        |
| 5.1.3    | Avaliação de monitorização de rede . . . . .             | 57        |
| 5.2      | Avaliação do desempenho . . . . .                        | 58        |
| 5.2.1    | Desempenho do <i>MRO</i> P . . . . .                     | 59        |
| 5.2.2    | Desempenho da estrutura de dados . . . . .               | 62        |
| 5.2.3    | Desempenho do Sistema de instrumentação . . . . .        | 63        |
| 5.3      | Conclusão . . . . .                                      | 64        |
| <b>6</b> | <b>Conclusões e Trabalho Futuro</b>                      | <b>65</b> |
| 6.1      | Conclusões . . . . .                                     | 66        |
| 6.2      | Trabalho Futuro . . . . .                                | 67        |



# Lista de Figuras

|     |  |    |
|-----|--|----|
| 2.1 | Arquitectura da biblioteca <i>PCap</i> e do <i>LSF</i> . . . . .           | 10 |
| 2.2 | Arquitectura do <i>Linux Trace Toolkit</i> [YD00] . . . . .                | 18 |
| 2.3 | Arquitectura do <i>OProfile</i> [Wil] . . . . .                            | 19 |
| 2.4 | Arquitectura do <i>Kprobes</i> . . . . .                                   | 21 |
| 2.5 | Execução de um <i>KProbe</i> . . . . .                                     | 21 |
| 2.6 | Arquitectura da monitorização de tráfego . . . . .                         | 27 |
| 3.1 | Arquitectura parcial de sockets do Linux . . . . .                         | 34 |
| 3.2 | Protocolo TCP . . . . .  | 35 |
| 3.3 | Protocolo UDP . . . . .  | 35 |
| 3.4 | Arquitectura do <i>MROP</i> . . . . .                                      | 40 |
| 4.1 | Arquitectura geral do <i>MROP</i> . . . . .                                | 44 |
| 4.2 | Elemento da árvore . . . . .   | 49 |
| 4.3 | Lista de endereços . . . . .   | 49 |
| 4.4 | exemplo do repositório <i>Estado do Processo</i> , com 4 sockets . . . . . | 50 |
| 4.5 | API interna do <i>MROP</i> . . . . .                                       | 51 |
| 4.6 | Execução da nova filtragem de pacotes pelo <i>LSF</i> . . . . .            | 52 |
| 5.1 | Testes de desempenho efectuados ao <i>MROP</i> . . . . .                   | 60 |
| 5.2 | Sobrecarga nos testes 5 e 6 . . . . .                                      | 61 |



## Lista de Tabelas

|     |   |    |
|-----|---|----|
| 2.1 | Tabela Comparativa dos sistemas de instrumentação . . . . .                                   | 23 |
| 2.2 | Tabela Comparativa de transferência de dados entre processos e núcleo<br>de sistema . . . . . | 26 |
| 5.1 | Tempos médios em segundos (s) . . . . .   | 60 |
| 5.2 | Sobrecarga das transferências (valores em percentagem) . . . . .                              | 61 |
| 5.3 | Custo das operações (tempos em nanosegundos) . . . . .  | 63 |
| 5.4 | Duração das chamadas em segundos . . . . .  | 64 |







# Introdução

## 1.1 Contexto

A monitorização de uma aplicação destina-se, normalmente, à obtenção de informações relevantes acerca do seu comportamento durante a execução, mas pode igualmente servir para verificar a correcção, recursos atribuídos e usados, desempenho de execução, etc.

A maioria dos sistemas de operação generalistas apresentam métodos de monitorização, devido à importância de que estes se revestem, quer no desenvolvimento das aplicações, quer na gestão destes sistemas. Se algumas ferramentas são específicas na monitorização de determinados recursos (como a biblioteca *PCap*, que é específica nas interacções com o exterior utilizando interfaces de rede), outras são mais generalistas podendo monitorizar recursos diversos (como o *LTT*, *OProfile*, etc.).

Esta dissertação foca-se na monitorização do núcleo, nas interacções das aplicações através de interfaces de rede.

O dinamismo das aplicações pode dificultar bastante o processo de monitorização. Esta situação é particularmente sentida ao nível da monitorização dos dispositivos de *IO* e interfaces de rede. Nas aplicações onde as ligações e interacções são bastante dinâmicas e efectuadas em cada execução de um modo nem sempre previsível, a monitorização é particularmente difícil de ser realizada.

A monitorização no núcleo pode gerar um elevado volume de dados que podem

mostrar-se irrelevantes para as análises efectuadas. Se considerarmos que a transferência dos dados gerados pela monitorização, do núcleo para o nível utilizador, onde se localizam as ferramentas que procedem à análise dos mesmos, é necessário efectuar a cópia destes dados e proceder à sua filtragem, de modo a obtermos apenas os eventos pretendidos. Se analisarmos atentamente este processo de monitorização, concluiremos que este irá produzir uma sobrecarga sobre o sistema. Assim, de forma a capturar apenas dados relevantes e simultaneamente minimizar os efeitos da monitorização, são aplicados filtros logo que possível, no núcleo do sistema. Considerando que aplicações *P2P* utilizam diversos portos de comunicação, revela-se difícil capturar os pacotes com base nos actuais filtros, sem que se assista a uma elevada degradação do desempenho. Não sendo exclusiva das aplicações *P2P*, a utilização de um elevado número de portos, também se verifica em sistemas *Voice over IP*. Estas aplicações, nas suas diversas comunicações, não utilizam sempre portas conhecidas *à priori*, pois por vezes fazem uso de protocolos em que no início da sessão negociam portos, o que dificulta substancialmente a utilização de filtros, por se revelar necessário conhecer todos os protocolos específicos das aplicações. Para além desta dificuldade, existem aplicações que são executadas pelo administrador que não podem ser monitorizadas em nível utilizador, designadamente no caso da utilização do carregamento dinâmico de bibliotecas instrumentadas, para efectuar a monitorização de aplicações.

No contexto de uma dissertação anterior [Far09], utilizou-se a biblioteca *PCap* para a captura do tráfego de rede com vista à monitorização das interacções entre processos distribuídos, sendo que, um dos principais desafios solucionados consistiu no isolamento de pacotes pertencentes a um processo. Porém, a solução encontrada, não se revela de fácil integração em qualquer outra ferramenta e acarreta elevada sobrecarga.

Nos actuais sistemas de monitorização de rede não existe suporte para a monitorização das interacções de rede dos processos. Para levar a cabo esta tarefa de monitorização é imprescindível que métodos alternativos de monitorização de processos sejam combinados com a monitorização genérica de rede. Esta combinação, quando possível, manifesta fraco desempenho e implica uma elevada especificidade na monitorização dos programas.

## 1.2 Principais contribuições

É objectivo desta dissertação investigar mecanismos, incluindo os internos ao núcleo, que permitam incorporar a filtragem com base no identificador do processo, assim como a sua possível integração nas funcionalidades do *PCap*, e proceder igualmente à avaliação da sobrecarga introduzida e respectiva optimização.

A abordagem efectuada beneficia dos mecanismos de instrumentação do núcleo para obter as interacções das aplicações com as interfaces de rede, criando uma extensão ao sistema de filtragem de pacotes do *Linux*, que apenas devolve à monitorização os pacotes referentes à aplicação instrumentada, reduzindo substancialmente o seu número, de modo a transferir apenas os dados relevantes para o monitor, evitando trocas de contexto e cópias de dados desnecessárias.

A possibilidade de monitorizar apenas o fluxo de rede de um determinado programa, poderá permitir que filtros construídos até ao momento possam ser simplificados. Para além desta simplificação, a monitorização do fluxo de rede de uma aplicação, permite a observação dos dados, sem necessitar de um sistema que previamente identifique os protocolos de mais alto nível, aplicados sobre a rede. A existência de um sistema de captura de tráfego de rede genérico, torna-se benéfico para a análise de protocolos, na medida em que nem sempre se tem acesso às especificações destes, presentes nas aplicações. Com este componente, o desempenho na obtenção dos dados relevantes poderá ser incrementado, mitigando anteriores problemas constatados (tais como trocas de contexto, cópias de dados, entre outros), entre o núcleo de sistema de operação e as ferramentas de análise de tráfego. Merece igualmente referência, a possibilidade de análise dos fluxos do processo sem necessidade de instrumentar o código da aplicação, uma vez que a instrumentação é efectuada no núcleo. À possibilidade anteriormente referida acresce a de monitorizar o fluxo de diferentes máquinas virtuais dentro de um sistema, sempre que estas sejam implementadas utilizando processos, permitindo individualizar e capturar o tráfego de cada uma. Esta funcionalidade pode ser particularmente interessante em centros de dados, visto ser possível efectuar a análise ao tráfego, sem necessidade de proceder à paragem da máquina.

Foi submetido e aceite um artigo para o *Inforum - 2011 Simpósio de Informática*<sup>1</sup>, contendo uma descrição sucinta do trabalho realizado nesta dissertação, bem como os resultados obtidos através das validações.[MD11]

## 1.3 Organização do Documento

Os restantes capítulos do documento, encontram-se assim distribuídos e estruturados:

- **Capítulo 2 - Monitorização de processos** - Introdução à monitorização de processos, evidenciando a monitorização de rede. Apresentação do estado da arte da monitorização do núcleo do *Linux* e trabalhos relacionados com a monitorização de processos.

---

<sup>1</sup><http://inforum.org.pt/INForum2011>

- **Capítulo 3 - Sistema Proposto** - Estrutura de comunicação e monitorização de rede do *Linux*, bem como os seus constituintes. Apresentação da estrutura do *MROp* e da sua interligação com a estrutura de rede do *Linux*.
- **Capítulo 4 - Implementação do sistema proposto** - Implementação do *MROp* e discussão da implementação.
- **Capítulo 5 - Avaliação** - Avaliação funcional e de desempenho do *MROp* e dos seus componentes. Análise do desempenho do sistema de monitorização utilizado (*KProbes*).
- **Capítulo 6 - Conclusões e Trabalho Futuro** - Apresentação das conclusões referentes à avaliação efectuada ao *MROp* e propostas para a sua evolução.



## Monitorização de processos

O recurso à monitorização do comportamento das aplicações, permite-nos obter dados relevantes sobre os recursos realmente utilizados, de modo a proporcionar-nos um conhecimento mais profundo do seu comportamento em execuções reais, permite também analisar o plano de execução, os métodos mais executados e detectar situações de eventos interessantes. As informações recolhidas permitem analisar o desempenho ou correcção da aplicação, sequencial ou distribuída, porquanto ao conseguir-se obter dados da utilização do *cpu*, da memória, dos dispositivos de *IO*, interacções, etc, é possível compreender o comportamento dinâmico das aplicações. Daí ser comum a utilização da monitorização como auxiliar na avaliação e depuração de programas, conseguindo um bom compromisso entre a qualidade dos dados recolhidos e a perturbação das aplicações [Dua05].

Na secção ?? ... será apresentada a forma de monitorização de rede no *Linux*. Na secção ?? ... são apresentadas ferramentas e mecanismos do núcleo para efectuar a monitorização deste e dos processos em nível utilizador. Como a obtenção de dados provenientes do núcleo é um dos problemas de desempenho da monitorização, são apresentados na secção ?? ... algumas das interfaces e mecanismos de transferência de dados do núcleo para os processos em nível utilizador.

## 2.1 Obtenção de informação

Os dados relativos ao comportamento das aplicações, obtidos de diferentes fontes, são normalmente coligidos e posteriormente analisados por ferramentas especializadas. Existem diferentes formas de coligir, visualizar e até interactivar com os monitores, cada uma com as suas especificidades e capacidades próprias. Algumas são desenvolvidas para objectivos específicos. Tendo em vista o conhecimento geral das capacidades de cada uma, podemos analisar estes sistemas de variados pontos de vista.

Se considerarmos por exemplo quanto à análise e visualização face ao instante da execução da aplicação alvo, podemos dividir os sistemas em:

**Online** - Enquanto decorre a monitorização da aplicação é possível observar os dados que são recolhidos pelo monitor. Como os eventos estão a ser recolhidos e visualizados em simultâneo, apenas podemos observar a história até ao momento, mas temos uma baixa latência entre os acontecimentos e a sua observação.

**Offline ou *Post-Mortem*** - A história do programa é analisada após este se ter completado, daí a designação *Post-Mortem*. Este método permite-nos analisar integralmente a sua história e correlacioná-la. Permite análises mais completas e computacionalmente mais exigentes.

Se a monitorização necessitar de interactividade do utilizador, é possível defini-la de duas formas:

**Activa** - Por iniciativa explícita do utilizador, é possível inquirir o sistema de monitorização sobre o estado da computação ou mesmo alterá-la. Este método, por vezes descrito como *computational steering*, é a forma com maior interactividade, uma vez que permite ir analisando e modificando os parâmetros da monitorização ou mesma da aplicação.

**Passiva** - Esta forma de monitorização é especialmente utilizada em ambientes onde é relevante a obtenção da totalidade dos dados e, apenas no final, nos debruçarmos sobre a sua análise. Esta forma é designada de passiva, pois o utilizador não tem intervenção na forma como os dados estão a ser obtidos, o que reduz a perturbação no sistema. Quanto muito a sua acção acontece antes da execução, para configuração da informação a recolher.

A própria instrumentação da aplicação pode ser de dois tipos: a estática e a dinâmica, cada uma com as suas características próprias:

**Estática** - Na instrumentação estática o código instrumentado é definido em tempo de compilação ou utilizando bibliotecas próprias para o efeito, como a utilização da função *assert*, que define os pontos a serem monitorizados. Durante a execução não podem ser adicionados ou removidos pontos de análise.

**Dinâmica** - Em contraste com a instrumentação estática está a dinâmica. É mais complexa que a estática e permite a inserção e remoção dos pontos a serem monitorizados. Caracteriza-se pela ausência do ciclo *introduzir ponto* → *compilar programa* → *executar* → *remover ponto* de instrumentação. A utilização de pontos de instrumentação dinâmica, pode ajudar a reduzir o grau de perturbação, uma vez que apenas são definidos os que se desejam observar. Tal pode ser efectuado no início ou durante a sua execução, criando, alterando, destruindo os pontos de observação sobre os recursos monitorizados.

A recolha de dados provenientes da monitorização, é uma das componentes mais sensíveis, relativamente ao grau de perturbação da monitorização. Em geral as informações recolhidas da monitorização são armazenados em memória central sendo posteriormente armazenados em memória persistente. Esta transferência de memória central para disco em geral deve-se a uma acção explícita do utilizador, ou então, a algum evento indicador da necessidade de guardar os dados, de forma a que novos dados possam ser armazenados em memória, uma vez que os *buffers* em memória geralmente têm dimensão fixa.

Existe claro a preocupação de que o sistema a ser monitorizado, tenha um baixo grau de perturbação, pois esta perturbação pode levar à alteração dos resultados obtidos e mesmo a comportamentos erráticos da aplicação (especialmente perante execuções concorrentes). Daí que, diversas abordagens foram criadas, para reduzir o impacto da monitorização num sistema em produção. Uma destas abordagens traduz-se na utilização de instruções especializadas, as quais alguns processadores dispõem para *debug*, de modo a utilizar os recursos que mais se adequem à monitorização. Estes métodos, nem sempre são utilizados, devido à sua dependência da arquitectura, o que dificulta a sua portabilidade. Com vista a minimizar a perturbação, alguns sistemas de monitorização utilizam uma técnica de amostragem, o que permite obter indicações sobre o estado da computação a cada intervalo de tempo. Esta técnica, em oposição à criação de um traço de execução, permite obter dados sobre os recursos apenas por

amostra, limitando à partida a perturbação, enquanto que na criação de um traço de execução, é possível obter a totalidade dos eventos de forma a criar uma história completa, conduzindo a uma grande sobrecarga do sistema perante uma elevada taxa de eventos.

No entanto, obter dados oriundos da monitorização pode revelar-se insuficiente, se não dispusermos de uma ferramenta onde estes possam ser tratados, de modo a obtermos análises mais completas relacionado-os com os detalhes de funcionamento da aplicação monitorizada. No entanto estas análises não serão o foco deste trabalho.

## 2.2 Monitorização de Rede

As ferramentas de monitorização de rede são em geral, baseadas na captura de pacotes de forma passiva. Estas capturam os pacotes que fluem através da interface de rede, para posterior análise ao tráfego, que pode incidir sobre a largura de banda utilizada, principais protocolos, eventuais problemas de segurança, etc.

A monitorização das interacções dos processos com o exterior, pode ser efectuada de diferentes formas, duas destas serão apresentadas. A primeira utiliza bibliotecas instrumentadas que obtêm os pacotes de dados quando estes chegam à aplicação, sendo que a segunda forma recorre a mecanismos genéricos do núcleo, para a monitorização de rede. Relativamente à primeira, é necessário conhecer o código da aplicação e instrumentá-lo, obtendo-se apenas os dados que chegam à aplicação. Este processo é específico a cada aplicação e pode ver o seu uso limitado por questões de segurança do sistema. No que à segunda forma se refere, a monitorização é efectuada de forma genérica, não sendo intrusiva para as aplicações, necessitando apenas de efectuar a monitorização do processo e da rede, separadamente.

**Dinamismo das aplicações -** As aplicações são dinâmicas quanto às interacções via rede (por exemplo criando e destruindo canais de comunicação). A este dinamismo devem-se algumas dificuldades com que nos deparamos, ao monitorizar as interacções dos vários fluxos de execução dos processos.

Relativamente à monitorização das interacções com o exterior, utilizando os mecanismos do sistema de operação, este dinamismo é um factor chave, pois existem dificuldades na forma de identificar os fluxos pertencentes a um processo face aos restantes, irrelevantes na nossa análise.

**Formas de reduzir o volume de dados utilizando filtros -** A utilização de filtros na captura do tráfego que circula na rede, é uma forma eficiente de apenas se obter os



dados relevantes, com vista à satisfação dos nossos objectivos e, são particularmente importantes, quando o volume de dados que circula na rede é extremamente elevado. Tendo em vista a eficiência, estes filtros são implementados no núcleo do sistema de operação, baseando-se em regras simples, que podem ser combinadas para contemplar situações mais complexas.

**Dificuldade de criação e alteração de filtros** - Os filtros actualmente suportados para capturar pacotes no *Linux*, são definidos *a priori*, não existindo forma eficiente de os alterar dinamicamente, uma vez que a captura tem de ser interrompida a fim de ser criado um novo filtro, posteriormente aplicado, procedendo-se em seguida à retoma da captura dos pacotes, de acordo com as novas regras.

**Filtros complexos** - Para aumentar o desempenho da captura de pacotes, os filtros são aplicados o mais cedo possível, ou seja, logo que chegam à interface de rede. Face a esta situação, o tipo de filtros que se podem aplicar têm de ser simples, baseados apenas no conteúdo do pacote. Quando os filtros são demasiado complexos, os módulos do núcleo têm de passar grande parte da informação para as camadas superiores, de forma a puderem ser aplicados filtros mais elaborados, que analisam um nível mais abstracto da informação, presente no *payload* dos pacotes, ou a relacionam com outra informação, para obterem apenas os dados relevantes.

**Técnicas para o aumento de desempenho** Diferentes técnicas têm vindo a ser desenvolvidas para aumentar o desempenho da monitorização das interfaces de rede. Como já referido, o aumento da sobrecarga é muito penalizante, daí que esta deva ser mantida bastante reduzida, o que contribuirá para aumentar o desempenho da captura.

Actualmente a utilização de máquinas equipadas com processadores *multi-core* é uma realidade, deste modo a sua utilização permite ultrapassar algumas dificuldades sentidas na captura de pacotes. Se os processadores *multi-core* atenderem em paralelo as diversas interrupções, originadas pelo envio ou recepção de pacotes, o desempenho da rede é passível de ser aumentado. No entanto, revela-se difícil atingir este paralelismo para todas as interfaces de rede, pois nem todas estão preparadas para beneficiar de arquitecturas com múltiplos *cores*.

## 2.3 Biblioteca *PCap* e *Linux Socket Filtering*

A monitorização de rede existente em muitos dos sistemas tipo *Unix* e mesmo no *Windows* permite capturar os pacotes de rede logo que estes chegam ao controlador de

rede. Assim a biblioteca PCap[Lib] permite às ferramentas de monitorização de rede fazerem uso deste mecanismo. Uma das principais características desta biblioteca, é a sua API de alto nível para a captura de pacotes, que é igual em todas as plataformas.

A monitorização de rede realizada pelas ferramentas que utilizam a biblioteca PCap, efectuem uma monitorização passiva da rede, adequando as suas análises aos pacotes recebidos, de modo *online* ou *offline*. Esta monitorização é não intrusiva para o código das aplicações e dependendo do modo como é efectuada, pode ter algum impacto no desempenho destas ou do sistema na sua globalidade.

Para filtrar os pacotes indesejados a biblioteca PCap utiliza filtros baseados no BPF (*Berkeley Packet Filtering*), normalmente implementados no núcleo de sistema, de modo a que a monitorização se torne mais eficiente e menos intrusiva.

Em seguida é apresentada a arquitectura da biblioteca PCap e o seu suporte no Linux, Linux Socket Filtering.

### 2.3.1 Arquitectura

A arquitectura do *Packet Capture (PCap)* divide-se em duas partes: a biblioteca a nível utilizador, e a implementação do *Linux Socket Filtering* no núcleo de sistema. Como anteriormente referido a biblioteca em nível utilizador é constituída por uma API homogénea, que oferece às ferramentas de monitorização de rede um mecanismo para efectuar as suas análises ao tráfego. As funções auxiliares da biblioteca efectuem chamadas ao sistema de modo a interagirem com o mecanismo de monitorização de rede do Linux, Linux Socket Filtering, presente no núcleo.

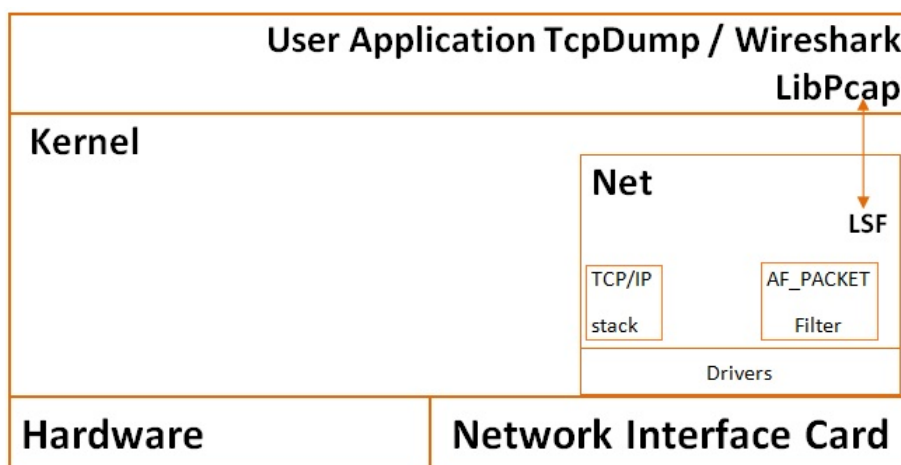


Figura 2.1: Arquitectura da biblioteca PCap e do LSF

### 2.3.2 Biblioteca

A biblioteca em nível utilizador ao ter uma *API* homogénea entre plataformas, permite aos programadores desenvolverem mais rapidamente ferramentas de monitorização de rede multiplataforma, sendo utilizada pela generalidade das ferramentas de monitorização de rede, tais como o *TcpDump*, o *Wireshark*, o *Snort*, etc. Esta biblioteca como é multi-plataforma, permite ao programador utilizá-la nos principais sistemas de operação, dado que a *API* da biblioteca PCap é transparente relativamente à implementação, esta sim, específica a cada plataforma.

No *Linux* para cada utilização da biblioteca PCap é criado um *socket* da família *AF\_PACKET*. Como este tipo de *socket* comunica directamente com a interface de rede permite que sejam activadas e, desactivadas propriedades das mesmas, tais como o modo promiscuo geralmente utilizado para efectuar a captura de pacotes que não pertencem às comunicações da máquina que efectua a monitorização. Apesar desta funcionalidade ser amplamente utilizada para analisar a rede, para o âmbito desta dissertação é totalmente desnecessária, dado que os pacotes a serem recolhidos para análise são os pretendentes a processos da máquina. Para além disso é através deste *socket* que é efectuada a monitorização e captura dos pacotes que atravessam a interface de rede. Um dos parâmetros para a monitorização é a interface sobre a qual irá ser efectuada, ou um modo especial que permite que todas as interfaces possam ser monitorizadas.

A fim de minimizar e capturar apenas os pacotes relevantes para as análises, pode ser necessário filtrar os pacotes não relevantes. Deste modo a biblioteca permite efectuar diversas formas de filtragem, quer em nível utilizador, para filtros mais complexos, quer no núcleo com filtros apenas baseados nos metadados dos pacotes. Se a filtragem for efectuada exclusivamente em nível utilizador, o filtro a ser aplicado indica que todos os pacotes devem ser capturados, senão a ferramenta que utiliza a biblioteca deve especificar um filtro utilizando para isso as instruções pertencentes à biblioteca PCap que as irá traduzir e otimizar para uma outra linguagem (*BPF*)

Caso exista a necessidade de filtrar os dados a capturar é necessário aplicá-lo ao *socket* criado através de *ioctl*, passando em argumentos o canal e o filtro a ser executado a cada pacote. É possível obter estatísticas relacionadas com a captura e filtragem de pacotes, estas estatísticas são também obtidas através de *ioctls* efectuados sobre o canal de recepção da monitorização. Estas funcionalidades estão agregadas à biblioteca, para que o programador não tenha de se preocupar com os mecanismos internos relativos à manipulação da monitorização de rede no núcleo.

Para se efectuar a captura de pacotes, é necessário especificar qual a interface de rede a monitorizar.

Como o número de pacotes que atravessa a interface de rede é elevado, existe a

possibilidade de capturar segundo critérios baseados nos dados dos pacotes. Como verificado anteriormente, quando o número de dados da monitorização é elevado, e nem todos os dados capturados são relevantes para as análises, estes são filtrados, de modo a apenas serem utilizados para análise os dados relevantes. Daí que a biblioteca *PCap* tenha uma linguagem própria para a especificação de filtros, sendo posteriormente traduzidos para a linguagem *BPF* para ser aplicado ao canal, obtendo assim apenas os pacotes relevantes para as análises a efectuar.

Se necessitar filtrar pacotes, especificará o filtro a utilizar, recorrendo às instruções presentes na biblioteca *PCap*, que através da função *pcap\_compile* o traduzirá e optimizará para o conjunto de instruções do *BPF*, sendo posteriormente aplicado no núcleo através da função *setfilter*. Se o filtro for demasiado complexo, não poderá ser aplicado no núcleo, sendo apenas possível aplicá-lo em nível utilizador, perdendo algum desempenho. Antes da introdução do novo filtro no canal, este tem de ser objecto de drenagem, para receber apenas os pacotes do novo filtro. Este modo de filtragem, evita imediatamente a captura de pacotes irrelevantes para a análise e permite que aqueles que não são visíveis às aplicações, devido à *firewall* do *Linux*, possam ser capturados e analisados pelas ferramentas de monitorização de rede.

A arquitectura de rede do núcleo de sistema *Linux*, utiliza *socket buffers*, que contêm dados referentes às diferentes camadas da pilha de rede *TCP/IP*. Esta estrutura que dispõe de toda a informação sobre o pacote, a receber pela aplicação, é igualmente utilizada nos sistemas de monitorização em nível utilizador. A monitorização da rede é efectuada directamente no controlador da interface de rede, utilizando para isso um *socket AF\_PACKET*.

Enquanto que as aplicações apenas recebem os dados (*payload* final), este *socket* recebe a totalidade do pacote (caso número de *bytes* por pacote seja inferior ao tamanho total do pacote este é cortado até ao tamanho indicado no filtro), ou seja os cabeçalhos e os dados, ficando a cargo da ferramenta efectuar as análises necessárias aos pacotes recolhidos.

### 2.3.3 Linux Socket Filtering

O módulo *AF\_PACKET* permite activar o modo promiscuo da interface a monitorizar, ou seja permite que sejam capturados pacotes que não são destinados à máquina utilizada. Na versão actual do núcleo de sistema do *Linux* (2.6.39), o *socket AF\_PACKET* pode utilizar um sistema de partilha de *buffer* (*MMAP*), com o espaço de utilizador. Neste caso, cabe ao utilizador definir explicitamente a forma como pretende utilizar o *socket*. Para atingir este objectivo o utilizador tem de pedir ao núcleo, uma região de memória, sendo esta, posteriormente indicada como argumento de configuração do

*socket*.

Como o referido anteriormente, o *AF\_PACKET*, permite a comunicação directa com o controlador da interface de rede, possibilitando capturar os diversos pacotes ainda antes destes poderem ser filtrados pela *firewall*, presente no núcleo de sistema, neste caso o *netfilter*. Podem ser aplicados filtros aos *sockets* de modo a aumentar a eficiência na captura dos pacotes, excluindo aqueles que são irrelevantes para a monitorização.

O *Linux Socket Filtering* é derivado do *BPF* (*Berkeley Packet Filtering*) que é o *standard de facto* para a criação de filtros para a captura de pacotes de rede, no núcleo de sistema. Este sistema de captura e filtragem permite aos administradores, definir e afectar filtros ao *socket* da família *PACKET*. Os filtros definidos são descritos numa linguagem simples, de forma a efectuarem com rapidez a selecção dos pacotes a capturar, rejeitando todos os outros. A linguagem utilizada para a criação dos filtros é traduzida para o *Instruction Set*, definido no *BPF*. Este *Instruction Set* utiliza operadores lógicos de forma a combinar as regras definidas nos filtros, criando-se apenas um filtro a ser aplicado aos pacotes. [MJ92].

### 2.3.4 Limitações e optimizações

Apesar da API da biblioteca *PCap* ser igual em diferentes plataformas (*Windows*, *Linux*, *FreeBSD*, etc), o desempenho desta pode variar. Quando exista pouco tráfego de rede as diferenças são pouco notórias, mas logo que este se intensifica, estas acentuam-se bastante, como demonstra o estudo *Improving Passive Packet Capture: Beyond Device Polling* [Der04].

De modo geral, para avaliar o desempenho da monitorização de rede, é frequente utilizar-se uma métrica que nos permita determinar a percentagem de pacotes que é possível capturar, sem que se verifiquem perdas. Esta métrica é calculada sabendo o número de pacotes que atravessam uma determinada porta de um *router* ou *switch*, e compará-lo com o número de pacotes capturados na interface. Uma variante desta medida de desempenho é combiná-la com o máximo número de regras que é possível aplicar, sem que se assista à perda de pacotes. É particularmente importante conhecer o número máximo de pacotes capturados ou regras aplicáveis aos filtros, afim de determinar a velocidade máxima expectável de captura. Quanto mais eficiente for a filtragem, menor o número de pacotes transferidos para as aplicações, o que se traduz numa menor sobrecarga para o sistema, aumentando o seu desempenho.

**Captura de pacotes com *TimeStamp*** Um dos pontos que pode influenciar negativamente o desempenho da obtenção de pacotes, é a necessidade de colocar nestes, uma estampilha temporal, com o tempo da sua chegada ao sistema. A biblioteca *PCap* pode

obter a estampilha de duas formas distintas, assim se a interface de rede fornecer a estampilha temporal associada ao pacote, a biblioteca pode utilizar este valor, caso contrário será necessário obter a estampilha temporal da chegada do pacote ao sistema. A forma de obtenção da estampilha pela biblioteca *PCap*, processa-se através da função *gettimeofday*, o que incorre numa maior sobrecarga do sistema face à solução anterior.

**Captura com dns activo** Como já referido a biblioteca *PCap* é utilizada por diversas ferramentas de análise do tráfego de rede, que dela tiram partido. Uma destas ferramentas é o *TcpDump*, este utiliza a biblioteca *PCap* para efectuar

A biblioteca *PCap* *tcpdump* com parametro *n* de dns para obter os nomes de domínios sempre que possível

**Captura especificando tamanho máximo de 65535 para pacotes** MTU de 1500 ethernet

**PACKET\_MMAP** Com base na técnica *MMAP*, foi criado o *PACKET\_MMAP*, disponível a partir da versão 1.0.0 da biblioteca *PCap*. Este módulo permite algumas melhorias em termos de desempenho, visto que foi reduzido o número de cópias efectuadas, face à anterior versão 0.9.8 da biblioteca *PCap*. Esta técnica consiste na utilização de um bloco de memória pertencente ao processo que irá utilizar o *socket*, de modo a que quando seja necessário passar um pacote do núcleo para a ferramenta, não seja necessário efectuar uma cópia da memória onde reside o pacote. Este novo método tem de ser acompanhado por contadores/índices de acesso à memória, dado que, esta zona de memória é acedida de forma circular. Apesar de melhorar significativamente o número de cópias de dados, como os blocos de memória para a introdução de pacotes é efectuada de modo estático, existem apenas um número fixo que podem ser alocados. Assim, se o ritmo de escrita dos pacotes na zona de memória for superior ao ritmo de leitura, podem existir perdas de pacotes dado que não existe espaço suficiente para alocar os novos pacotes enquanto os anteriores não forem consumidos.

---

Para além destas modificações no *PACKET\_MMAP*, o núcleo de sistema de operação *Linux* na sua versão 2.6, passou a contar com a nova *API* de rede (*NAPI*).

Se as interfaces de rede suportarem um mecanismo de mitigação de interrupções, é possível obter melhores resultados, conforme [Der04].

---

**PF\_RING** Este é um novo módulo para o núcleo de sistema, criado com base em duas técnicas *mmap* e *ring\_buffers* anteriormente descritas.

Este módulo difere na abordagem utilizada no *PACKET\_MMAP*, porquanto nesta a memória é mapeada entre a ferramenta e o controlador da interface, enquanto que no *PACKET\_MMAP* a memória é mapeada entre a ferramenta e um *buffer* externo ao controlador da interface, mas interno ao núcleo de sistema. Esta abordagem permite que os dados fiquem disponíveis para a aplicação directamente, verificando-se a inexistência de cópia dos dados do *buffer* do controlador, para o *buffer* partilhado entre a ferramenta e o núcleo[PFR].

**PF\_RING com DNA (Direct NIC Access)** Baseando-se na técnica anteriormente descrita de utilizar um *buffer* partilhado entre a ferramenta de monitorização e o controlador, assiste-se a uma evolução desta técnica, ao permitir que a interface de rede partilhe um *buffer* com a ferramenta de monitorização, possibilitando que os pacotes passem directamente para esta[Int]. Esta partilha é efectuada utilizando *mmap*, *ring\_buffers* e *DMA*. Para se utilizar esta técnica é necessário que a interface de rede, permita a utilização de memória partilhada e *DMA*.

Diversos esforços no sentido de aumentar o desempenho da captura de pacotes têm sido efectuados. Relativamente ao *software*, são conhecidos esforços na utilização da técnica de *mmap*, de forma a reduzir o número de cópias de dados entre as aplicações e o núcleo de sistema. Igualmente no *hardware*, tem-se assistido a uma evolução no sentido de reduzir as interrupções efectuadas ao *cpu*, adicionando nas interfaces de rede processadores dedicados às funcionalidades presentes no núcleo, de modo a libertá-lo da execução destas tarefas.

## 2.4 Optimizações

**NAPI - New API** - Foi criada uma nova *API* (*Application Program Interface*) de atendimento de interrupções do processador, oriundos das interfaces de rede, com o objectivo de aumentar o desempenho da utilização de rede de elevado débito. Esta nova *API*, permite desligar a atenção do processador a novas interrupções, oriundas da interface de rede, durante um certo período de tempo.

Este tempo foi definido, de forma a que não se verifiquem situações de elevada latência na chegada dos pacotes às aplicações. A forma anterior de recepção de pacotes, era efectuada de modo a atender uma interrupção por cada pacote que chegava à interface de rede, gerando assim demasiada sobrecarga no sistema. Actualmente a arquitectura de rede, está balanceada de forma a mudar o modo de atendimento de interrupções para NAPI, caso existam demasiadas interrupções por unidade de

tempo [Adm09].

Utilizando esta nova *API*, é possível explorar um escalonamento mais eficiente das interrupções, em situações de intensa actividade da interface de rede e do processador. O *NAPI* apenas pode ser aplicado nos caso em que os controladores das interfaces de rede estejam preparados para utilizar alguma forma de mitigação da interrupção, caso contrário esta *API* não é aplicada, resultando em interrupções por cada pacote, enviado ou recebido.

A utilização desta nova *API* permite diminuir o número de trocas de contexto entre o controlador da interface e o núcleo de sistema. Sempre que o *cpu* atende uma interrupção de rede, obtém um número maior de dados, que combinado com um sistema de memória partilhada aumenta substancialmente o desempenho.

**Sistemas *multi-core* e *multi-processador*** Com o aparecimento de sistemas *multi-core* e *multi-processador* a que a generalidade do público tem acesso, a paralelização de código ou a forma de tirar partido destas arquitecturas, que permitem um melhor aproveitamento dos recursos, assumem uma particular importância. De forma a tirar partido das arquitecturas *multi-core*, é necessário que o controlador e interfaces de rede, os *buffers*, e os controladores de *DMA* (*Direct Memory Access*), sejam modificados de forma a conhecerem esta estrutura. É pois, determinante um esforço conjunto envolvendo todas estas componentes, com vista à obtenção do máximo rendimento destas arquitecturas[Der10].

## 2.5 Sistemas de monitorização no núcleo do *Linux*

A monitorização do núcleo é uma das melhores formas de conhecer como os recursos da máquina são partilhados entre o núcleo e os processos, assim nesta secção são apresentados ferramentas e mecanismos de monitorização do núcleo. A utilização dos sistemas de monitorização ao nível do núcleo, permitem efectuar a monitorização genérica dos processos, de forma não intrusiva para as aplicações. Ao efectuar a monitorização no núcleo, a sobrecarga imposta ao sistema é menor que em nível utilizador, por ser possível recolher apenas as informações relevantes, evitando filtrar os elementos não relevantes, para análise.

Neste documento constam apenas mecanismos e ferramentas de monitorização dinâmicas, pois apenas estas são as desejadas para a criação de uma componente de monitorização de rede orientada ao processo. De entre as ferramentas de monitorização dinâmica analisadas é possível agrupá-las em duas categorias: eventos pré-definidos



e instrumentação dinâmica. Na categoria instrumentação dinâmica existem dois sistemas mais relevantes: o *KProbes* e o *Linux Kernel Stace Tracer*, enquanto que para os eventos pré-definidos foram verificados o *Linux Trace Toolkit* e o *OProfile*. Estes quatro sistemas são apresentados em seguida considerando as suas categorias:

### 2.5.1 Eventos pré-definidos

Nesta categoria encaixam-se duas ferramentas o *Linux Trace ToolKit* e o *OProfile*. Em cada uma destas ferramentas a monitorização é efectuada sobre pontos previamente definidos, não permitindo a adição de novos pontos de monitorização. Cada um destas ferramentas analisa todo o sistema, sendo a filtragem extra dos eventos efectuada em nível utilizador. Como os eventos detectados através destas ferramentas geram um mapa de execução, a utilização de apenas alguns eventos pré-definidos não é limitativa de uma ferramenta.

#### 2.5.1.1 Linux Trace ToolKit

O *Linux Trace ToolKit* (*LTT*) é uma das ferramentas mais utilizadas efectuar um traços de execução do núcleo *Linux*. Esta ferramenta foi criada no final da década de 1990, sendo posteriormente substituído em 2005 pelo *Linux Trace ToolKit new generation*, (*LTTng*) projecto derivado do *LTT*. O *LTT* é constituído por quatro componentes: o *Kernel Patch*, o *Kernel Module* o *Trace Daemon* e o *Data Decoder* tal como é visível na figura 2.2. É através do *Trace Daemon* que o utilizador pode comunicar com o sistema de monitorização. Os dados recolhidos têm diversos valores tais como o nome da função, o tempo, e o identificador do processador, pelo menos estes atributos de modo a conseguir construir os traço de execução. O *LTT* e o *LTTng* são ferramentas de monitorização dinâmica que efectuam um traço de execução. Estas ferramentas tiram partido dos mecanismos de instrumentação do núcleo de modo a obter dados que permitam efectuar o traço de execução.

Para além do *daemon* em nível utilizador existem outras ferramentas pertencentes ao *LTT* para efectuar análises aos dados recolhidos durante a monitorização.

Os dados provenientes da monitorização são recolhidos através do sistema de ficheiros virtual *RelayFs*.

[YD00] foi substituída pelo *Linux Trace Toolkit New Generation*. O *Linux Trace Toolkit New Generation*, e permite utilizar mecanismos como o *KProbes*, o *Tracepoints*[MR09] e *Linux Kernel Markers*[MR09] para efectuar as suas monitorizações. Os *Tracepoints* e *Linux Kernel Markers* fazem parte da instrumentação estática pertencente ao núcleo de

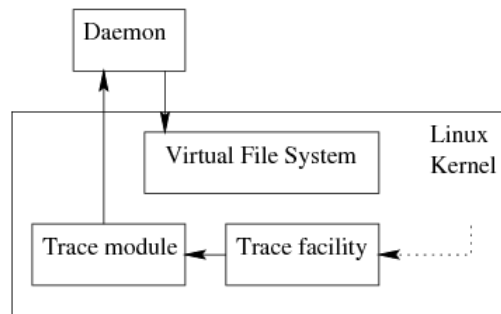


Figura 2.2: Arquitectura do *Linux Trace Toolkit* [YD00]

sistema do *Linux*. A transferência dos dados do núcleo de sistema para a aplicação, é efectuada utilizando o sistema de ficheiros virtual *RelayFs*.

O *Linux Trace Toolkit Viewer (LTTV)* é um projecto desenvolvido em paralelo com o *LTT* e *LTTng*, de modo a criar uma análise visual dos dados recolhidos por estas aplicações. Esta ferramenta possibilita igualmente realizar um traço de execução, uma vez que os dados recolhidos contêm uma estampilha temporal, do momento em que foram obtidos.

### 2.5.1.2 OProfile

O *OProfile* é outra das ferramentas de monitorização do núcleo, esta efectua um perfil de execução das funções do núcleo através de amostragem. Com o *OProfile*, em vez de a cada evento utilizar uma função para obter os dados, apenas a utiliza decorridos um certo número de eventos. O recurso a este critério permite ao *OProfile* ser menos perturbador, uma vez que nem sempre é necessário o universo dos eventos que as ferramentas de monitorização capturam, mas tão somente uma amostra 2.3. Deste modo ao utilizar a amostragem, o *OProfile*, beneficia de um menor grau de perturbação no sistema[Wil].

O *OProfile* é uma ferramenta que efectua um perfil de utilização dos recursos da máquina, utilizando para o efeito a técnica de amostragem. Esta ferramenta é composta por 3 componentes principais, o módulo no núcleo (*oprofile driver*), o *daemon*/processo em nível utilizador que recolhe os dados obtidos da monitorização pelo módulo do núcleo (*opcontrol*) e o *opreport* e *opannotate* que permitem criar relatórios, tal como se pode constatar na figura 2.3.

O *OProfile* não efectua apenas o perfil de utilização do núcleo *Linux*, pois também



Figura 2.3: Arquitectura do *OProfile* [Wil]

cria o perfil de utilização dos processos que estão em execução na máquina. Este perfil é criado com o auxílio do sistema de ficheiros virtual *ProcFs*, que indica quais os programas que estão em execução e respectivas zonas de memória utilizadas por estes.

O *OProfile* regista os valores do *program counter* (registo do processador que contém a próxima instrução a executar) e o tempo quando quando foi — é triplo com estes dados e as zonas de memória e qual o processo corrente consegue efectuar o perfil de utilização do processo.

De modo a reduzir a sobrecarga / contenção utiliza buffers por cpu ...

### 2.5.2 Suporte à monitorização

Nas secções anteriores foram apresentadas ferramentas de monitorização e os seus mecanismos para lidar com a monitorização. As ferramentas são importantes quando existe a necessidade de efectuar análises aos dados obtidos. Nas ferramentas os diversos componentes que as constituem estão muito interligados dificultando, por vezes, a sua integração com outros componentes.

Por seu lado os mecanismos devem ser genéricos tirando partido das arquitecturas para que possam ser desenvolvidas ferramentas que as possam utilizar.

Como suporte à monitorização destaca-se o mecanismo *KProbes* que é um mecanismo de instrumentação dinâmica do *Linux*.

e a ferramenta *Linux Kernel Stace Tracer*.

Em ambos é possível definir novos pontos, funções ou eventos a serem monitoriza- dos, as funcionalidades já existentes nestas ferramentas.

### 2.5.2.1 KProbes

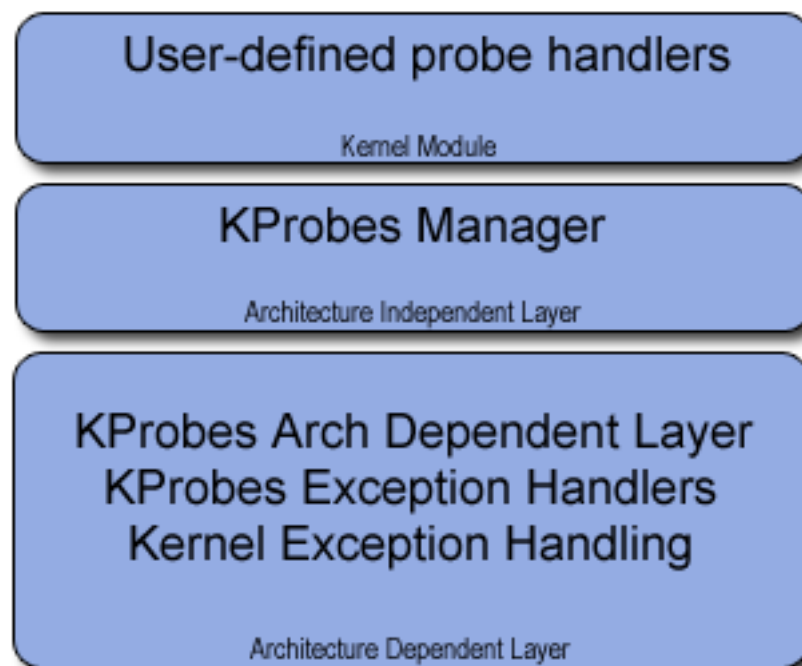
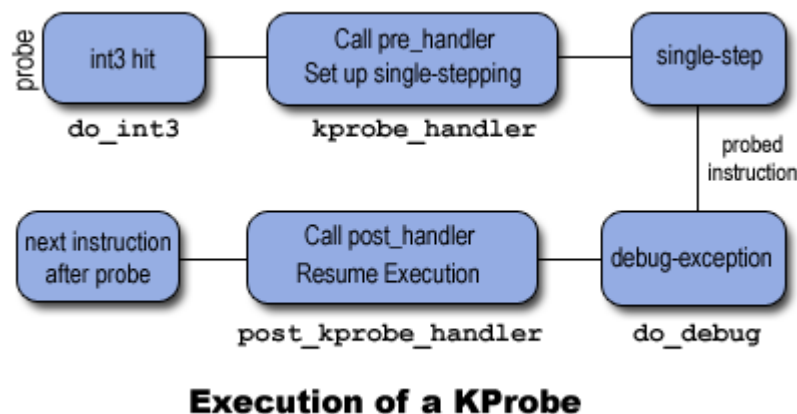
O *KProbes* é um mecanismo de instrumentação dinâmica do núcleo do *Linux*. A API permite ferramentas, como o *DProbes* ou o *SystemTap* possam aceder às suas funcionalidades. Esta encontra-se na versão principal do núcleo do sistema *Linux* desde a versão 2.6.9, o que indica tratar-se de uma ferramenta bastante estável[Pan04, KPr].

Utilizando o *KProbes* existem três tipos de instrumentação que podem ser efectuados: o *KProbe*, o *JProbe* e o *KRetProbe*. Cada um destes três tipos é específico de uma determinada funcionalidade.

- **KProbe** - Utilizando um *KProbe* pode-se analisar a chamada de uma função ou de uma instrução. Para analisar apenas uma instrução, tem de ser indicada a distância relativamente ao início da função instrumentada. A indicação da função a instrumentar, pode ser definida recorrendo ao seu nome ou ao seu endereço de memória. Múltiplos *KProbe* podem ser definidos para uma mesma função ou instrução.
- **JProbe** - Um *JProbe* destina-se a analisar os parâmetros, da função a instrumentar. Quando é declarado um *JProbe* a função de *handler* tem de conter os mesmos tipos de argumentos que a função que se destina a ser instrumentada, de forma a serem monitorizadas.
- **KRetProbe** - Este tipo de análise tem como objectivo obter o valor de retorno da função que se pretende analisar. Para efectuar esta operação é utilizada a técnica do trampolim [HMC94], uma vez que uma função pode terminar em pontos diversos.

Para se obterem informações sobre as funções ou instruções que estão a ser monitorizadas, o *KProbes* disponibiliza-as através de um ficheiro no *DebugFs*. Para utilizar o *KProbes* é necessário criar um módulo para o núcleo do sistema de operação, com informações relativas às rotinas a instrumentar. Para além destas, o módulo tem de conter igualmente os *handlers* de análise, assim como outros parâmetros necessários à realização da instrumentação.

Quando o módulo é inserido no núcleo de sistema, o registo dos *handlers* das funções a serem instrumentadas é efectuado de forma atómica. A execução do *KProbes* não utiliza nenhum *mutex* ou outra forma de controlo de concorrência, apenas funciona com a preempção desligada. Dependendo do contexto, os *handlers* podem ser executados com as interrupções desligadas.

Figura 2.4: Arquitetura do *Kprobes*Figura 2.5: Execução de um *KProbe*

A instrumentação, processa-se copiando a instrução que se quer analisar para uma zona de memória, e essa instrução no código por uma interrupção (*int3*). Quando possível o *KProbes* utiliza instruções do processador especializadas no *debug* de forma a minimizar o grau de perturbação. A utilização da instrução *int3* tem problemas de desempenho associados, por isso algumas formas de ultrapassar esta instrução têm sido desenvolvidos. Uma destas formas foi a utilização de um *jmp* incondicional, desta forma é possível obter um desempenho superior à utilização da instrução *int3*. Apesar de poder ser aplicado em diversas situações, não pode ser sempre utilizado dado que

instruções que dependam do registo do processador *eip* ou *rip*, não podem usufruir desta optimização.

Apesar de ser possível instrumentar praticamente todo o núcleo do sistema de operação, existem algumas situações que podem requerer alguma perícia na forma como são tratadas. Quando o compilador realiza substituição de funções por código *inline*, as instruções que anteriormente estavam dentro da função passam a integrar o ponto onde foram substituídas, deixando de existir um ponto de entrada na função. Assim sendo, a função deixa de existir, impossibilitando a instrumentação destas funções. Não existe a possibilidade de instrumentar recursivamente uma função, ou seja, se uma função está a ser instrumentada e nas funções de *handler* essa função é chamada, o *KProbes* detecta que já está em execução e não permite que seja novamente interrompido, incrementando uma variável que indica que existiu um *miss*.

Este mecanismo de instrumentação do núcleo é utilizado por diversas ferramentas de modo a instrumentar dinamicamente código pertencente ao núcleo do *Linux*. Duas das ferramentas que tiram partido deste mecanismo são: *DProbes* e o *SystemTap*.

**DProbes** O *DProbes* utiliza o *KProbes* como suporte para efectuar a instrumentação dinâmica do núcleo de sistema. Como a criação de pontos de análise é escrito em linguagem C ou *assembly*, foi desenvolvida uma linguagem de mais alto nível de forma a simplificar a utilização desta aplicação.

Após os dados terem sido obtidos, estes podem ser transferidos para um ficheiro, para o *daemon logging* do núcleo de sistema, ou para uma porta série. Existe igualmente a opção de interoperabilidade com o *Linux Trace Toolkit*[DPr].

**SystemTap** O *SystemTap* é uma ferramenta que possibilita o desenvolvimento de módulos para o *KProbes*, utilizando uma linguagem específica, de forma a ser segura e fácil de trabalhar. Uma vez que os dados gerados estão no espaço de endereçamento de memória do núcleo, e o programa que os analisará se situa no espaço de endereçamento do utilizador, é necessário efectuar uma transferência de dados de um para outro espaço. O *SystemTap* efectua esta transferência, recorrendo à utilização do sistema de ficheiros *ReplayFs*, onde é possível escrever de forma rápida, sem comprometer a segurança do sistema[DIH<sup>+</sup>07, Jon09].

Apesar de no pacote de aplicações desta ferramenta, existir um visualizador gráfico de dados, recolhidos pelo *SystemTap*, este só consegue analisar os *TapSets* pré-definidos no *SystemTap*. Para ultrapassar esta limitação, novas ferramentas estão a ser desenvolvidas. Uma destas ferramentas designada por *bootlimn*, está actualmente a ser desenvolvida no *Google Summer of Code*, com o objectivo de utilizar o *SystemTap* para recolher informações sobre o arranque do sistema, agregando estes dados no formato *XML*. Os

dados recolhidos, permitem visualizações gráficas do processo de inicialização do sistema, bem como da utilização de disco e *CPU*. Outro projecto conhecido como *Systemtap GUI*, que engloba o *System Tap Editor Plug-in* para a ferramenta Eclipse, é um ambiente onde podem ser analisados e visualizados os dados recolhidos pelo *System-Tap*.

### 2.5.2.2 Linux Kernel State Tracer

O *Linux Kernel State Tracer* (LKST) obtém informações referentes ao núcleo de sistema, de forma a criar um traço de execução, e consegue capturar diferentes eventos como trocas de contexto, envio de sinais, alocação de memória, transmissão de pacotes, etc.

Actualmente, está a ser desenvolvido um subprojecto do *Linux Kernel State Tracer* designado por *Direct Jump Probe*. O *Direct Jump Probe* é uma optimização à utilização do *trap int3*, presente em alguns processadores, e que pode trabalhar em conjunto com o *KProbes*. Esta optimização pode ser verificada no relatório[Hir05].

### 2.5.3 Comparação entre os diferentes sistemas de instrumentação

Tabela 2.1: Tabela Comparativa dos sistemas de instrumentação

| Instrumentação | Amostragem / Traço | Análise de Parâmetros | Deamon        |
|----------------|--------------------|-----------------------|---------------|
| KProbes        | Traço              | Permite               | Não necessita |
| LKST           | Traço              | Permite               | Necessita     |
| LTT            | Traço              | Não permite           | Necessita     |
| OProfile       | Amostragem         | Não permite           | Necessita     |

A tabela 2.1 permite fazer uma comparação entre algumas das características destes sistemas de instrumentação presentes no núcleo de sistema do *Linux*, segundo os seguintes critérios: metodologia da captura (por traço ou por amostragem), análise dos parâmetros das funções instrumentadas e a necessidade de ter um *daemon* para a recolha de dados provenientes do sistema de monitorização. Se a necessidade de ter um *daemon* a executar de forma a coligir e organizar os dados, pode penalizar o desempenho do sistema, a possibilidade de analisar os parâmetros das funções instrumentadas é um ponto a favor dos sistemas *KProbes* e do *LKST*.

## 2.6 Transferência de dados

Quando se recorre a alguma fonte para a obtenção de dados, não é imprescindível que estes sejam apenas requeridos no sítio onde são adquiridos. Assim, quando dados que foram obtidos pelo núcleo de modo a serem transferidos para as aplicações

No intuito de melhorar a partilha de recursos externos, o controlo dá-se ao nível do núcleo do sistema, pelo que parte das comunicações dos utilizadores com dispositivos externos, é efectuada através do núcleo de sistema. Não sendo apenas exclusivo dos recursos externos, recursos que sejam partilhados por mais que um processo, em geral são partilhados através do núcleo. Desta forma é necessário existirem modos de comunicação / transferência de dados entre o núcleo e os processos, por isso existem diversos sistemas de ficheiros virtuais que permitem o controlo e a disponibilização aos processos dos dados adequados.

### 2.6.1 Técnicas de transferência de dados

Devido há necessidade de transferir informações entre diferentes *buffers* dentro do núcleo de sistema, foram desenvolvidas técnicas tendo sempre como referência a minimização da sobrecarga. De entre as diferentes técnicas de transferência interna de informação ao núcleo do sistema, merecem especial relevo:

**MMAP** - Esta técnica é implementada utilizando páginas de memória partilhadas, reduzindo as transferências e gastos de memória, proporciona a possibilidade de mapear um canal para memória potenciando assim, a partilha de dados não só entre diferentes processos como igualmente entre os processos e o núcleo, que necessitem de aceder aos dados do mesmo canal. A partilha de dados de *IO* é uma vertente que tem sido objecto de desenvolvimento, de forma a aumentar o desempenho, porquanto permite a partilha de dados entre um ou mais programas, e o núcleo de operação. Esta partilha evita a necessidade de copiar dados dos espaços de endereçamento do núcleo para o do utilizador e vice versa.

**Zero Copy** - Esta técnica de transferência de dados sem que existam cópias dos *buffers* pertencentes ao núcleo de sistema e ao espaço de utilizador, permite que os dados sejam partilhados por diferentes entidades, sem necessidade de criação de novos *buffers* e cópias de dados, uma vez que apenas são passadas as referências.



**Ring Buffers** - Esta técnica consiste num aproveitamento dos recursos já alocados, mas que se tornaram desnecessários, estando desta forma disponíveis para nova utilização. Os *Ring Buffers* são utilizados tendo presente a relação custo / benefício, isto é, quando o peso da criação e destruição de elementos é comparativamente superior à reutilização dos recursos já alocados. É igualmente utilizada em situações de "produtor-consumidor", ou seja, quando é necessário manter a ordem dos elementos.

### 2.6.2 Interface do sistema com os processos

Devido à necessidade de análise de estruturas e dados do núcleo de sistema, foram criados diferentes subsistemas com vista à sua satisfação. Um destes subsistemas é o *ProcFs*, que existe no núcleo de sistema do *Linux* desde as primeiras versões. Outros como o *DebugFs* ou o *RelayFs* são mais recentes e com novas abordagens.

De forma a obter informações relativas a estruturas dentro do núcleo de sistema, foram referenciados os seguintes sistemas de comunicação entre o utilizador e o núcleo, a saber:

**ProcFs** - Desenvolvido para obter informações relativas aos processos tem sido utilizado desde as primeiras versões do núcleo de sistema do *Linux*. Apesar de novos sistemas como o *SysFs* terem sido criados, continua a ser extensivamente utilizado pelas aplicações, pois este contém informações disponíveis para o nível utilizador, sobre estruturas e dados dos processos localizadas no núcleo. Esta é uma das *API's* de comunicação, entre o núcleo e as aplicações, mais utilizadas, não obstante ter crescido de forma desorganizada.

**SysFs** - Este sistema de ficheiros virtual foi desenvolvido para colmatar algumas deficiências encontradas no *ProcFs*. Estes problemas situam-se basicamente na forma desorganizada como a quantidade de informação disponibilizada está distribuída sobre o *ProcFs*. Este novo sistema de ficheiros virtual, evidencia restrições na disponibilização das informações, uma vez que apenas é possível visualizar e modificar um *KObject* por ficheiro. Esta limitação conduziu a que outros sistemas de ficheiros virtuais, nomeadamente o *DebugFs* e o *RelayFs*, fizessem a sua aparição com vista a colmatar esta situação.

**DebugFs** - Este sistema foi essencialmente criado para ultrapassar as dificuldades sentidas pelos programadores, na utilização de um sistema de ficheiros tão restritivo como o *SysFs*. Apesar de já existirem dois sistemas de ficheiros virtuais, o *ProcFs* e o

*SysFs*, este tentou colmatar os problemas que se apresentavam, não obstante não se mostrar tão estruturado como o *SysFs*, tem melhor organização de dados que o *ProcFs*.

O sistema de instrumentação do núcleo de sistema *Linux*, *KProbes*, utiliza este sistema de ficheiros virtual, de forma a apresentar os pontos, onde existe instrumentação no núcleo de sistema.

**RelayFs** - Este sistema de ficheiros virtual foi desenvolvido tendo em mente a transferência de grandes quantidades de dados entre o núcleo de sistema e o espaço de utilizador. O *RelayFs* responde a esta exigência, fazendo uso de novas primitivas onde são mais reduzidas as zonas de controlo de concorrência[DIH<sup>+</sup>07, TZYW03].

**NetLink** - O *NetLink* utiliza uma método de comunicação baseado em *sockets*, para estabelecer a comunicação entre o núcleo de sistema e os processos, em nível utilizador. Tendo como abstracção os *sockets*, este sistema tem igualmente a possibilidade de enviar dados para múltiplos processos, devido à utilização das primitivas de envio colectivo (*Multicast*). É com base no *NetLink*, que a comunicação entre processos dentro da mesma máquina (*IPC*) é implementada. A comunicação com diferentes partes do subsistema de rede é efectuada recorrendo a este sistema de *sockets*.

### 2.6.3 Comparação entre sistemas de transferência de dados

| Sistema | Estruturação de Dados | Volume de dados  |
|---------|-----------------------|------------------|
| ProcFs  | Com                   | Reduzido         |
| SysFs   | Com                   | Reduzido         |
| DebugFs | Com                   | Reduzido         |
| RelayFs | Sem                   | Bastante Elevado |
| NetLink | Sem                   | Elevado          |

Tabela 2.2: Tabela Comparativa de transferência de dados entre processos e núcleo de sistema

Como se pode verificar através da tabela 2.2, que compara a estruturação dos dados de cada Sistema, com o volume de dados que é possível transferir, o *RelayFs* e o *NetLink* são os dois sistemas de comunicação, que não tendo estruturação fixa dos dados, conseguem transferir um volume superior de dados, comparativamente com os restantes, objecto da análise.

## 2.7 Captura de tráfego de um processo específico

A captura do tráfego respeitante a um processo genérico, foi alvo de estudo em [LML09] ou e em [Far09], esta última objecto de uma dissertação de mestrado.

No primeiro trabalho, foi desenvolvido um sistema de captura de pacotes de um determinado processo, utilizando um módulo no núcleo de sistema que intercepta e captura os pacotes do processo.

Este sistema é constituído por três componentes essenciais, uma dentro do núcleo de sistema e duas em nível utilizador. Na realização do trabalho, foi utilizada a ferramenta *KProbes* para a monitorização de algumas funções do núcleo de sistema *Linux*, de forma a conhecer quais os portos que vão ser utilizados por uma determinada aplicação. Logo que obtida, a informação é enviada para um processo em nível utilizador que tem o registo de todos os portos que estão a ser utilizados pela aplicação objecto de monitorização. Se esse porto não estiver a ser monitorizado, essa informação é passada a outro processo que utiliza a biblioteca *PCap* de forma a capturar o tráfego existente nesse porto.

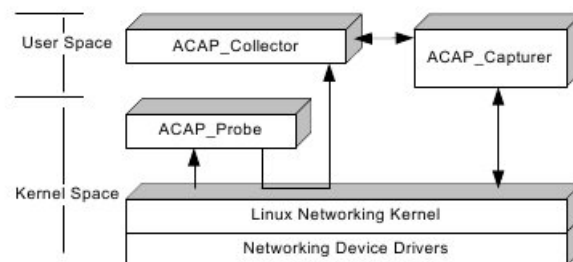


Figura 2.6: Arquitectura da monitorização de tráfego

Analisando a arquitectura de monitorização de tráfego representada na figura 2.6, o ACAP\_Collector e o ACAP\_Capturer, estão em nível utilizador, de onde se infere que o desempenho desta ferramenta é passível de ser aumentado, caso esta seja completamente implementada dentro do núcleo de sistema de operação.

Relativamente ao segundo trabalho, foram implementadas duas abordagens: uma com monitorização da aplicação e outra através de informações pertencentes ao núcleo do sistema de operação. No que à primeira diz respeito, a monitorização efectuada processou-se através da interceptação das chamadas à biblioteca *LibC*, para a utilização de *sockets*, criando uma biblioteca partilhada com a mesma sintaxe das chamadas que são utilizadas. A opção pela utilização deste método, implica definir a variável de

ambiente *LD\_PRELOAD*, de forma a operar esta intercepção. Como esta biblioteca está em nível utilizador, mostra-se necessário capturar todos os pacotes e apenas em nível utilizador visualizar o tráfego respeitante ao processo. Esta obrigatoriedade de capturar todos os pacotes pode constituir uma tarefa com elevado grau de perturbação do sistema.

No que à segunda abordagem se refere, o método de monitorização utilizado baseou-se na consulta efectuada de forma regular com intervalos reduzidos, dos dados referentes aos portos de comunicação, utilizados pela aplicação, que são exportados pelo núcleo através do *ProcFs*. Esta forma de monitorização consome demasiados recursos e não se mostra totalmente fiável, na medida em que quanto menor o intervalo de tempo utilizado, maior é a perturbação apresentada pelo sistema.

O dinamismo das aplicações, nomeadamente das aplicações multimédia, deu origem a diversos estudos sobre a forma de monitorização de rede, que estas necessitam. Estas aplicações utilizam diversos fluxos de dados, designadamente de transmissão e de recepção. Em geral, as aplicações multimédia, com base na *internet* utilizam uma metodologia cliente/servidor, onde o servidor aguarda pedidos do cliente num determinado porto. O cliente, conhecendo antecipadamente este porto, liga-se. A partir deste ponto, iniciam-se trocas de informações, que irão originar a troca de portos dinâmicos e posteriormente o processo de transmissão/recepção de dados multimédia.

As aplicações multimédia assentes na *internet* são apenas um exemplo de aplicações com diversos fluxos, em que as portas de comunicação entre as aplicações, são negociadas dinamicamente.

Como o referido em 2.3, a captura de pacotes é definida em filtros estáticos e para capturar este tipo de tráfego, é necessário modificar os filtros definidos, de modo a acompanhar o protocolo. Esta forma de captura mostra-se bastante ineficaz, o que motivou a que fossem estudadas algumas alternativas, tendo em vista a correcção desta situação. Os projectos *mmdump*[vdMChCS00], e *Swift*[WXW08] são dois destes casos estudados, que merecem especial relevância e que adiante se analisam:

**MMDump** - É uma ferramenta de monitorização de protocolos multimédia com suporte na rede. Esta aplicação tem como base o *tcpdump*, sendo a captura de pacotes efectuada através da utilização de filtros. Para determinar os portos a obter, é necessário analisar o conteúdo dos pacotes direccionados a portos específicos, e a partir destes é possível identificar os novos portos negociados dinamicamente pela aplicação, e proceder à alteração dos filtros a aplicar. Como a alteração, que é constituída pela cópia do novo filtro para o núcleo e verificação de segurança, de forma a validá-lo, é um processo demorado, é necessário reduzir este tempo, com o objectivo de ter uma aplicação

que consiga minimizar o grau de perturbação no sistema.

Na alteração do filtro, foi verificado que existe um certo padrão, que consiste em pré-estabelecer uma parte comum a ser adicionada ao filtro, e apenas alterar a parte referente aos portos.

Esta forma de monitorização é muito específica, pelo que é imprescindível todo o protocolo interno de comunicação. Assim, para cada novo protocolo a monitorizar, é necessário acrescentar um novo módulo com a interpretação desse protocolo.

*Swift* - É uma ferramenta de criação de filtros, cujo principal objectivo é a melhoria do desempenho da utilização destes, na captura do tráfego de rede. Nesta ferramenta, foi avaliado o tempo de alteração dos filtros, utilizando o *Linux Socket Filtering*, pois estes são os filtros de referência no sistema de operação *Linux*. A aplicação destes filtros a partir da biblioteca *PCap* compreende três fases: cópia do filtro definido em nível utilizador para o núcleo de sistema, verificação de segurança, e aplicação do filtro. De forma a diminuir a latência de actualização dos filtros, foi criada uma especificação de modo a que estes não necessitassem de ser analisados relativamente à segurança, visto que a própria linguagem garante as propriedades de segurança necessárias. Com este novo *instruction set*, e sem necessidade de verificação, foi reduzida a latência de actualização dos filtros, o que conduziu a um aumento de desempenho na utilização de filtros dinâmicos.

## 2.8 Conclusão

Como se pode observar neste capítulo, existe a possibilidade de melhorar a monitorização de rede do *Linux*. O mecanismo de instrumentação genérico do *Linux*, *KProbes*, permite que se efectuem análises no núcleo de modo generalista, o que para obter as interacções dos processos com as interfaces de rede sem que se assista à instrumentação do código dos processos e com baixa sobrecarga será um sistema de instrumentação dinâmico a ter em conta para o desenvolvimento de um mecanismo de monitorização genérico de rede de um processo.

Sistema de *logging*, são por vezes um peso demasiado elevado e por isso utilizam mecanismos para fazer face a este problema de transporte da informação dos dados do núcleo para o nível utilizador o mais eficiente possível.

Quaisquer das soluções apresentadas tem limitações e defeitos ...





## Sistema Proposto

De modo a conceber a arquitectura do mecanismo de Monitorização de Rede orientado ao Processo (*MRoP*), foi necessário conhecer, como os processos em nível utilizador interagem com o exterior, via rede. Uma vez conhecida a arquitectura de rede do *Linux*, tornou-se imprescindível compreender a forma, como se processa a comunicação e a monitorização de rede.

Tendo em conta os factores anteriormente mencionados, este capítulo irá apresentar o funcionamento dos processos e a sua estruturação, bem como, a constituição do sistema de rede do *Linux*, desde o momento do envio dos dados pelo processo até estes atingirem a interface de rede, tendo em vista desenhar o sistema *MRoP*. Considerando a abrangência desta visão, apenas serão focados os componentes essenciais da arquitectura, nomeadamente o descritor de processo, as famílias de *sockets*, a *firewall*, o sistema de escalonamento de rede e a monitorização de rede.

Após esta visão geral dos processos e do sistema de rede do *Linux*, será apresentada a arquitectura proposta para o sistema de Monitorização de Rede orientado ao Processo (*MRoP*).

### 3.1 Estrutura de um processo

Um processo é uma instânciação de um programa em execução, sendo igualmente uma unidade de escalonamento de execução (*task*) no sistema de operação . Este consome

recursos (físicos e lógicos), partilhados com outros processos num sistema multiprogramado. O núcleo de sistema gere uma estrutura para cada processo (*struct\_task*), onde estão identificados todos os recursos a este atribuídos. Esta estrutura contém apontadores para diversos recursos nomeadamente: zonas de memória requisitadas, canais abertos (ficheiros, *sockets*, entre outros), contabilização de utilização de *cpu*, apontadores para a sua árvore genealógica (pai, irmãos e filhos), etc [BDK<sup>+</sup>97, BDKV02].

A árvore genealógica, anteriormente referida, contém o apontador para o processo pai (o processo que efectuou o *fork*, que lhe deu origem), a lista de irmãos e dos descendentes. O processo pai, tal como cada um dos elementos presentes na lista, são estruturas do tipo *task\_struct*, possibilitando, desta forma, a navegação ao longo da árvore genealógica do processo.

Um dos recursos partilhados pelos diferentes processos é o acesso à rede, ou seja, o recurso que permite a comunicação com o exterior. Para que as comunicações tenham lugar, é necessária a alocação de canais de comunicação, cuja criação, utilização e destruição é efectuada pela *API* definida no sistema de operação. Quando um processo cria um canal, o núcleo devolve-lhe um identificador, permitindo ao processo referenciar o canal dentro do núcleo e, deste modo, efectivar o seu controlo e realizar futuras comunicações. No núcleo, este identificador é gerido na estrutura que identifica os canais abertos, pelo processo.

Para reconhecer qual o tipo do canal aberto, existem funções específicas que o analisam e lhe permitem efectuar operações. Assim, ao manipular um ficheiro, existe a possibilidade de avançar e recuar, tendo como referência um determinado ponto. Todavia, esta situação deixa de fazer sentido, quando se pretende controlar um *socket* ou um *pipe*, na medida em que as comunicações nestes são destruídas quando consumidas (situação que não se verifica aquando da manipulação de um ficheiro).

Apesar destas informações estarem disponíveis no descritor de processo, não existe suporte neste descritor de modo a obter as interacções do processo com as interfaces de rede. Desta forma, apenas através da instrumentação das chamadas ao sistema, é possível obter as interacções anteriormente referidas. Assim, quando um processo comunica com a rede, executa uma chamada ao sistema e por intermédio da instrumentação destas, é possível obter algumas informações relevantes que, combinadas com os dados do descritor do processo, permitem que se obtenham as informações relativas aos portos, endereços e protocolos utilizados na comunicação com a rede. Com estes dados (protocolos, portos e endereços) é possível filtrar e capturar eficientemente os dados, através da extensão efectuada ao *LSF*.



## 3.2 Arquitectura de rede em *Linux*

Os processos necessitam de comunicar para obter dados, de modo a completar as suas execuções. Estas comunicações podem ser efectuadas interna ou externamente ao sistema. Em geral, as comunicações externas processam-se através de uma interface de rede, sendo necessário, para tal, criar canais de comunicação. A(s) interface(s) de rede são recursos do sistema, que são partilhados pelos diversos processos. Esta partilha é realizada pelo núcleo, pois apenas este tem a capacidade de a efectuar correctamente.

A chamada ao sistema *socket*, é efectuada para a criação de um canal de comunicação e este varia em função dos parâmetros: *família*, *tipo* e *protocolo*. Existe um agrupamento em família de endereços (*Address Family*), consoante se destina à comunicação remota ou local. Em termos gerais, as comunicações locais podem incidir na comunicação entre processos (*AF\_UNIX*), ou entre estes e o núcleo (*AF\_NETLINK*), enquanto as comunicações remotas podem ter lugar sobre protocolos da Internet, dispondo igualmente de uma família própria para o efeito (*AF\_INET*). Estes canais possuem um conjunto de funções (*API POSIX*) bem definidas, para utilização e controlo das operações, permitindo estruturar a comunicação de um modo eficiente.

Hoje em dia, os administradores de sistemas necessitam filtrar o fluxo de informação que circula nas suas redes. Para atingirem este objectivo, tem-se assistido ao desenvolvimento de sistemas de filtragem de comunicações, conhecidas por *firewalls*. Sendo o *Linux* um sistema de operação frequentemente utilizado em ambientes de servidores, tornou-se essencial a aplicação deste tipo controlo. A solução encontrada resultou no desenvolvimento de uma *firewall*, denominada de *NetFilter*, capaz de filtrar os fluxos de dados que circulam através da interface de rede.

Para além da existência da filtragem do fluxo de dados, o sistema de operação *Linux* dispõe de um sistema de escalonamento do tráfego de rede, (*Traffic Control*), que permite ao administrador indicar quais os fluxos de dados prioritários ou que necessitam de determinada largura de banda, possibilitando-lhe efectuar uma reserva antecipada da mesma.

Adiante apresentar-se-ão diferentes constituintes da estrutura de rede do *Linux*, desde as famílias de *sockets* até aos controladores das interfaces de rede.

Por último será apresentado o sistema de monitorização genérica de rede do *Linux*.

### 3.2.1 Sockets e as suas famílias

Como referido na secção 3.2, um processo para comunicar via rede, tem de criar um canal. Estes são criados para efectuar comunicações locais ou remotas e, embora as acções realizadas sejam relativamente comuns, o meio de transmissão é diferenciado,

assim como os canais e as formas utilizadas.

No núcleo do sistema, encontram-se implementadas diferentes famílias de endereços tal como apresentado na figura 3.1. No entanto desse conjunto destancam-se as famílias de endereços *UNIX*, *NETLINK*, *INET* e *PACKET*. [Ben05, WR05]

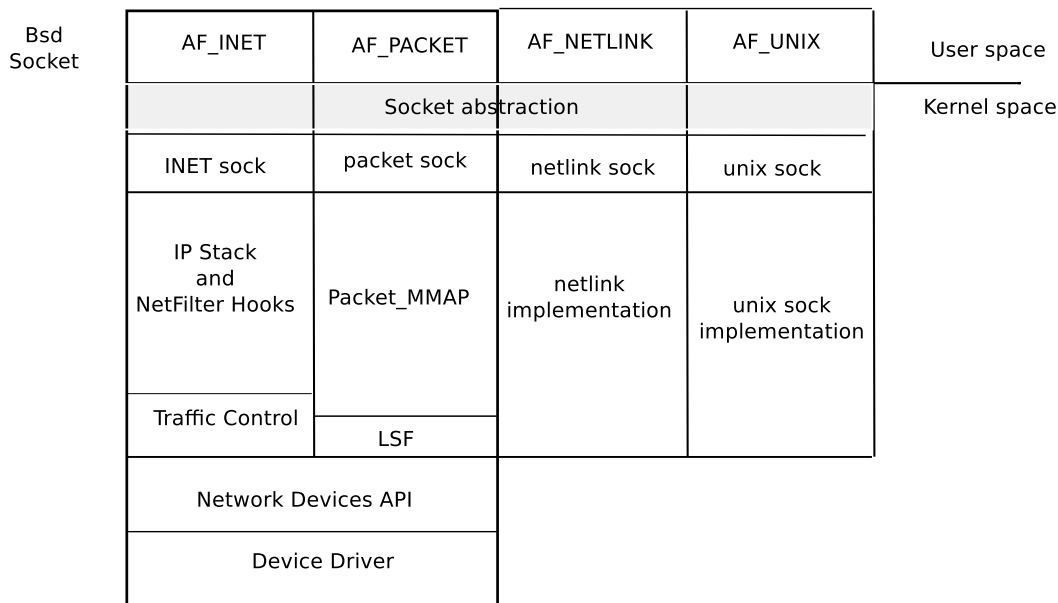


Figura 3.1: Arquitectura parcial de sockets do Linux

### 3.2.1.1 AF\_UNIX

A *AF\_UNIX* consiste numa família de *sockets* utilizada para efectuar a comunicação entre processos dentro da mesma máquina. É um dos sistemas de *Inter Process Communication (IPC)*, utilizado em sistemas *Unix*, que permite utilizar um ficheiro, contido no sistema de ficheiros, como um canal de comunicações.

### 3.2.1.2 AF\_NETLINK

A família *NETLINK* é utilizada pelos processos, em nível utilizador, para comunicar com o núcleo. É possível efectuar comunicações ponto-a-ponto ou multi-ponto, isto é, é possível que um ou mais processos comuniquem com o núcleo, designadamente o *netfilter*, o sistema de encaminhamento de pacotes de rede, ou ainda o sistema de configuração de interfaces de rede, etc., através de um *socket* nele criado. Cabe aos processos, em nível utilizador, conectarem-se ao *socket* definido no núcleo, o qual não é completamente passivo, na medida em que este pode iniciar comunicações assíncronas com os processos.

Embora as interfaces de rede sejam configuradas através do programa *ifconfig* em nível utilizador, o qual utiliza *ioctl*s para efectuar essa configuração, uma outra ferramenta foi desenvolvida (*ethtool*), que tira partido de *socket netlink* para efectuar estas configurações, dado que as *ioctl*s não permitem especificar correctamente os parâmetros, contrariamente ao que sucede nos *socket netlink*.

### 3.2.1.3 AF\_INET

A *AF\_INET* representa a família de protocolos utilizada na comunicação através da *Internet* e que faz uso do protocolo *IP versão 4*.

A estruturação por camadas permite que a implementação de protocolos seja efectuada de forma simples e rápida, sendo que ao utilizar as camadas inferiores como suporte para as superiores, aumenta a abstracção e complexidade dos protocolos.[SV08] Existem vários protocolos de nível transporte sobre *IPv4*, sendo os mais utilizados o *TCP* e *UDP*, conforme evidenciado nas figuras 3.2 e 3.3. A criação de canais sobre o nível de transporte é efectuada utilizando a chamada ao sistema *socket*, indicando para o parâmetro tipo, os valores *SOCK\_STREAM* ou *SOCK\_DGRAM* para os protocolos *TCP* e *UDP*, respectivamente.

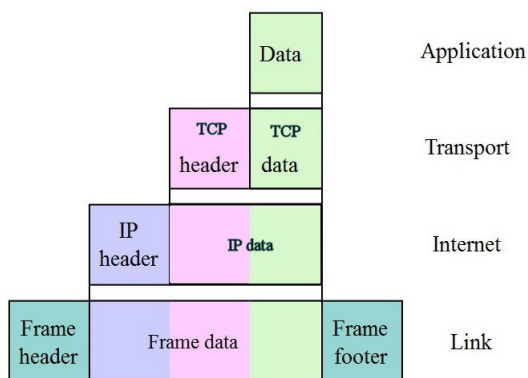


Figura 3.2: Protocolo TCP

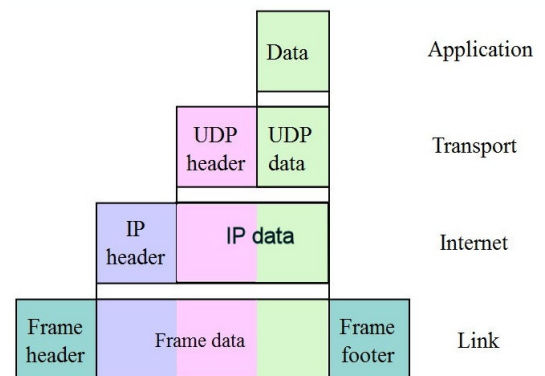


Figura 3.3: Protocolo UDP

O *TCP* é utilizado para comunicações com controlo de fluxo de dados, que se adaptam ao canal existente, tolerando a perda momentânea de pacotes e procedendo à sua retransmissão logo que possível. O controlo das retransmissões é efectuada através de números de sequência e do envio de pacotes (*acknowledges*), que informam o transmissor dos que foram correctamente recebidos, permitindo-lhe reenviar os que se encontram em falta.

O *UDP* é um protocolo mais leve que o *TCP*, pois não utiliza o controlo anteriormente referido. É habitualmente utilizado quando se está perante a possibilidade de

perda de pacotes durante a transmissão ou quando o controlo da retransmissão destes, é efectuado em camadas superiores.

Para além dos protocolos atrás referidos, existe ainda um denominado *SOCK\_RAW*. Este protocolo permite o acesso directo ao nível rede, através da criação de um canal que comunica directamente com o nível rede da pilha de protocolos *TCP/IP*, permitindo executar protocolos de comunicação em nível utilizador. Para utilizar *sockets* através do modo *RAW*, é necessário que o utilizador tenha permissões de *super user*, uma vez que neste modo, é permitido ao utilizador especificar parte dos cabeçalhos dos pacotes a enviar, sem que o núcleo os valide. Contudo, a não validação, pode proporcionar que pacotes maliciosos, isto é, com falhas deliberadas, possam ser introduzidos na rede.

#### 3.2.1.4 *AF\_PACKET*

A família *AF\_PACKET* comunica directamente com o controlador da interface de rede, possibilitando assim efectuar a monitorização de rede. A utilização destes *sockets* requer uma autorização de *CAP\_NET\_RAW*, apenas disponível ao *super user*. Este requisito previne que utilizadores não autorizados possam monitorizar ou injectar pacotes na rede, influenciando o seu comportamento. Por dispor da possibilidade de enviar pacotes directamente para o controlador de rede, uma das suas utilizações é a implementação de protocolos de rede em nível utilizador. Os canais da família *AF\_PACKET* são frequentemente utilizados em sistemas de detecção de intrusos, uma vez que permitem analisar os pacotes e detectar falhas na sua formatação.

Como a obtenção dos pacotes por estes canais, é efectuada antes da recepção destes pelos processos, permite que sistemas como o *NetFilter* (secção 3.2.2) bloqueem a recepção dos pacotes, impedindo as aplicações de os receberem, não obstante o canal já os ter obtido para análise.

Como todos os pacotes recebidos pela(s) interface(s) de rede são obtidos por este canal, é necessário efectuar uma cópia destes e fornecê-los à aplicação em nível utilizador. Esta monitorização implica um peso extra na computação, pelo que diversas técnicas (designadamente *lazy cloning*, *mmap*, etc.) têm sido desenvolvidas no sentido de minimizá-lo. A utilização de *lazy cloning*, destina-se apenas a efectuar a cópia, caso esta seja absolutamente necessária, permitindo reduzir a perda de eficiência do sistema, aquando da monitorização dos pacotes de rede.

### 3.2.2 NetFilter

Este sistema, está implementado no núcleo do sistema de operação do *Linux*, que controla o fluxo de dados dos processos de e para as interfaces de rede. O *NetFilter* está presente na pilha de protocolos *TCP/IP* em apenas cinco pontos (*PRE ROUTING*, *LOCAL IN*, *FORWARD LOCAL OUT* e *POST ROUTING*), em cada um dos quais, existe uma lista de funções a ser executada sobre o pacote, que foi recebido ou transmitido. Após a finalização destas funções e dependendo do valor retornado, o pacote irá ou não, prosseguir para as restantes camadas, até atingir o *Traffic Control* (apresentado na subsecção 3.2.3) ou a aplicação, efectuando deste modo, o controlo sobre o fluxo de rede. O *NetFilter* pode ser controlado pelo *iptables*, uma ferramenta em nível utilizador, que através de regras, controla as entradas e saídas de dados das comunicações de rede.

Um dos componentes do *NetFilter* designado por *connection tracking*, permite que se efectue o acompanhamento das ligações desde o seu início até ao seu *terminus*. Para realizar este acompanhamento, este componente necessita conhecer os diferentes protocolos utilizados. Tal como o restante sistema, pode ser controlado pelo administrador utilizando uma ferramenta de nível utilizador, no caso o *conntrack*. O *connection tracking*, permite efectuar uma gestão mais eficiente e inteligente das conexões, sendo geralmente referido como *firewall* com estado. Por outro lado, é igualmente utilizado para efectuar *Network Address Translation* (*NAT*) sobre as interfaces de rede.

### 3.2.3 Traffic Control

No núcleo, para além da existência do *NetFilter*, que controla o fluxo de dados na rede, existe o *Traffic Control* que efectua o escalonamento da transferência de dados para o exterior. Na rede, o escalonamento do tráfego é particularmente importante para os seus administradores e, dado que a largura de banda é um recurso limitado, a sua gestão rigorosa é criteriosamente observada.

A gestão é efectuada através de regras definidas em função da largura de banda, ritmo de envio, ou outros parâmetros. O tráfego baseado nas regras anteriormente referidas, é agrupado em classes, que podem ter diferentes níveis de prioridade. Para além desta classificação, é ainda possível definir os algoritmos de escalonamento a utilizar, de modo a definir diversas políticas de qualidade de serviço.

Quando se verifica a iminência do envio de um pacote pela função *ip\_output*, este é transmitido para o *Traffic Control*, onde o pacote é marcado com a *tag* da respectiva classe e ao *QDisk* correspondente. Apartir deste momento, o modo e o ritmo de envio dos pacotes, para a interface de rede, é efectuado segundo as regras definidas no *QDisk*

a que estes pertencem.

### 3.2.4 Interfaces de rede

A interface de rede é o dispositivo que efectua a transmissão e recepção de dados na rede. Do ponto de vista do *cpu*, a interface de rede apenas efectua pedidos de interrupção, sendo que o restante trabalho é realizado pelo controlador de rede, que efectua a ponte entre as funcionalidades de rede do núcleo e a interface de rede.

O núcleo mantém a informação sobre as interfaces de rede, inicializadas pelos seus controladores. Estas, independentemente de estarem ou não em utilização, constam de uma lista duplamente ligada de *net\_devices*. Em cada posição desta lista existem informações referentes a uma interface de rede, nomeadamente a informação sobre o seu endereço *IP*, o seu endereço *MAC*, etc., bem como outras configurações.

No registo de um novo controlador de interface de rede definem-se as operações a executar em determinados eventos, tais como a recepção e transmissão de *frames*, activação e remoção da interface, etc. Estas operações são definidas através de uma interface comum (*struct netdev\_ops*), que permite afectar a esta estrutura, funções presentes nos módulos dos controladores, de modo a serem efectuadas as acções necessárias à utilização da interface de rede.

#### 3.2.4.1 Estrutura de um *sk\_buff*

Os dados envolvidos no fluxo de comunicação e utilizados pelos controladores correspondem aos *socket buffers*, estruturas do tipo *sk\_buff*. Esta estrutura contém diversos elementos, merecendo especial relevância os apontadores para a estrutura de rede do controlador, os apontadores para secções do pacote, os diversos cabeçalhos pertencentes ao nível de rede e transporte, bem como a dimensão dos espaços alocados para estes e outros apontadores.

### 3.2.5 Captura dos fluxos de dados das interfaces de rede

O sistema de monitorização de rede, opera de forma transparente à normal utilização da rede, por parte das aplicações. Desta forma, quando um pacote chega à interface de rede, esta envia ao *cpu* um pedido de interrupção da computação. Neste ponto, é desligada a atenção do processador a novas interrupções, passando a computação para o controlador da interface de rede. Com a atenção do processador a novas interrupções desligada, a computação deve ser breve e restabelecer-se de imediato, de modo a que o normal funcionamento seja retomado. Apenas a computação crítica é efectuada,

diferindo a restante execução através da invocação de um *softirq*, para que seja terminado o tratamento dos pacotes que chegaram à interface. O escalonamento do *softirq* potencia o aproveitamento dos recursos ao equilibrar a execução de interrupções com as restantes tarefas.

Os pacotes recebidos são entregues aos *sniffers* registados no sistema, antes de o serem aos *packet handlers* respectivos. A monitorização de rede é efectuada pelos *sniffers*, sendo os pacotes entregues a cada um deles, para que possam proceder à sua análise. Cada *sniffer* tem um filtro associado escrito em linguagem *bpf*, a ser executado na máquina virtual, implementada para o efeito.

A modificação da função que executa o filtro, de modo a incluir a chamada de um novo sistema de filtragem, permite a extensão do *LSF*, sem que contudo, se assista a um elevado aumento da sobrecarga quando esse sistema de filtragem não estiver activo.

### 3.3 Arquitectura do MRoP

O sistema proposto foi desenvolvido procurando cumprir os seguintes requisitos:

- seleccionar as comunicações que envolvem apenas um processo (ou um conjunto de processos);
- manter a compatibilidade com o sistema já existente, incrementando a sua funcionalidade;
- minimizar eventuais perdas de desempenho;
- a implementação deve envolver poucas alterações ao código do sistema, de modo a facilitar a sua manutenção e evolução aquando das novas versões do sistema *Linux*.

O mecanismo criado está dividido em quatro componentes principais: filtragem dos pacotes de rede, instrumentação das chamadas ao sistema, repositório do estado das interacções via rede, controlo e informação sobre o estado da monitorização (ver figura 3.4).

A função de filtragem, invocada por um *hook* que estende o *LSF*, permite que apenas o tráfego do processo alvo seja analisado pelo restante sistema de filtragem do *LSF*. Relativamente à componente instrumentação das chamadas ao sistema (ou outras funções contidas no sistema de rede), esta actualiza a componente, onde é mantido o estado das interacções via rede do(s) processo(s) alvo, com as alterações efectuadas pelo



Figura 3.4: Arquitectura do MRoP

processo. Existe ainda um sistema para controlo/configuração, destinado a configurar este sistema, assim como a obter informações sobre o estado da monitorização.

O MRoP (Monitorização de Rede orientada ao Processo), captura os pacotes de rede de um processo, sem que o utilizador necessite exista um conhecimento prévio sobre o(s) protocolo(s) ou portas utilizadas. A utilização de um sistema de instrumentação do núcleo foi necessária apenas para monitorizar as chamadas envolvendo *sockets*, e identificar o processo responsável, permitindo desta forma obter e manter, permanentemente actualizada, a informação relativa ao estado do processo alvo. De modo a minimizar a redução de desempenho, todo o sistema foi desenvolvido no núcleo do *Linux*, sem alterações nas interfaces já existentes. Assim, ferramentas que façam uso da biblioteca *PCap*, como o programa *tcpdump* ou as suas variantes, podem beneficiar desta extensão sem qualquer alteração e sem impacto relevante no seu desempenho.

Este módulo do núcleo foi desenvolvido com base nos quatro componentes que em seguida serão apresentados:

### 3.3.1 Instrumentação das chamadas ao sistema de rede

A principal vantagem deste sistema, assenta no pressuposto de que todas as interações desencadeadas por um processo com o exterior são detectadas. Para tal, foi necessário recorrer à monitorização das chamadas ao sistema de rede ao nível do núcleo, para obter as interações dos processos, com as interfaces de rede, de um modo não



intrusivo, possibilitando assim reduzir as cópias de dados e as trocas de contexto. Fazendo uso do sistema de monitorização *KProbes*, foi possível realizar a monitorização a um limitado conjunto de chamadas ao sistema, nomeadamente: *sendto*, *recvfrom*, *bind*, *accept*, *connect* e *close*. Ao efectuar esta monitorização é possível obter os portos, endereços e protocolos utilizados, de modo a detectar todas as modificações ao estado dos *sockets* do processo. Com os dados relevantes obtidos desta monitorização, mantem-se permanentemente actualizado o estado do(s) processo(s) alvo relativamente aos portos, endereços e protocolos, utilizados. O filtro de pacotes consultando este estado, pode decidir quais os pacotes relevantes a capturar, permitindo uma filtragem dinâmica orientada para o processo que é alvo de monitorização or parte do utilizador.

### 3.3.2 Estado do processo

O estado dos portos dos protocolos *TCP* e *UDP* em uso no processo alvo, é mantido num repositório de dados e permanentemente actualizado pelo componente anteriormente referido em 3.3.1. Esta deverá permitir, para cada pacote, verificar se o respectivo porto se refere a um *socket* do processo alvo.

A árvore *Red and Black* já disponível no núcleo do sistema, foi a estrutura de dados escolhida para criar o repositório pretendido. O conteúdo de cada folha da árvore é uma estrutura com duas listas de elementos, contendo cada uma endereços *IP* utilizados pela aplicação, sendo a chave de indexação das folhas, o número do porto. Desta forma, a árvore poderá conter no máximo 65535 elementos, por ser este o número máximo de portos em utilização por um endereço *IP*. No pior caso, a procura de um porto na árvore necessitará de efectuar dezasseis iterações ( $\log_2 65536 = 16$ ).

O uso deste tipo de estrutura, permite obter um bom compromisso entre o tempo de acesso aos dados e a quantidade de memória utilizada.

### 3.3.3 Filtro de pacotes

A função de filtragem implementada neste sistema assenta no estado do processo alvo, mantido pelos módulos anteriormente descritos. Através da extensão do *LSF* com um *hook*, este quando ligado, invoca esta filtragem que devolve uma resposta ao *LSF*, informando-o se deve ou não analisar o pacote em causa. Caso seja indica que este pertence ao processo alvo, e irá ser sujeito a um processo de avaliação, baseado nas restantes regras de filtragem. Caso contrário, este pacote será logo passado à aplicação, não sendo sequer analisado pelo restante sistema *LSF*. Mantêm-se assim, a compatibilidade e os benefícios da utilização do *Linux Socket Filter*. Quando não existe uma ligação (*hook*) activa, assiste-se a um ligeiro decréscimo no desempenho na utilização

da filtragem estática. Tal facto deve-se apenas à necessidade de constatar se a ligação (*hook*) está ou não activa.

Como se verifica, este sistema interage com o filtro estático do *LSF*, efectuando uma conjunção entre o filtro definido pelo utilizador e a captura do tráfego da aplicação a ser monitorizada.

### 3.3.4 Controlo e Informação

Para facilmente controlar e configurar o sistema desenvolvido, foi definida uma interface baseada em ficheiros virtuais, numa directoria do (*DebugFS*). Esta interface que contém ficheiros, de modo a indicar ao *MROp* quais os identificadores do(s) processo(s) a monitorizar, permite também ao administrador obter informações estatísticas da monitorização, bem como controlar o repositório de dados do estado do processo. É através desta interface, que os actuais sistemas de monitorização de rede, podem usufruir da funcionalidade disponível através do *MROp*.

Estes ficheiros, com permissões apenas acessíveis ao utilizador *root*, impedem o acesso por parte dos restantes utilizadores da máquina ao sistema de monitorização.

## 3.4 Conclusão

A arquitectura de rede do *Linux* é bastante complexa, em que a recepção e transmissão de pacotes, é realizada em vários contextos de execução.

# 4

## Implementação do sistema proposto

O principal objectivo desta dissertação consiste no desenvolvimento de um módulo do núcleo que consiga estender as funcionalidades do *LSF*, de forma a introduzir a funcionalidade de monitorização do tráfego realizado por um processo, tirando partido da instrumentação de código do núcleo. O facto desta instrumentação ser efectuada no núcleo permite que qualquer aplicação possa ser monitorizada, sem que o seu código tenha de ser alterado. Esta extensão ao *LSF* também pode ser utilizada por qualquer ferramenta de monitorização baseada no *LSF*. As modificações necessárias à extensão das funcionalidades do *LSF* no núcleo estão confinadas ao ficheiro *filter.c*, presente no directório *net/core* do código do *Linux*.

### 4.1 MRoP e a sua implementação

A implementação do *MRoP* teve em consideração o desenvolvimento de código que implicasse ligeiras alterações ao código do núcleo e, simultaneamente, tirasse partido de *APIs* internas. Por outro lado, é transparente à implementação da biblioteca *PCap*, podendo ser integrado na mesma. A implementação está contida num módulo do núcleo, de modo a ser carregada e libertada do mesmo, pelo administrador. A modularização das diversas componentes, permite um desenvolvimento autónomo de cada subcomponente.

No capítulo 2 apresentaram-se, a título exemplificativo (secção 2.5), alguns sistemas que permitem efectuar monitorização ou filtragem de pacotes, com a indicação de

um processo. O *MRoP*, embora utilize o sistema de instrumentação do núcleo *KProbes*, apenas o aplica para instrumentar as interações com o sistema de rede, diferenciando-se das soluções apresentadas na secção 2.7, o que lhe permite encontrar-se totalmente implementado no núcleo, com reduzida utilização de memória e perturbação do sistema.

Assim quando uma aplicação efectua uma chamada ao sistema (*connect*, *accept*, *bind*, *sendto*, *recvfrom*), o *handler* da função instrumentada é executado. Na execução deste *handler*, é obtido o identificador do processo que efectuou a chamada ao sistema e comparado com os identificadores, *pid*, *ppid* e *tgid* internos ao *MRoP*, caso um destes seja igual, é obtido o canal (através dos parâmetros passados à chamada ao sistema) e adquiridos os dados sobre o *socket* (porto, endereço e protocolo), para serem adicionados ao repositório. Deste modo quando executar o novo sistema de filtragem irá verificar se os metadados do pacote (endereço, porto, protocolo), recebido ou enviado, existem no repositório *Estado do processo* e, caso existam, indica à função de filtragem do *LSF* para avaliar os filtros normais e, eventualmente capturar o pacote, sendo posteriormente passado para o monitor em nível utilizador.

Nas subsecções seguintes, irão apresentar-se os quatro componentes do *MRoP* conforme figura 4.1.



Figura 4.1: Arquitectura geral do MRoP

## 4.2 Instrumentação de funções do núcleo

A metodologia aplicada à resolução do problema de desempenho, baseia-se no desenvolvimento de uma componente para efectuar uma análise aos canais de comunicação de rede, utilizados por um processo, que insira a informação necessária no repositório. Assim, logo que um pacote chega ao sistema de monitorização, o *LSF* tira partido do repositório para decidir a captura do pacote.

De entre os sistemas analisados, o *KProbes* foi aquele que permitiu obter menor sobrecarga, apesar do seu carácter dinâmico. Os restantes sistemas contêm componentes de registo que, para a realização deste mecanismo, apresentam uma sobrecarga desnecessária, afectando negativamente o desempenho.

Para além do sistema de instrumentação utilizado, foi igualmente considerado o número de funções a instrumentar. A sobrecarga total exercida pela instrumentação de funções do núcleo, tem em consideração não só o número de funções que são instrumentadas, como também o número de vezes que estas são executadas.

O conhecimento adquirido com a análise efectuada referida no capítulo 3, sobre a *Arquitectura de rede em Linux*, permitiu identificar as funções a instrumentar.

O *MROp*, foi desenvolvido para monitorizar a utilização de canais da família *INET*, tendo como particularidade, a utilização de canais baseados nos protocolos *TCP* e *UDP*, e instrumentar um número reduzido de funções do núcleo, pertencentes ao sub-sistema de rede.

O *TCP* e o *UDP* apresentam algumas funções em comum com protocolos de outras famílias, principalmente ao nível das chamadas ao sistema, onde o nível de abstracção sobre estes protocolos é elevado. No entanto, com o objectivo de diminuir o número de funções a instrumentar, optou-se mesmo assim por instrumentar as chamadas ao sistema, mesmo que algumas vezes estas não digam respeito à família *INET*.

As chamadas ao sistema instrumentadas correspondem a: *sendto*, *recvfrom*, *connect*, *bind*, *accept* e *close*. As análises inicialmente efectuadas demonstraram que apenas a chamada ao sistema *close*, era demasiadas vezes executada, penalizando o desempenho global do sistema. Esta situação radica no facto da chamada ao sistema *close*, ser utilizada extensivamente para fechar canais, independentemente destes serem ficheiros, *sockets*, *pipes*, etc. Para contornar esta dificuldade, foi necessário encontrar uma função que lidasse exclusivamente com o fecho de *sockets*, de modo a reduzir a sobrecarga imposta pela instrumentação.

### 4.2.1 Filtro de processos

O *KProbes*, é um sistema de instrumentação do núcleo que não distingue entre que funções, não *inline*, está a efectuar a instrumentação. Não existindo suporte no *KProbes* para filtrar o processo, que efectuou a chamada ao sistema, foi necessário desenvolver um modo que permitisse reduzir a sobrecarga, quando a chamada é de um processo, que não o desejado.

No intuito de ultrapassar esta dificuldade, poder-se-ia ter recorrido à criação de um repositório com a informação sobre os identificadores dos processos a monitorizar, sendo necessário, uma vez mais, uma estrutura de suporte a este repositório, bem como funcionalidades de adição, remoção, actualização e consulta. Este repositório teria de conter a estrutura genealógica do processo a monitorizar, assim como uma componente que actualizasse essa informação. A actualização desta estrutura poderia ser efectuada através da instrumentação da chamada ao sistema *fork* ou *clone*. Sempre que fossem invocadas as funções já referidas, seria efectuada uma consulta ao repositório e, a partir da informação obtida, este seria ou não, actualizado. Esta possibilidade, ao contemplar a remoção de dados do repositório, necessitaria também de instrumentar a função de término de processos. Considerando que a actualização dinâmica deste repositório, sem aplicar alterações no código do núcleo, afectaria negativamente o desempenho do sistema na sua totalidade, optou-se por excluir esta alternativa. Face a esta situação, decidiu-se efectuar uma análise aos campos da estrutura (*task\_struct*), o que permitiu compreender o modo como os identificadores dos processos se relacionam com os identificadores dos membros da árvore genealógica do processo. Desta análise conclui-se que, os identificadores *pid*, *tid* e *ppid* permitem, na grande maioria das aplicações, identificar toda a árvore genealógica.

## 4.3 Estado dos *sockets* do processo

De modo a manter a informação relativa aos endereços e portos em utilização por uma aplicação, sem requerer a consulta sobre os canais de rede, foi necessário criar um repositório de dados, que contivesse informações relevantes para as decisões do filtro de captura. Neste repositório, existe a necessidade de ter funções de inserção, remoção e consulta, sendo que qualquer uma destas, deverá ser efectuada com celeridade. A estrutura de dados necessária para suportar o *Estado dos sockets do processo*, será uma estrutura que tenha, no máximo, uma complexidade temporal de  $O(\log n)$  sobre as pesquisas, dado que estas serão muito superiores às inserções e remoções.

Assim, as estruturas de dados com suporte no núcleo, que permitem a criação de um repositório de dados, como o requerido, são:

**BitMap** - O núcleo do sistema possui suporte para o tratamento de mapas de *bits*, permitindo representar cada porto usado pela aplicação por um *bit*. O recurso a um mapa de *bits* permite, de um modo bastante rápido e com uma reduzida utilização de memória, determinar se um porto está em utilização. Embora este processo seja extremamente rápido, carece de modularidade, na medida em que apenas controla os portos, não dispondo de suporte para protocolos ou múltiplos endereços de rede.

**Listas** - No núcleo existe uma implementação bastante eficiente da estrutura de dados *lista* (*list*), contendo apenas dois apontadores, destinando-se um ao elemento que o precede e outro ao que se lhe segue.

Embora não seja necessário definir uma lista com o número máximo de portos, dado que estes podem ser adicionados dinamicamente, a complexidade temporal de pesquisa, no pior caso, é de  $O(n)$  (traduzindo-se num mau indicador de desempenho para o estudo pretendido nesta dissertação). No entanto, quando o número de elementos não é elevado, a utilização de uma lista apresenta-se como uma possível solução.

#### **Árvore Balanceada** - *Red-black Tree*

No núcleo existe uma implementação parcial de uma árvore *Red-black* genérica, a fim de permitir o acesso aos dados através de chaves, ou seja, trata-se de uma estrutura associativa. A árvore *Red-black* é semi-balanceada, isto é, a diferença de alturas entre o ramo mais profundo e o mais curto é de apenas de 1 nível. Esta propriedade é mantida através do rebalanceamento da árvore em inserções e remoções, o que provoca um custo na sua utilização. Contudo, no caso esperado, existe maior número de consultas do que inserções e remoções, fazendo com que o custo associado ao rebalanciamento da árvore seja amortizado. De modo a tirar partido da utilização desta estrutura de dados, é necessário definir três funções da manipulação da árvore (inserção, remoção e consulta). O suporte disponibilizado pelo núcleo, apenas permite manusear a árvore, sendo necessário definir as funções que utilizam a chave de acesso para aceder ao conteúdo dos dados, que devido à sua especificidade, não podem ser oferecidas.

De referir igualmente que, para o objecto deste estudo, o número do porto dos protocolos (*TCP* e *UDP*), é considerado a chave mais adequada. Tendo em conta o número máximo de portos possíveis nos protocolos (*TCP* e *UDP*), a árvore poderá conter 65535 elementos, com uma altura máxima de 16, ou seja, para pesquisar um dos elementos nos extremos (máximo ou mínimo) é necessário efectuar sobre

ela, 16 iterações. Embora não constitua um requisito, é possível obter de forma ordenada todas as chaves, bem como os valores que lhe estão associados.

**Tabela de Dispersão** - No núcleo existe uma implementação de tabelas de dispersão, que efectua a dispersão e o controlo sobre as suas chaves. O controlo sobre as chaves e a forma de dispersão é efectuada pela implementação, ou seja, não existe controlo do programador sobre as chaves nem sobre a forma de dispersa-las.

No núcleo existem subsistemas que implementaram outras tabelas de dispersão, com base em *arrays* e listas, permitindo deste modo utilizar as vantagens desta estrutura de dados. Estas implementações tiram partido do conhecimento do domínio do problema que resolvem, pelo que, os *arrays* são criados com dimensões fixas, dado que não necessitam efectuar redispersão dos elementos nela contidos.

Como existe a necessidade de uma estrutura que se adapte ao comportamento dinâmico das interacções das aplicações com as interfaces de rede, é necessário um estudo aprofundado sobre estatísticas do número de portos utilizados pelas aplicações e, caso este estudo seja realizado é necessário continuar a efectuar uma redispersão dos elementos, a não ser que, seja utilizado um *array* com 65535 posições, valor este que representa o valor máximo de portos possíveis através dos protocolos *TCP* e *UDP*, desaproveitando assim memória do sistema.

Após terem sido verificadas quais as estruturas de dados com suporte no núcleo, foi necessário escolher qual a melhor escolha para a necessidade do *MRoP*. O *BitMap* apesar de ser uma estrutura de dados em que os acessos são bastante rápidos e com uma utilização de memória reduzida, carece de modularidade para a utilização com múltiplos endereços e protocolos. Apesar de, na pior situação estarem em utilização apenas 1024 portos, seria necessário manter em memória todo o mapa referente aos 65 535. Relativamente à estrutura *lista*, a análise dos portos no pior caso, aquele em que o pacote em análise não pertence ao processo alvo, é bastante prejudicial, pois é necessário verificar que não se encontra na lista, o que obriga a percorrer todos os elementos da lista. Esta situação pode ser minimizada se a lista tiver os elementos ordenados pelo número do porto, o que obriga a inserções ordenadas. A *tabela de dispersão* seria a melhor escolha se existisse o controlo das chaves, por parte do programador, assim como tal não é possível e, como a opção de implementação de uma tabela de dispersão, necessitaria de uma verificação de correcção e desempenho, tal opção foi rejeitada. Apesar dos processos monitorizados apresentarem elevado dinamismo nas interacções das comunicações, o maior número de operações sobre o repositório concentra-se ao nível da consulta. As restantes operações (inserção e remoção) repartem entre si igual



número. Assim, é esperado que para cada inserção exista uma remoção. No final da monitorização de um processo, a componente *Estado dos Sockets* deverá apresentar um número de elementos idêntico ao que antecedeu a monitorização. Deste modo a árvore balanceada *Red-black*, é a melhor opção entre as disponíveis no núcleo, pois a procura no pior caso é melhor que na *lista*, uma vez que tem os elementos ordenados e permite uma utilização de memória reduzida e com possibilidade de extensão dos seus elementos.

Assim o ciclo de desenvolvimento do *MROP*, foi efectuado mais rapidamente, tendo em consideração a confiança que merece a validação da estrutura de dados *Red-black tree*, em utilização no núcleo do sistema *Linux* e sujeita a uma análise extensiva ao longo dos anos.

#### 4.3.1 Estrutura utilizada

Os elementos do repositório criado, através de uma árvore *Red and Black*, têm uma estrutura bem definida, contendo obrigatoriamente um *rb\_node*, para possibilitar a manipulação da árvore e um outro elemento, de carácter comparativo, utilizado como chave. Além dos referidos, esta estrutura contempla outros elementos que seguidamente se descrevem:

PortInfo

|                         |
|-------------------------|
| rb_node node            |
| u16 port                |
| local_address_list *udp |
| local_address list *tcp |
| int tcp_list_counter    |
| int udp_list_counter    |

Figura 4.2: Elemento da árvore

local\_address\_list

|                |
|----------------|
| list_head list |
| u32 address    |
| int counter    |

Figura 4.3: Lista de endereços

A figura 4.2 apresenta a disposição dos elementos da estrutura *PortInfo*, sendo que as listas de endereços *IP* das interfaces de rede, são adicionadas através da estrutura apresentada na figura 4.3. Os elementos do repositório, correspondem a instâncias da

estrutura *PortInfo*, os quais são adicionados através dos *handlers* das funções instrumentadas, tal como no esquema exemplificado na figura 4.4.

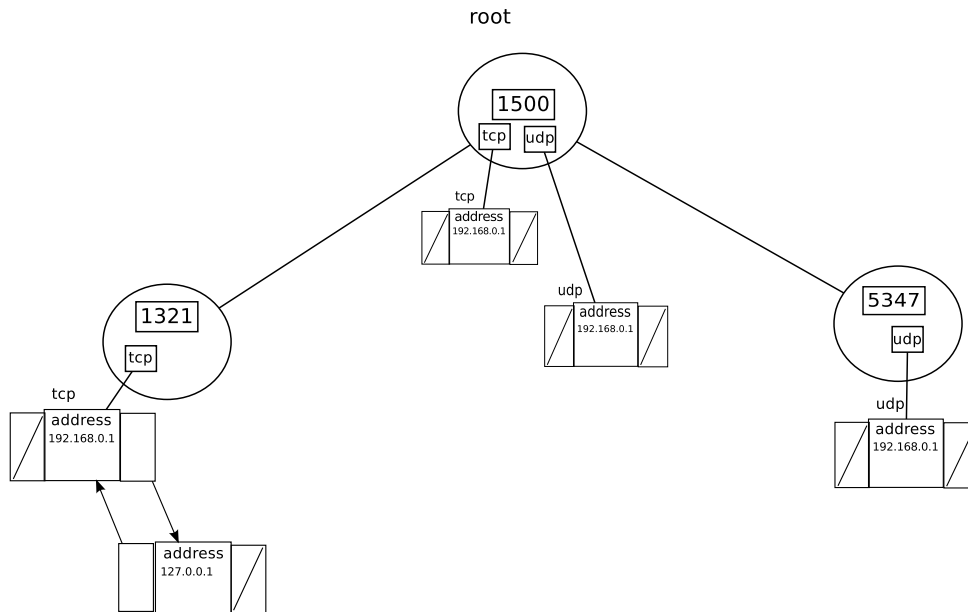


Figura 4.4: exemplo do repositório *Estado do Processo*, com 4 *sockets*

Apesar dos processos monitorizados apresentarem elevado dinamismo nas interações das comunicações, o maior número de operações sobre o repositório concentra-se ao nível da consulta. As restantes operações (inserção e remoção) repartem entre si igual número. Assim, é esperado que para cada inserção exista uma remoção. No final da monitorização de um processo, a componente *Estado dos Sockets* deverá apresentar um número de elementos idêntico ao que antecedeu a monitorização.

### 4.3.2 API de comunicação interna do MRoP

Com o objectivo de efectuar inserções, consultas e remoções dos dados do repositório, foi desenvolvida uma API interna ao MRoP, que permite validar os parâmetros passados às funções do repositório de dados. Através desta API foi possível realizar a separação das componentes do MRoP beneficiando, deste modo, a modularidade do código. Esta API permite aos *handlers* das funções instrumentadas, efectuar as operações de inserção e remoção sobre a componente *Estado dos Sockets*. Relativamente à operação de consulta, esta é particularmente importante, na medida em que é usada na filtragem de pacotes das interfaces de rede.

A API, não obstante reduzir o desempenho, devido à necessidade de chamar os métodos específicos ao repositório, permite a substituição deste, sem que se verifique

alterações do código.

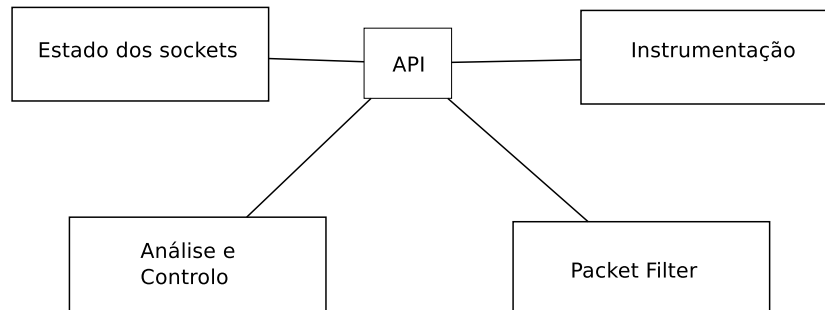


Figura 4.5: API interna do MRoP

falar sobre  
imagem

## 4.4 Filtro de pacotes, extensão ao LSF

No sistema de monitorização de rede, um dos componentes corresponde a uma função que serve de máquina virtual às instruções do *LSF*. Esta função itera sobre o filtro, executando instrução a instrução, sobre o pacote (recebido ou enviado), até ao momento em que identifica uma das instruções de retorno (*BPF\_RET* ou *BPF\_RET\_A*). Consoante o valor retornado nestas instruções, o pacote analisado é, ou não, capturado. Caso seja capturado, é efectuado um *clone* do pacote e colocado num *ring buffer*, partilhado com a aplicação monitora de rede em nível utilizador. Caso o valor retornado não corresponda a uma captura, a computação sobre esse pacote termina, reduzindo a sobrecarga da monitorização. Assim, o facto de se tirar partido da utilização de um filtro, que identifique de forma célere a rejeição de um pacote, diminui consideravelmente a sobrecarga imposta ao sistema por parte da monitorização.

Face aos benefícios obtidos pela utilização do filtro, considera-se que o sistema de instrumentação do núcleo (*KProbes*), poderia ser utilizado para invocar o novo sistema de filtragem criado e modificar o valor de retorno, caso se tornasse necessário. Todavia, face ao número de vezes que a função de filtragem é invocada (uma para cada pacote, recebido ou enviado), a sobrecarga da utilização do *KProbes* é de todo desaconselhável.

Foi necessário modificar o código do *Linux* de modo a inserir um *hook*, ou seja, um apontador para uma função. Esta função será invocada quando o filtro estático for avaliado para captura, permitindo que a função de filtragem do *MRoP* analise o pacote com base no estado do processo, possibilitando efectuar uma conjunção entre o filtro

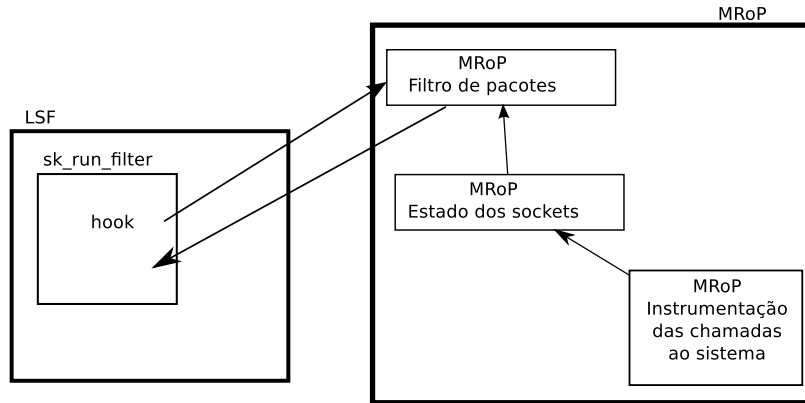


Figura 4.6: Execução da nova filtragem de pacotes pelo LSF

estático definido pelo utilizador do *LSF/PCAP* e a filtragem dinâmica efectuada pelo *MRoP*, como pode ser observado na figura 4.6.

Além da definição de um *hook* para activação e desactivação deste novo sistema, foi efectuada uma alteração ao código do *Linux* na função *sk\_run\_filter*, que se traduziu no retorno da decisão conjunta do filtro estático com o dinâmico, ambas realizadas no ficheiro *filter.c*. Esta alteração permitiu adicionar uma nova funcionalidade, com uma mínima sobrecarga no sistema de monitorização, independentemente da mesma estar ou não activa.

## 4.5 Informação de análise e controlo

O *MRoP* foi desenhado e implementado de modo a ser relativamente autónomo, apenas necessitando da configuração de alguns parâmetros, essenciais ao seu funcionamento.

Com o propósito de obter informações sobre o estado da computação dos diversos componentes da ferramenta e de invocar a monitorização, procedeu-se à criação de alguns ficheiros no *DebugFs*. Assim, no que se refere à componente *Informação de análise e controlo*, esta fica responsável pelos diversos aspectos da instrumentação e do repositório.

### 4.5.1 Informação de controlo

Os ficheiros de controlo criados foram: *pid*, *ppid*, *tgid* e *option*. Estes ficheiros, à excepção do último, podem ser lidos e escritos pelo administrador e, se já tiverem sofrido uma escrita, contêm o identificador referente ao processo, ou processos, a monitorizar. Caso não tenham sido alterados, o ficheiro conterá o valor por omissão, neste caso zero (0). O ficheiro *option*, tem apenas a permissão de escrita e, dependendo do valor escrito,

pode activar a procura de portos no processo indicado em *pid*, apagar todos os dados do repositório e activar, ou desactivar, o filtro dinâmico.

#### 4.5.2 Informação de análise

A informação de análise apenas é adicionada, caso o seu suporte seja activado na compilação. Caso esteja activa é possível obter estatísticas sobre os diferentes componentes internos ao *MROp*.

Os ficheiros de análise apenas estão disponíveis para leitura, devolvendo os valores presentes nos contadores internos do *MROp*. Relativamente à componente de filtragem dinâmica, os contadores declarados contêm o número de pacotes que foram analisados para captura, bem como quantos destes foram transferidos para o monitor em nível utilizador. No que se refere à monitorização do processo, existe também um ficheiro que devolve, em relação a cada uma das funções instrumentadas, o número de execuções do *handler* e, quantas destas pertenciam ao processo alvo. Relativamente ao repositório de dados, foi também criado um ficheiro com estatísticas, que incidiram sobre o número de portos em utilização e, para cada porto, a indicação de estar em utilização, através do protocolo *TCP*, *UDP* ou em ambos, e por qual ou quais endereço(s) de rede.

Estes valores serviram para depuração do sistema e para aferir que os pressupostos que levaram à implementação apresentada se verificavam.



# 5

## Avaliação

Antes de se utilizar um novo sistema, é conveniente proceder-se a testes alargados de verificação, com o objectivo de tomar conhecimento do seu correcto funcionamento e das sobrecargas introduzidas, para que possa ser utilizado como uma mais-valia. O mecanismo implementado (*MROp*) foi avaliado funcionalmente, através de diversos testes aos dados de entrada nos diversos componentes que o constituem. Como existem diversas fontes de dados para o *MROp* (dados de controlo, dados do processo alvo de monitorização, fluxos de rede, etc.), foram criados vários testes de modo a que todas estas fontes de dados fossem analisadas. Para além destas fontes directas de dados foi, igualmente, verificada a correcção de que todos os fluxos de dados obtidos, através da monitorização de rede, são exclusivos do processo alvo. Assim todos estes testes e avaliações foram executados com sucesso e serão apresentados na secção 5.1.

Como um dos principais objectivos desta dissertação é criar um mecanismo com melhor desempenho que os anteriores, apresentados na secção 2.7, foi assim necessário verificar se este tinha sido atingido. Para o efeito foram efectuados testes que serão apresentados na secção 5.2, incidindo sobre o desempenho global na transferência de 1GB de dados, através de protocolos conhecidos, bem como através de outros testes que visam o mecanismo de instrumentação e a estrutura de dados escolhida para manter o estado do processo alvo.

Por último, serão apresentadas na secção 5.3 algumas conclusões sobre as análises efectuadas ao *MROp*.

## 5.1 Avaliação Funcional

A análise funcional ao *MRoP* teve várias vertentes, uma vez que este mecanismo recebe dados de diferentes fontes. Assim os testes visaram a verificação dos dados inseridos pelo administrador, pelos dados monitorizados do processo alvo e os recolhidos das estruturas do núcleo. Foi necessário garantir a inexistência de falhas nos mecanismo de entrada de dados no *MRoP*, porque como este está a executar no núcleo tem acesso a todo o sistema, logo qualquer problema na validação de dados poderá comprometer o sistema quer ao nível de segurança quer ao nível de disponibilidade.

### 5.1.1 Teste ao componente de controlo

Como o *MRoP* foi desenvolvido para ser um módulo para o núcleo, foi necessário adicioná-lo a este, através do programa *insmod*. Após a inclusão do *MRoP* ao núcleo, para que este funcione é necessário que um utilizador com permissões de administrador (*root*), indique no ficheiro *option*, criado no *DebugFs* para o controlo do *MRoP*, que deseja dar início à monitorização. Dado que o *MRoP* é parte do núcleo, é imprescindível garantir que a fiabilidade dos dados que lhe são transmitidos, não comprometem a segurança nem a disponibilidade do sistema, pelo que as funções que recebem dados oriundos do sistema de controlo, efectuam verificações antes de os transmitirem às restantes componentes do *MRoP*. Como os dados de controlo são cadeias de caracteres, a função de verificação limita o comprimento máximo destes, baseado no valor máximo expectável para a utilização do ficheiro. Assim cadeias de caracteres superiores a um determinado comprimento são truncadas, não existindo a possibilidade de explorar falhas de *overflow*. O valor zero (0) serviu como inicializador, sendo que valores superiores a este, são indicadores de que existe uma opção activa no *MRoP*. Para além destas análise, foi também verificado que as permissões dos ficheiros são as estritamente necessárias e, restringem a sua disponibilidade ao utilizador *root*.

### 5.1.2 Obtenção do estado dos canais do processo alvo

A análise funcional de detecção dos protocolos, portos e endereços locais, foi efectuada recorrendo à criação de dois conjuntos de programas *cliente/servidor*. Para o primeiro conjunto foi criado um servidor e um cliente, ambos utilizando o protocolo *tcp*, em que o programa servidor esperava conexões numa porta e endereço pré-definidos. Como o *MRoP* instrumenta as chamadas ao sistema *connect*, *accept*, *bind*, *recvfrom*, *sendto* e a função *sock\_close*, é possível capturar todas as operações relativamente às comunicações de rede utilizando os protocolos *tcp* e *udp*. Nas chamadas ao sistema *bind* e *connect*, um dos



seus argumentos é uma estrutura *sockaddr* que contém informações sobre o endereço e porto, sendo que na função *connect* estes são relativos ao destino, e na função *bind* são relativos à origem. Assim os dados anteriormente mencionados, são de extrema utilidade para a construção do repositório do *MROp*, de modo a capturar apenas o tráfego respeitante ao processo alvo. Como os dados são provenientes do processo alvo, não é possível garantir a sua fiabilidade, assim caso a função instrumentada retorne com um valor de erro, os dados anteriormente adicionados (porto, endereço e protocolo) são retirados do repositório, ficando novamente num estado consistente com o processo alvo. De modo a verificar que os dados que se encontram no repositório são completos e fiáveis, é efectuada a leitura de todos os valores do repositório e apresentados através da função *printk* para o terminal do núcleo, ou são apresentados no ficheiro criado para o efeito no sistema de ficheiros virtual *DebugFs*, verificando-se a sua exactidão com o que a aplicação *netstat* declara para o processo alvo.

Quando o processo alvo já se encontra em execução e se inicia a monitorização, caso os *sockets* estejam em utilização contêm os dados relativos aos protocolos, portos e endereços dos *sockets* e, por isso consideram-se correctos e válidos, sendo automaticamente adicionados ao repositório.

### 5.1.3 Avaliação de monitorização de rede

Foi efectuada uma verificação à correcção dos dados transferidos entre dois processos, um cliente e outro servidor. Estes testes foram apenas efectuados para protocolos assentes sobre *tcp*, pois nestes existe a certeza que, devido ao sistema de controlo de comunicação não existem falhas, o que permite analisar a transferência do início ao fim.

A execução do teste consistiu na transferência de um ficheiro entre duas máquinas, ligadas por um *switch* com portas a 100 *Mbit/s*, através do protocolo aplicacional *ftp*, enquanto existiam outros fluxos de rede de outros processos. Foi escolhido este protocolo pois, apesar de ser necessário conhecer *a priori* a porta de comunicação com o servidor, a transmissão de dados é efectuada noutra porta negociada dinamicamente, permitindo demonstrar também o potencial do *MROp*. Todas as comunicações remotas referentes a esta transferência foram monitorizadas através do programa *tcpdump*, com o módulo do núcleo *MROp* activo e com a indicação do processo a monitorizar. Esta monitorização foi guardada através do *tcpdump* num ficheiro para posterior análise através do *Wireshark*. Utilizando o ficheiro capturado no *Wireshark*, foi possível observar que apenas os fluxos de rede da transferência existiam no ficheiro. Após esta verificação, foi efectuada a recuperação do ficheiro transmitido através da agregação dos dados presentes nos pacotes, exceptuando os cabeçalhos. Esta recuperação foi

guardada num ficheiro temporário, de modo a poderem ser aplicadas funções de síntese (*md5* e *sha1*), com o objectivo de verificar se o conteúdo dos pacotes recuperado era exactamente igual ao do ficheiro original, e à sua transferência. Esta verificação foi confirmada através do retorno do mesmo valor para os três ficheiros, para cada uma das funções de síntese utilizadas.

Foi igualmente verificada a correcção das transferências sobre o protocolo *udp*, para o que se procedeu à realização de um teste utilizando o programa *iperf*, com recurso a *sockets udp*. Para a realização do referido teste recorreu-se à mesma infra-estrutura de rede, em que numa das máquinas foi executado o *iperf* em modo servidor utilizando *sockets udp*, enquanto na outra foi executado o modo cliente utilizando *sockets udp*. Foi ainda efectuada a monitorização de rede na máquina que executou o *iperf* em modo cliente, através do programa *tcpdump* com o módulo do núcleo activo e com a indicação do identificador do processo *iperf*, enquanto existiam outros fluxos de rede de outros processos. O resultado da monitorização pelo *tcpdump* foi guardado num ficheiro, que indicou que o total de *bytes* correctamente recebidos pelo *iperf* em modo servidor, foram igualmente obtidos pela monitorização. Neste ficheiro não existiam outros fluxos de dados que não os respeitantes à utilização do *iperf*.

Como foi anteriormente referido, aquando da realização dos testes existiram outros fluxos de rede, nomeadamente de tráfego *web*, e de aplicações de conversação instantânea, entre outros, de modo a comprovar a eficácia do *MRoP*.

## 5.2 Avaliação do desempenho

Tendo em vista a avaliação do desempenho, foram efectuados diversos testes com o objectivo de avaliar a sobrecarga gerada pela introdução do *MRoP*. Estes testes basearam-se na recepção ou transmissão de 1 GigaByte de dados, utilizando diferentes programas e protocolos, entre duas máquinas ligadas directamente. Ambas as máquinas, que se optou por designar de máquina 1 e máquina 2, procederam à transmissão/recepção de dados, utilizando cada uma, apenas, um processador activo de 2 e de 2.6 Ghz, respectivamente. As máquinas anteriormente referidas encontravam-se conectadas directamente, por interfaces de rede a 100 MBit/s, ficando uma responsável pela execução dos servidores *ftp*, *http* e *iperf*, e a outra pelos respectivos clientes. A versão do sistema de operação utilizado, em ambas as máquinas, correspondeu ao 2.6.39, sendo que na máquina 1 foram introduzidas as modificações, para incluir o *hook* do *MRoP* e as suas funções auxiliares, enquanto na máquina 2 se executou o sistema original.

### 5.2.1 Desempenho do *MRoP*

Na execução destes testes, foram efectuadas dez iterações, isto é, cada teste foi executado dez vezes, para cada experiência considerada, de modo a obter um valor médio e um desvio padrão considerado aceitável. Os testes efectuados, em particular os primeiros, não mostraram vantagem em ter o mecanismo *MRoP* activo, com vista a medir a sobrecarga do *MRoP*. Os resultados obtidos constam nas tabelas 5.1 e 5.2.

Os primeiros quatro testes foram efectuados utilizando apenas uma conexão ao servidor, enquanto o 5º e o 6º testes utilizaram mais uma comunicação, de modo a aumentar o peso sobre o processador e o número de pacotes a circular entre as máquinas. Desta forma, foi possível identificar a sobrecarga exercida enquanto o *tcpdump* executava e capturava todos os pacotes ou apenas um subconjunto destes, ou seja, os pacotes relativos aos processos alvo no novo sistema.

Na tabela 5.1, a coluna "Original" corresponde aos valores resultantes dos tempos médios das execuções das transferências na ausência de monitorização. Na mesma tabela e na coluna "Com *TcpDump*", é apresentada a média dos tempos de transferência com a captura total do tráfego utilizando a biblioteca *PCap/LSF* original, enquanto que a coluna identificada com "Com *TcpDump* e *MRoP*" regista a média dos tempos para a transferência com captura pelo *tcpdump* e o módulo *MRoP* desenvolvido no núcleo, de forma a capturar, apenas, o tráfego da transferência do processo alvo. Nos primeiros quatro testes é possível verificar que a utilização do *MRoP*, aumentou de forma irrelevante o tempo de execução (figura 5.1).

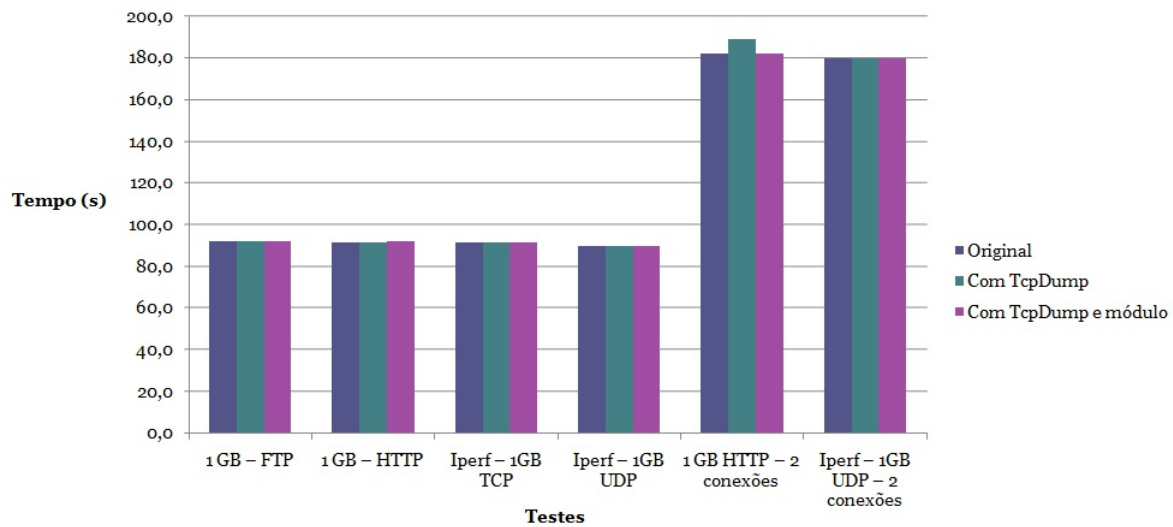


Figura 5.1: Testes de desempenho efectuados ao MRoP

É igualmente possível observar que no 1º e 3º testes, aquando da utilização do *tcpdump*, a execução sem o *MRoP*, mostrou-se ligeiramente mais rápida, como se pode verificar na tabela 5.1 e 5.2. Esta situação deve-se ao facto de, quando a máquina

Tabela 5.1: Tempos médios em segundos (s)

| Teste                                   | Original | Com TcpDump | Com TcpDump e MRoP |
|---|----------|-------------|--------------------|
| 1GB - FTP <sup>1</sup>                  | 91.8508  | 91.8500     | 91.8854            |
| 1GB - HTTP <sup>2</sup>                 | 91.6391  | 91.6472     | 91.6674            |
| IPerf - 1GB TCP <sup>3</sup>            | 91.3790  | 91.2535     | 91.2672            |
| IPerf - 1GB UDP <sup>4</sup>            | 89.7975  | 89.8007     | 89.8464            |
| 1GB HTTP - 2 conexões <sup>5</sup>      | 182.1573 | 188.7156    | 182.0161           |
| IPerf - 1GB UDP 2 conexões <sup>6</sup> | 179.4930 | 179.6280    | 179.6369           |

se encontra em sobrecarga, leva ao aumento do tamanho médio dos pacotes, reduzindo o seu número e o volume de dados transferidos, em virtude da diminuição dos seus cabeçalhos. Este comportamento já tinha sido detectado numa dissertação anterior [Far09].

Nos 5º e 6º testes, como o tráfego na interface é duplicado e o *tcpdump* tem que capturar todos os pacotes, é possível evidenciar a sobrecarga exercida por estas cópias de dados e consequentes transferências (para nível utilizador) face ao novo sistema onde apenas captura um fluxo de dados.

Na tabela 5.2 e na figura 5.2 é possível observar que, para o teste 5, a sobrecarga do *tcpdump* atinge os 3.6% face ao original, enquanto que a sobrecarga do *tcpdump* com o *MROp*, permitiu uma ligeira melhoria face ao original (-0.0775%). Conclui-se, portanto, que quando o fluxo de dados que não pretendemos capturar aumenta consideravelmente, torna-se mais vantajoso utilizar o *MROp*, do que capturar todos os pacotes. Este modo de captura minimiza a sobrecarga, capturando apenas os dados relevantes, evitando-se a identificação e filtragem dos pacotes pertencentes ao processo alvo em nível utilizador, tendo como consequência uma sobrecarga adicional.

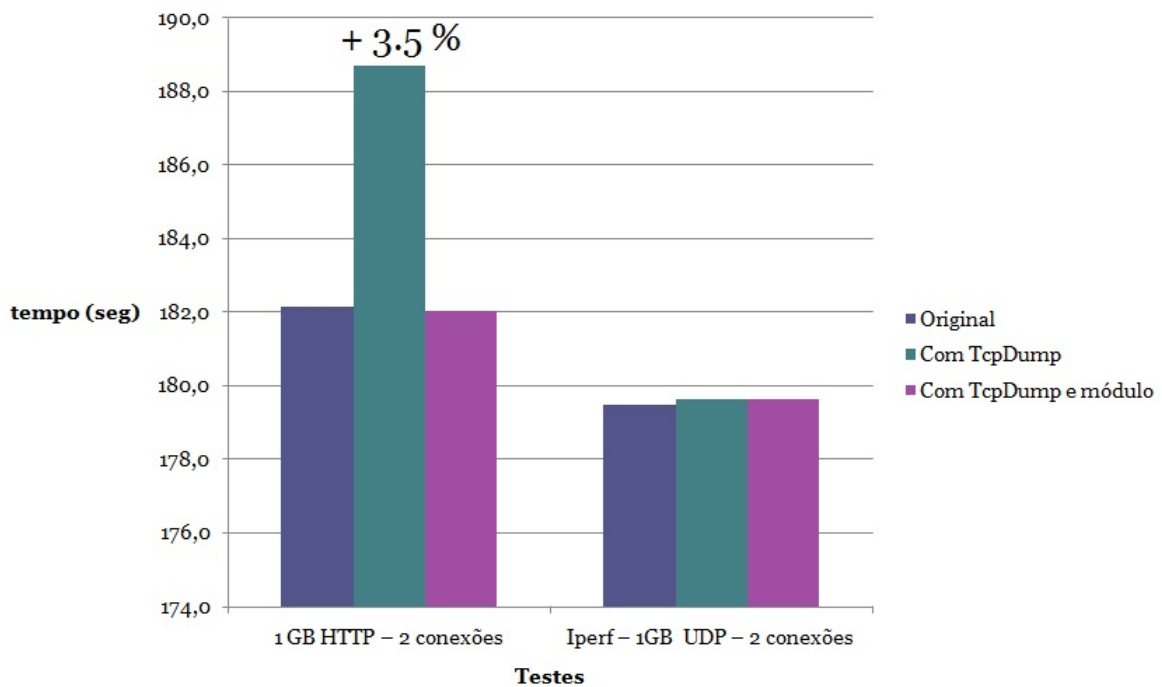


Figura 5.2: Sobrecarga nos testes 5 e 6

Tabela 5.2: Sobrecarga das transferências (valores em percentagem)

| Teste                                   | TcpDump | TcpDump com MROp |
|---|---------|------------------|
| 1GB - FTP <sup>1</sup>                  | -0.0009 | 0.0377           |
| 1GB - HTTP <sup>2</sup>                 | 0.0088  | 0.0309           |
| IPerf - 1GB TCP <sup>3</sup>            | -0.1373 | -0.1223          |
| IPerf - 1GB UDP <sup>4</sup>            | 0.0036  | 0.0545           |
| 1GB HTTP - 2 conexões <sup>5</sup>      | 3.6003  | -0.0775          |
| IPerf - 1GB UDP 2 conexões <sup>6</sup> | 0.0752  | 0.0802           |

Os resultados da sobrecarga do *tcpdump* com o *MROp*, em todos os casos analisados,

foram sempre inferiores a 0.15%, chegando mesmo a atingir 0.0309%. Se compararmos este resultado com os obtidos em [Far09], assiste-se a uma substancial melhoria. Efectuando a comparação deste resultado com o obtido no 2º teste, da transferência por *HTTP*, não por ser o melhor resultado obtido, mas por o teste ser idêntico, estamos perante uma melhoria no desempenho de 64 vezes.

Como o anteriormente referido, os testes realizados apenas com o *tcpdump* capturam todo o tráfego e, como não existia a monitorização do processo alvo, não era efectuada a filtragem dos pacotes. Desta forma os valores apresentados para o *tcpdump* são meramente indicativos relativamente ao *tcpdump* com o *MROp* activo, uma vez que lhe falta a componente de filtragem dos dados, aumentando o tempo de execução e a sua sobrecarga.

### 5.2.2 Desempenho da estrutura de dados

Para além das avaliações anteriormente descritas, tornou-se essencial analisar o comportamento da estrutura de dados utilizado no componente “estado do processo”, de modo a verificar o seu desempenho. Assim para esta análise, foi elaborado um teste para determinar o desempenho da estrutura de dados, relativamente às inserções e remoções. Este teste utilizou o sistema de alta resolução de temporizadores (*HRTimer*)[GM], presente no núcleo do sistema de operação.

Para decidir qual o número de elementos a ser utilizado para este teste, foi verificado qual o valor máximo de descritores de canais que um processo pode ter. O valor foi obtido através da função *getrlimits*, que indicou que o valor máximo de descritores de canais abertos, para um processo, é de 1024 canais. Dado que este valor (1024) é o máximo de canais por processo, considerou-se um optimo valor para verificar o comportamento da estrutura de dados no pior caso, no cenário em que todos os descritores de canais são *sockets* e estão activos. Os valores obtidos através deste teste demonstram os valores máximos espectáveis, em que se assume que o processo alvo está a utilizar o número máximo de canais de rede.

Assim o teste consistiu em obter o tempo anterior e posterior à inserção dos 1024 elementos, representando outros tantos portos/endereços, afim de determinar o tempo decorrido. De igual modo, foi calculado o tempo de remoção dos referidos elementos. Os resultados obtidos estão reproduzidos na tabela 5.3.

Pode verificar-se, pela tabela 5.3, que a inserção de um elemento na árvore é inferior a 1 microsegundo, demonstrando que a estrutura utilizada não introduz uma elevada sobrecarga. A inserção de dados no repositório, não é efectuada com um desempenho constante, dado que quando é necessário efectuar um rebalanceamento da árvore, o

Tabela 5.3: Custo das operações (tempos em nanosegundos)

| Teste                     | Duração | Média por elemento |
|---------------------------|---------|--------------------|
| Adição de 1024 elementos  | 869 244 | 848.8711           |
| Remoção de 1024 elementos | 675 086 | 659.2637           |

processo de inserção é mais demorado devido à necessidade de efectuar rotações na árvore. Para além de estabelecer um bom compromisso de desempenho e utilização de memória, a sua disponibilidade de utilização no núcleo do sistema, possibilitou um elevado grau de confiança na sua utilização.

O tempo médio despendido na procura do elemento com o menor valor de chave, nos 1024 elementos adicionados, foi de 1327 nanosegundos. Com este valor é possível verificar que para efectuar 10 iterações de procura na árvore, incorre-se numa penalização de 1.3 microsegundos. Verifica-se que o tempo médio de procura de elementos na estrutura, é sempre menor ou igual a 1.3 microsegundos. Considerando que a maioria das aplicações não utiliza tantos portos em simultâneo, são expectáveis tempos inferiores em aplicações reais.

O teste de desempenho realizado demonstra que, mesmo nas piores condições, ou seja nas condições mais extremas, consegue-se obter um desempenho aceitável, tendo em conta que se utilizou uma estrutura de dados dinâmica, em que se tem flexibilidade para modificar os seus dados.

### 5.2.3 Desempenho do Sistema de instrumentação

Sendo a instrumentação das chamadas ao sistema um dos pontos fundamentais na execução do *MRoP*, a análise ao seu comportamento é deveras importante, na medida em que é necessário verificar se a introdução deste tipo de instrumentação irá produzir uma elevada penalização sobre o sistema de operação. A título exemplificativo da sobrecarga introduzida pelo *KRetProbe* em cada função instrumentada, foi adicionado *KRetProbe* à chamada ao sistema *getpid*. Esta chamada ao sistema foi escolhida, devido à simplicidade da função que apenas devolve o identificador do processo que a invocou. Este teste consistiu em avaliar o tempo decorrido entre o início e o fim do total das chamadas, com e sem o *KRetProbe*, de forma a avaliar a sobrecarga e verificar a sua coincidência com o indicado pelos criadores do sistema.

O valor de referência obtido pelos criadores do *KProbes*, referentes ao *KRetProbe* sem optimizações, é de 0.7 microsegundos[KPH], sendo que o valor médio obtido foi

Tabela 5.4: Duração das chamadas em segundos

| Teste                  | Original | Com <i>KRetProbe</i> | Sobrecarga por chamada  |
|------------------------|----------|----------------------|-------------------------|
| 100 000 000 chamadas   | 12.65    | 73.6600              | $610.10 \times 10^{-9}$ |
| 1 000 000 000 chamadas | 126.85   | 737.2100             | $610.36 \times 10^{-9}$ |

de 0.61 microsegundos, ou seja, ligeiramente inferior, visto que a máquina de referência apresenta uma frequência de *cpu* inferior à máquina onde foram realizados estes testes. Esta sobrecarga é constante, o que indica que qualquer melhoria ao *KProbes* afecta positivamente a sobrecarga do *MROp*.

Consideram-se estes valores bastante aceitáveis e espera-se que tenham ainda reduzido impacto no desempenho normal do sistema. Note-se ainda que a instrumentação só é introduzida aquando do carregamento do *MROp*, de modo a executar a monitorização com esta nova funcionalidade.

### 5.3 Conclusão

Com os testes efectuados ao nível funcional e ao nível de desempenho ao *MROp* verificou-se que este mecanismo oferece um desempenho aceitável nas condições mais adversas, o que indicia que para as condições das aplicações reais o seu desempenho será superior, enquanto que a sobrecarga introduzida no sistema poderá ser inferior ao constatado.

Pode verificar-se em relação ao trabalho [Far09], que a sobrecarga gerada pelo *MROp* foi muito inferior, demonstrando que apesar da dificuldade de operar no núcleo, criar um sistema de monitorização de rede orientado ao processo através de um mecanismo não intrusivo, que logo que possível rejeite a captura de pacotes, reduz consideravelmente a sobrecarga no sistema.

Existe ainda a possibilidade de reduzir a sobrecarga no sistema, se ao invés de se instrumentar as chamadas ao sistema, forem instrumentadas as funções específicas relacionadas com a pilha de protocolos *tcp/ip*.





## Conclusões e Trabalho Futuro

Esta dissertação teve como objectivo a criação de uma extensão ao actual *PCap/LSF*, utilizado no *Linux*, de modo a restringir a captura dos pacotes referentes a uma determinada aplicação, contribuindo deste modo para a redução da sobrecarga no sistema de monitorização com uma nova funcionalidade e identificar os fluxos de rede de uma forma não intrusiva.

Para a realização da *Monitorização de Rede orientada ao Processo (MRoP)*, foi necessário, para além de identificar os pontos e conhecer as razões que estiveram na génese do insucesso de anteriores trabalhos, criar alternativas que permitissem ultrapassá-los. Para atingir tal objectivo, foram estudados os mecanismos de monitorização internos ao núcleo e os sistemas de comunicação entre o núcleo e as aplicações em nível utilizador.

O estudo centrou-se, principalmente, nos mecanismos de monitorização ao nível do núcleo, pois estes permitem efectuar análises não intrusivas e, devido à sua localização, dispensam a permuta de dados entre o nível utilizador e o núcleo, o que a não acontecer contribuiria para o aumento da sobrecarga.

Para executar a extensão ao *PCap*, foi architectada uma solução utilizando o sistema de instrumentação dinâmica do núcleo (*KProbes*), para a análise das interacções do processo alvo com o exterior. Os dados relevantes desta interacção, são adicionados a um repositório, de modo a que o sistema *LSF* os possa consultar e decidir quais os pacotes a capturar, com base nestas informações.

Esta extensão foi analisada funcionalmente através de programas que criavam, comunicavam e destruíam canais de comunicação, onde todas estas interacções eram registadas e verificadas. Na avaliação funcional, foram igualmente executados programas que efectuavam transferências utilizando os protocolos *HTTP* e *FTP*, enquanto decorriam outros fluxos na rede. O *MROp* foi aplicado a estes programas, de modo a capturar todo o tráfego respeitante às transferências destes dois protocolos, garantindo a compatibilidade e a possibilidade de continuar a utilizar as ferramentas existentes como o *tcpdump* e o *Wireshark*. Foi possível constatar a correcção dos protocolos e verificar que apenas as interacções da aplicação alvo com o exterior, foram capturadas. Para além desta análise funcional, foi efectuada uma outra, onde se compararam os desempenhos do *PCap*, com e sem esta nova extensão, a fim de determinar qual a sobrecarga introduzida pelo *MROp* na monitorização já existente. Os resultados apurados evidenciam que a sobrecarga é praticamente inexistente nos piores casos, trazendo vantagens na monitorização quando estivermos perante vários fluxos de dados irrelevantes para a análise.

Nas secções seguintes são apresentados as principais conclusões da realização desta dissertação, assim como as possíveis evoluções e extensões ao *MROp*.

## 6.1 Conclusões

O objectivo proposto de criar uma extensão ao actual sistemas de monitorização genérico de rede, que incluía a monitorização das interacções de rede de um processo com base no identificador deste e efectuar a captura dos pacotes da aplicação, foi atingido, constatando-se ainda, que a sobrecarga imposta sobre o actual sistema de monitorização é praticamente irrelevante.

O *MROp* é um módulo do núcleo que permite monitorizar as interacções de rede de um processo, sem que exista um conhecimento prévio dos portos utilizados pela aplicação. Esta monitorização é inócua para a aplicação, porquanto esta desconhece que está a ser monitorizada, possibilitando ao administrador monitorizar aplicações sem acesso ao código fonte.

Se analisarmos a aplicação, do ponto de vista de segurança na rede, é possível verificar se está, ou não, a enviar indevidamente informação para o exterior. Esta situação, não é possível de observar com o recurso ao normal funcionamento da biblioteca *PCap*, sem um conhecimento detalhado do seu funcionamento e protocolos, ou a uma monitorização da aplicação de forma intrusiva, o que pode originar comportamento errático e afectar negativamente o desempenho da aplicação e do sistema. O recurso à introdução desta funcionalidade não intrusiva, para a captura e análise do tráfego

de um processo, constitui um avanço relativamente à monitorização de rede efectuada através da biblioteca *PCap*.

O *MROp*, ao oferecer a possibilidade de capturar exclusivamente os pacotes de um determinado processo ou de uma família de processos, facilita as análises a efectuar e reduz a sobrecarga neste tipo de sistemas, dispensando a biblioteca *PCap* de capturar o tráfego não pretendido e, de conhecer os protocolos e portos utilizados pela aplicação.

Esta funcionalidade, é transparente para todas as ferramentas desenvolvidas com base no *PCap*, pelo que todas podem dela usufruir.

Relativamente à sobrecarga introduzida, tendo como referência a monitorização de rede já existente, esta revelou-se insignificante mesmo nos piores casos, melhorando substancialmente aqueles em que incide sobre o tráfego de um único processo, reduzindo igualmente o trabalho realizado pelo *LSF* e pela biblioteca *PCap*.

As vantagens do sistema criado assumem maior notoriedade, quando a máquina se encontra perante uma carga mais elevada de trabalho, ou um grande volume de tráfego de rede e/ou muitos fluxos irrelevantes, dado manter o uso dos recursos na proporção aproximada apenas do tráfego do processo alvo.

## 6.2 Trabalho Futuro

Como trabalho futuro existe a possibilidade de expandir e melhorar o suporte para múltiplos processos a serem monitorizados. Acresce ainda a possibilidade de verificar problemas de concorrência na presença de múltiplos *cores/cpus*, e garantir a impossibilidade de ocorrência de *race conditions*.

Considerando que o sistema implementado se limita a monitorizar protocolos acen-tes em *TCP* e *UDP*, poderia ver a sua contribuição alargada se abrangesse outros protocolos como *icmp*, *arp*, *stp*, etc.

Outra possibilidade será procurar optimizar a instrumentação, evitando a instrumentação das funções *sendto* e *recvfrom*, restringindo a sua aplicação a funções internas específicas dos protocolos monitorizados.

Pretende-se partilhar o uso destas funcionalidades submetendo este sistema a análise da comunidade utilizadora do sistema *Linux* com vista à sua implementação na versão principal do núcleo do *Linux*. Considera-se ainda a integração deste trabalho com o anterior [DF10, Far09], com vista à obtenção de uma ferramenta de monitorização distribuída com baixa sobrecarga.



# Bibliografia

- [Adm09] Linux Foundation Administrator. napi. *The Linux Foundation*, Novembro 2009.
- [BDK<sup>+</sup>97] M. Beck, M. Dziadzka, U. Kunitz, R. Magnus, e D. VerWorner. *Linux Kernel Internals*. Addison-Wesley, 1997.
- [BDKV02] M. Beck, M. Dziadzka, U. Kunitz, e D. VerWorner. *Linux Kernel Programming*. Addison-Wesley, 2002.
- [Ben05] Christian Benvenuti. *Understanding Linux Network Internals*. O'Reilly, 2005.
- [Der04] Luca Deri. Improving passive packet capture: Beyond device polling. Proceedings of SANE, Oct. 2004.
- [Der10] Luca Deri. *Exploiting Commodity Multi-core Systems for Network Traffic Analysis*, 2010.
- [DF10] V. Duarte e N. Farruca. Using libpcap for monitoring distributed applications. In *International Conference on High Performance Computing & Simulation (HPCS 2010)*, pág. 92–97. IEEE, 06 2010.
- [DIH<sup>+</sup>07] Stephanie Donovan, Gerrit Huizenga Ibm, Andrew J. Hutton, Andrew J. Hutton, C. Craig Ross, C. Craig Ross, Linux Symposium, Linux Symposium, Linux Symposium, Martin K. Petersen, Wild Open Source, Tom Zanussi, Karim Yaghmour, Robert Wisniewski, Richard Moore, e Michel Dagenais. relayfs: An efficient unified approach for transmitting data from kernel to user space. 2007.
- [DPr] *DProbes*. <http://dprobes.sourceforge.net/> Site visitado em Abril de 2010.

- [Dua05] V. Duarte. *Uma Arquitectura para a Monitorização de Computações Paralelas e Distribuídas*. Tese de Doutoramento, Faculdade de Ciências e Tecnologia, May 2005.
- [Far09] Nuno Miguel Galego Farruca. Wireshark para sistemas distribuídos. Tese de Mestrado, Faculdade de Ciências e Tecnologia - Universidade Nova de Lisboa, 2009.
- [GM] Thomas Gleixner e Ingo Molnar. hrtimers - subsystem for high-resolution kernel timers. Site consultado em janeiro de 2011.
- [Hir05] Masami Hiramatsu. Overhead evaluation about kprobes and dprobe (direct jump probe). Relatório técnico, Julho 2005.
- [HMC94] Jeffrey Hollingsworth, Barton P. Miller, e Jon Cargille. Dynamic program instrumentation for scalable performance tools. pág. 841–850, 1994.
- [Int] *Introducing PF\_RING DNA (Direct NIC Access)*. <http://www.ntop.org/blog/?p=50> site consultado em junho de 2010.
- [Jon09] M. Tim Jones. Linux introspection and systemtap an interface and language for dynamic kernel analysis. Relatório técnico, Nov. 2009.
- [KPH] Jim Keniston, Prasanna S Panchamukhi, e Masami Hiramatsu. Kernel probes (kprobes). Consultado em abril de 2010.
- [KPr] Kprobes. <http://sourceware.org/systemtap/kprobes/> Site consultado em Março de 2010.
- [Lib] *LibPcap*. <http://www.tcpdump.org/> Site consultado em Abril de 2010.
- [LML09] Byungjoon Lee, Seong Moon, e Youngseok Lee. Application-specific packet capturing using kernel probes. In *IM'09: Proceedings of the 11th IFIP/IEEE international conference on Symposium on Integrated Network Management*, pág. 303–306, Piscataway, NJ, USA, 2009. IEEE Press.
- [MD11] Nuno Martins e Vítor Duarte. Pcap com filtragem orientada ao processo. *Inforum 2011 Simpósio de Informática*, Setembro 2011.
- [MJ92] Steven Mccanne e Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. pág. 259–269, 1992.

- [MR09] Desnoyers Mathieu e Dagenais Michel R. Lttnng, filling the gap between kernel instrumentation and a widely usable kernel tracer. 2009.
- [Pan04] Prasanna S. Panchamukhi. Kernel debugging with kprobes insert printk's into the linux kernel on the fly. Relatório técnico, aug 2004.
- [PFR] PF\_RING. [http://www.ntop.org/PF\\_RING.html](http://www.ntop.org/PF_RING.html) Site consultado em junho de 2010.
- [SV08] Sammer Seth e M. Ajaykumar Venkatesulu. *TCP/IP Architecture, Design, and implementation in Linux*. Wiley, 2008.
- [TZYW03] From Kernel To, Tom Zanussi, Karim Yaghmour, e Robert Wisniewski. relayfs: An efficient unified approach for transmitting data. In *In Proceedings of the Ottawa Linux Symposium 2003*, pág. 494–507, 2003.
- [vdMChCS00] Jacobus van der Merwe, Ramón Cáceres, Yang hua Chu, e Cormac Sreenan. mmdump: a tool for monitoring internet multimedia traffic. *SIGCOMM Comput. Commun. Rev.*, 30(5):48–59, 2000.
- [Wil] E. Cohen. William. Tuning programs with oprofile. *WIDE OPEN MAGAZINE*.
- [WR05] Klaus Wehrle e Harmut Ritter. *The Linux Networking Architecture, Design and Implementation of Network Protocols in the Linux Kernel*. Pearson - Prentice Hall, 2005.
- [WXW08] Zhenyu Wu, Mengjun Xie, e Haining Wang. Swift: a fast dynamic packet filter. In *NSDI'08: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pág. 279–292, Berkeley, CA, USA, 2008. USENIX Association.
- [YD00] Karim Yaghmour e Michel R. Dagenais. Measuring and characterizing system behavior using kernel-level event logging. In *Proceedings of the annual conference on USENIX Annual Technical Conference, ATEC '00*, pág. 2–2, Berkeley, CA, USA, 2000. USENIX Association.