3
A ferramenta proposta

Este capítulo descreve as decisões do projeto e o desenvolvimento da ferramenta de depuração proposta por este trabalho, onde sua representação em blocos é exibida na figura 3.1. As figuras do núcleo e do código de monitoração estão inseridas no contexto do sistema operacional. As figuras representadas pela interface núcleo-usuário e a representação do componente equivalem a processos em modo usuário.

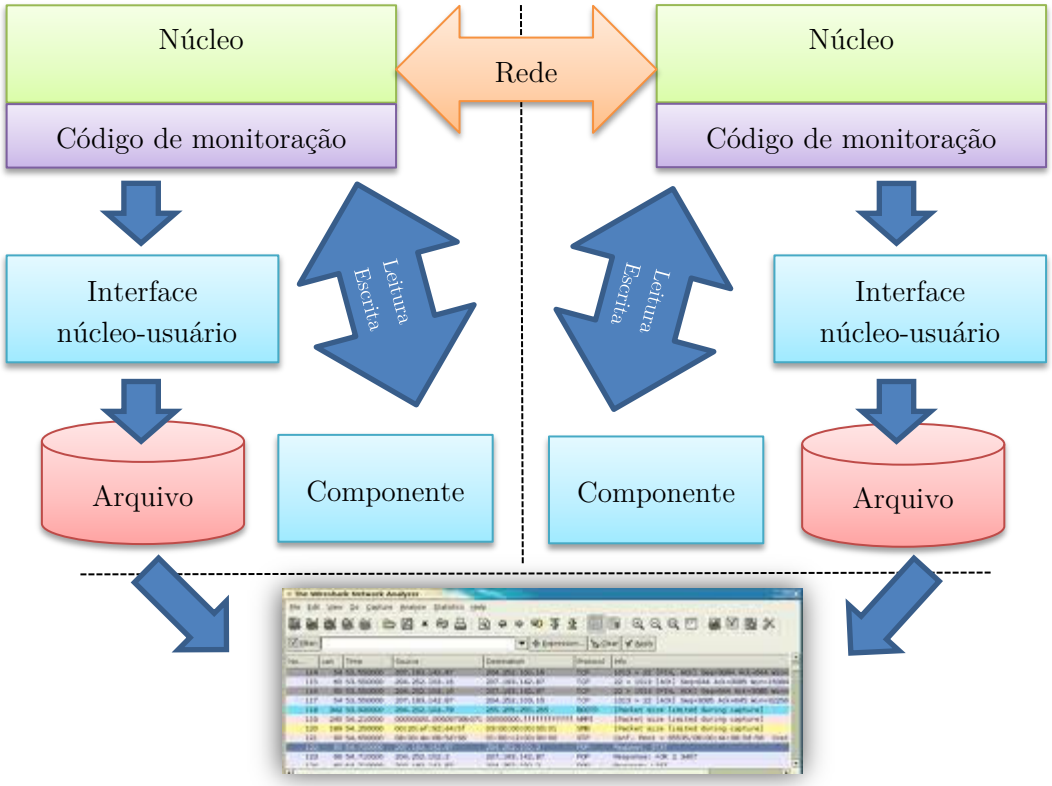


Figura 3.1 Representação gráfica da ferramenta

Ao longo deste capítulo, a seção 3.1 será dedicada a mostrar as decisões tomadas para o desenvolvimento da ferramenta, seguida da seção 3.2, sobre o desenvolvimento e a forma de funcionamento do módulo para o núcleo do sistema operacional. Na seção 3.3, teremos a descrição do dispositivo de comunicação com o modo usuário e na seção 3.4, a apresentação do formato das mensagens de instrumentação. Finalizamos o capítulo com a seção 3.5, com o enquadramento da ferramenta segundos as metodologias apresentadas no capítulo 2 e mostrando as melhores práticas para a utilização da ferramenta.

3.1 Modelagem da ferramenta

As chamadas de sistemas são um artifício do sistema operacional para manter, em posições fixas na memória, as funções que servem de interface para o modo usuário. Um dos motivos para usar esse artifício é evitar que se exponham diretamente os endereços das funções usadas para a interface entre o núcleo e o usuário, uma vez que, a cada nova versão do núcleo, essas posições de memória, muito provavelmente, sofreriam alterações.

Dessa forma, as chamadas de sistema são agrupadas em um vetor denominado vetor de chamadas de sistema. As posições desse vetor correspondem as identificações das chamadas de sistema, e cada posição desse vetor possui um ponteiro para a posição de memória onde se encontra, no núcleo, o código correspondente a chamada de sistema relativa ao índice do vetor, como mostra a figura 3.2. Logo, o vetor de chamada de sistema é um vetor contendo indireções para funções localizadas no núcleo.

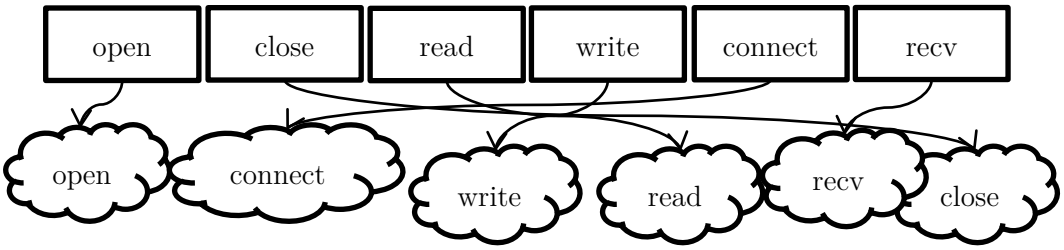


Figura 3.2 Indireções formadas pelo vetor de chamadas de sistema

A principal função da ferramenta descrita por este trabalho é a monitoração de eventos de leitura e escrita, através do uso de desvios no vetor de chamadas de sistema, para processos previamente selecionados pelo usuário. A figura 3.3 demonstra como um processo de usuário invoca uma chamada de sistema. Nela, é necessário ressaltar que o código responsável por invocar a chamada de sistema se encontra nas bibliotecas utilizadas pelo software, logo, fora do código fonte do software.

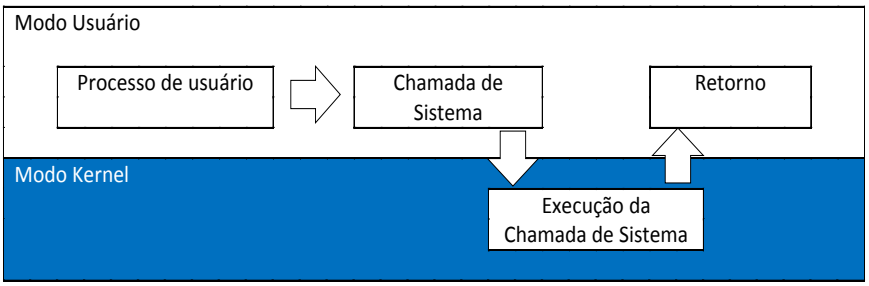


Figura 3.3 Executando uma chamada de sistema

Porém, a habilidade de monitorar chamadas de sistema só é dada a códigos que executem com o mesmo privilégio do núcleo. Para tal, é importante lembrar que, para os processadores da linha Intel x86 [23], os privilégios são controlados por

intermédio de anéis de proteção, ou *protection rings*. Nesses anéis, nos sistemas operacionais mais conhecidos, somente os anéis de número 0 e 3 são utilizados. Dessa forma, conforme ilustrado na figura 3.4, o anel 0 é utilizado para o código executado pelo núcleo, e o anel 3 é utilizado para os demais processos de usuário [24].

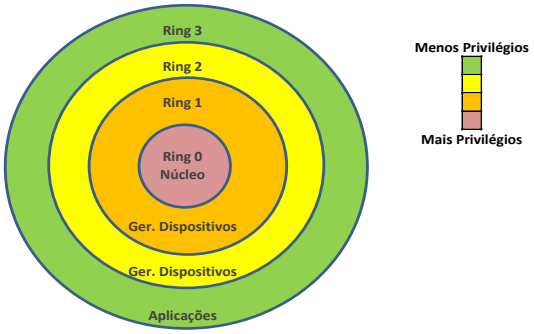


Figura 3.4 Separação de privilégios nos processadores x86

Logo, os anéis 1 e 2, por questões de portabilidade com outras arquiteturas de microprocessadores, não são utilizados. Assim, é restrito o acesso ao núcleo por parte dos processos de usuário, logo, para ser possível ultrapassar a restrição imposta pelos anéis de proteção, é necessário que o núcleo, com acesso privilegiado, faça a carga do código desejado. Para isso, o núcleo conta com a habilidade de incorporar códigos desenvolvidos por terceiros em tempo de execução, sem a necessidade reinicialização ou alteração do seu código fonte.

Todavia, aplicações de usuários não podem executar com o mesmo privilégio do núcleo. Assim, um software que tenha, simultaneamente, código a ser executado pelo núcleo e pelo usuário, não pode ser executado integralmente no núcleo ou em modo usuário. Sendo assim, é necessário separar o software em pelo menos duas camadas. A primeira, com os mesmos privilégios do núcleo, e a segunda camada, com os mesmos privilégios de usuário.

O código executado no núcleo, descrito na seção 3.2, será responsável por monitorar e interceptar, por intermédio das chamadas de sistema de leitura *read()* e escrita *write()*, a comunicação entre os componentes. Lembrando que para o núcleo, componentes são equivalentes a processos.

Entretanto, o desenvolvimento de código para o núcleo apresenta desafios que complicam a elaboração de tarefas mais sofisticadas. A falta de uma ferramenta de depuração amigável e a inexistência de várias funções presentes na biblioteca C padrão faz do desenvolvimento de código para o núcleo uma tarefa árdua. Assim, optou-se por escrever a maior quantidade de código possível fora do núcleo, originando o código descrito na seção 3.3. Dessa forma, o código desenvolvido para o núcleo é apenas aquele que só pode ser executado dentro deste.

O código da referida seção armazena informações em um arquivo, porém, ainda é necessária uma interface para o usuário que o auxilie na visualização e interpretação destas informações. Nesta interface, descrita na seção 3.4, o usuário é capaz de realizar pesquisas e visualizar os dados trocados entre componentes obtidos pela monitoração.

Em suma, através do código inserido no núcleo, os componentes, selecionados de acordo com o número identificador do seu processo, têm seus eventos de leitura e escrita monitorados e enviados para a interface núcleo-usuário. A interface núcleo-usuário trata as informações obtidas do núcleo e as armazena no sistema de arquivos. A interface com o usuário lê os arquivos do sistema de arquivos e exibe as informações para o usuário da interface.

3.2 Instrumentação do núcleo do SO

A função da instrumentação do núcleo é permitir a monitoração das chamadas de sistema e repassar para o modo usuário os dados obtidos pela monitoração. Por este módulo, deseja-se acesso integral os parâmetros de entrada das chamadas de sistema e que tais parâmetros possam ser transferidos, sem restrições, para o modo usuário. Dessa forma, para realizar a monitoração das chamadas de sistema, existem pelo menos três métodos conhecidos, que são listados a seguir:

i. ***ptrace***: ptrace é uma chamada de sistema que permite que um processo A observe e controle a execução de um processo B, permitindo A examinar a área de memória e alterar registradores de B [25]. Via ptrace, A também é capaz de modificar a maneira como as chamadas de sistema são realizadas pelo processo B.

ii. **Rastreadores dinâmicos do núcleo**: Também conhecido por dynamic *kernel tracing*, *DTrace* [26] e *SystemTap* [27] são, atualmente, os rastreadores para núcleo mais conhecidos. Por meio de uma linguagem de programação interpretada, os rastreadores são capazes de monitorar a entrada e saída de todos dos pontos considerados importantes no núcleo do sistema operacional, incluído as chamadas de sistema, sem a necessidade de reiniciar o sistema operacional ou realizar qualquer modificação no código do núcleo.

iii. **Carga dinâmica de código no núcleo**: A maioria dos sistemas operacionais modernos permite extensões ao núcleo [28]. Tais extensões, também conhecidas como módulos, são códigos compilados, introduzidos no núcleo do sistema operacional em tempo de execução, sem a necessidade de reiniciá-lo.

Para o método descrito em (i), mesmo sendo uma chamada que segue o padrão POSIX [29] (*Portable Operating System Interface*, ou, Interface de Portabilidade de Sistemas Operacionais), comum na maioria dos sistemas operacionais modernos, suas características foram se tornando problemáticas ao longo do tempo [30]. Porém, mesmo sendo uma chamada padronizada, para sua utilização existe a necessidade do conhecimento prévio do funcionamento do sistema operacional e da arquitetura do processador em uso. A dificuldade encontrada na utilização deste método é devido à *ptrace* não fornecer uma interface própria para o desvio de cha-

madas de sistemas, trata-se apenas de uma interface para o acesso, leitura e escrita de regiões de memória do processo monitorado.

Por exemplo, em [31] é descrito o desenvolvimento da monitoração de chamadas de sistema, apenas para processadores x86 e sistema operacional Linux, usando *ptrace*. Lá é possível avaliar a dificuldade de utilizar este método. Como agravante, dependendo de como for desenvolvido, ainda há a possibilidade de considerável degradação de desempenho, devido às excessivas trocas de contexto entre núcleo, processo depurador e processo depurado [32].

Assim, podemos concluir que, devido a *ptrace* poder gerar um impacto de desempenho indesejado e sua utilização variar conforme o sistema operacional e a arquitetura utilizada pelo processador [25], desenvolver uma solução genérica, compatível com todas as plataformas aderentes ao padrão POSIX e com baixa degradação de desempenho, corresponde a uma tarefa infactível. Em contrapartida, o método descrito em (ii) é mais moderno e mais apropriado para o objetivo deste trabalho. Apesar de haver duas apresentações distintas, *DTrace* e *SystemTap* compartilham o mesmo propósito, de ser uma ferramenta de rastreamento localizada no núcleo.

Por executarem dentro do núcleo, o impacto de desempenho é bem menor se comparado à *ptrace* [32], além de serem suportadas pelos desenvolvedores do próprio núcleo, sendo sua utilização, de acordo com os desenvolvedores, uma opção segura e encorajada. Ainda que sejam linguagens de programação diferentes (*DTrace* utiliza D, uma linguagem que se assemelha a uma versão simplificada de C [33], enquanto *SystemTap*, utiliza TCL), são semanticamente equivalentes, não havendo impedimentos para escrever um código monitor de chamadas de sistema que apresentem os mesmos resultados para ambas.

Entretanto, como enfatizado na documentação, apenas as entradas e saídas das funções são monitoradas. Tal informação é especialmente percebida na chamada de sistema de leitura. Na leitura, o ponto de entrada recebe um vetor de dados vazio, na saída, o vetor de dados não está mais disponível, como demonstra o diagrama da figura 3.5, onde apenas o resultado da chamada pode ser monitorado.

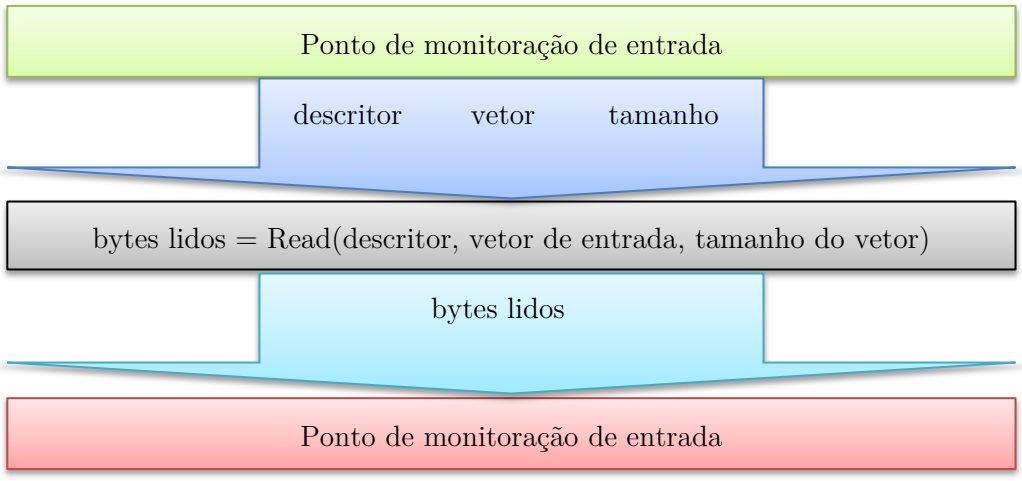


Figura 3.5 Entrada/saída da monitoração de leitura

Já na chamada equivalente à escrita, o ponto de entrada da chamada recebe o vetor preenchido com os dados a serem enviados para o destino, entretanto, não há uma interface própria para a cópia da área de memória do vetor para uma área de memória acessível a programas de usuários. Logo, podemos deduzir que rastreadores dinâmicos do núcleo, devido às limitações já mencionadas, não satisfazem a questão de monitorar, com acesso integral aos parâmetros, as chamadas de sistema. Assim, também não atendem ao objetivo deste trabalho.

Por eliminação, o método descrito em (iii), consequentemente, se torna o mais indicado. Entretanto, módulos de núcleo não são intercambiáveis entre sistemas operacionais, sendo assim, nos leva a abandonar a proposta de um método multi-plataforma de monitoramento de chamadas de sistemas. Por esse motivo, como referência, foi escolhido o sistema operacional Linux [34], arquitetura x64 [23]. Com esta especificação, e por ser um ambiente muito comum, a distribuição *Red Hat Enterprise* versão 5 [35], ou suas versões livres CentOS [36] ou Scientific Linux [37], serão utilizadas para a implementação de referência do módulo de monitoramento.

Apesar de ser necessária a escolha de um sistema operacional para o desenvolvimento do módulo, trata-se de uma proposta genérica. Em todos os sistemas operacionais modernos, a interface criada pelas chamadas de sistemas é a base de comunicação entre o núcleo e os programas de usuários. Assim, mesmo com variações sobre a maneira como monitorar chamadas em cada arquitetura e sistema operacional, a proposta ainda é válida. A figura 3.6 mostra a objetivo deste módulo, onde a biblioteca C é a camada de abstração entre o processo e o núcleo. O propósito do módulo é realizar desvios nas chamadas de sistema originais, não interferindo na biblioteca C, promovendo um funcionamento similar ao já existente nos rastreadores dinâmicos de núcleo, porém, com acesso integral aos parâmetros de entrada e aos códigos de retorno.

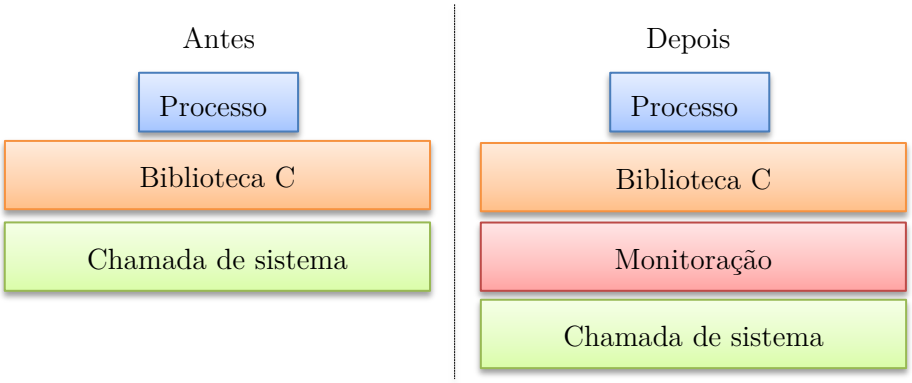


Figura 3.6 O módulo, antes e depois

Entretanto, em um sistema operacional como o Linux, existem mais de 300 chamadas de sistemas disponíveis. Monitorar todas elas, além de ser desnecessário, representaria um esforço de desenvolvimento muito grande. Dessa forma, sabe-se que a principal forma de comunicação entre componentes de um software distribuído se dá pelo uso *sockets*, que é uma interface oferecida pelo sistema operacional

para programas que desejam se comunicar através de uma rede [38]. Também se sabe que *sockets* dependem, principalmente, das chamadas de leitura e escrita para realizar a comunicação entre programas. Por este motivo, estas foram as chamadas de sistema selecionadas para a monitoração pelo módulo.

Todavia, em sistemas operacionais, as chamadas de sistema mais intensamente utilizadas são, sem sombra de dúvida, as de leitura *(read)* e escrita *(write)* [39]. Com isso, devemos tomar especial cuidado ao manipular tais chamadas, pois qualquer alteração no comportamento delas poderá levar ao comprometimento do funcionamento do sistema operacional.

Assim, para reduzir qualquer interferência que possa vir a ser causada pelo desvio das chamadas de leitura e escrita, a monitoração e a interface núcleo-usuário foram isoladas por um buffer circular, conforme figura 3.7. O buffer circular é um algorítmo que garante que uma operação de leitura possa funcionar paralelamente, de forma concorrente e sem intertravamentos, a uma operação de escrita [40].

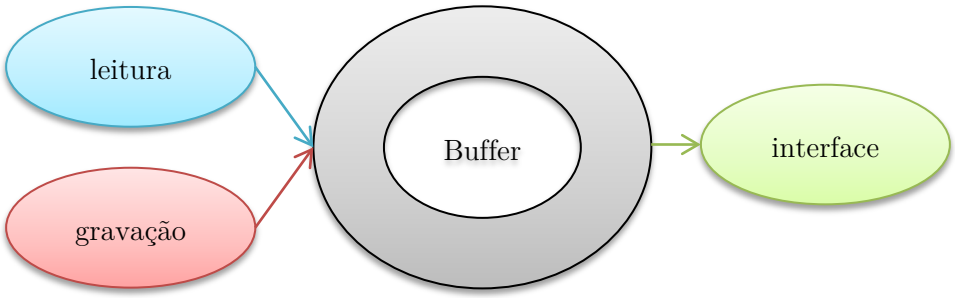


Figura 3.7 Isolamento do monitoramento e interface núcleo-usuário

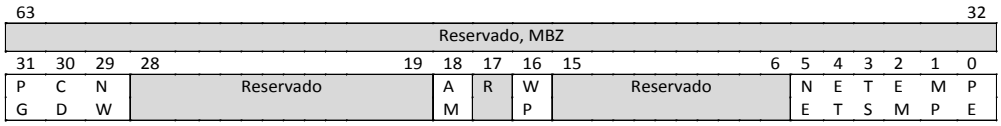
3.2.1 Monitoramento

Para que seja possível o monitoramento por intermédio do núcleo, existem dois desafios a serem vencidos: (i) inibir, da região de memória do núcleo, as proteções contra modificações; (ii) encontrar o vetor de chamadas de sistema.

O núcleo, de forma a garantir sua integridade, é protegido por um mecanismo que evita com que modificações acidentais possam ser realizadas em sua região de memória, fazendo com que a memória ocupada pelo núcleo seja protegida contra escrita. Tal proteção é habilitada no início do processo de inicialização do sistema operacional. Por este motivo, tentativas de alterar tal região de memória resultam em uma falha de proteção, e consequentemente, a paralização do funcionamento do sistema operacional.

O mecanismo responsável, nos processadores baseados na arquitetura x86 e x64, por tal proteção no núcleo é o registrador *CR0*. Tal registrador contêm vários bits de controle, entre eles, conforme a figura 3.8 [41], o bit de proteção contra escrita *WP* (*write protect*). Alterando esse bit, a proteção contra escrita, mesmo em regiões de memória marcadas como “somente leitura”, como o caso da região onde

o núcleo reside, passa a ser inibida, permitindo que ocorram alterações nessa região [41,42].



Bits	Mnemônico	Descrição	R/W
0-32	Reservado	Reservado, Deve ser Zero	
31	PG	Paginação	R/W
30	CD	Cache Desabilitado	R/W
29	NW	Não Writethrough	R/W
28-19	Reservado	Reservado	
18	AM	Máscara de Alinhamento	R/W
17	Reservado	Reservado	R/W
16	WP	Proteção Contra Gravação	R/W
15-6	Reservado	Reservado	
5	NE	Erro Numérico	R/W
4	ET	Tipo Estendido	R
3	TS	Tarefa Trocada	R/W
2	EM	Emulação	R/W
1	MP	Monitor do Coprocessador	R/W
0	PE	Proteção Habilitada	R/W

Figura 3.8 Atribuições do registrador CR0

Poder gravar na área restrita ao núcleo é condição necessária para realização dos desvios necessários. Com essa condição satisfeita, o próximo desafio é encontrar e alterar o vetor de chamadas. É importante ressaltar que o núcleo do Linux, a partir da versão 2.6, a mesma versão utilizada neste trabalho, o vetor de chamadas de sistemas deixou de ser exposto publicamente aos módulos por motivos de segurança, como ocorria em versões anteriores [43]. Sua localização, entretanto, é conhecida apenas no momento da compilação do núcleo. Devido a esta restrição, para termos acesso ao vetor de chamadas, é necessário conhecer em qual posição da memória o início do vetor está localizado. Para tal, existem pelo menos dois métodos conhecidos.

O primeiro método consiste em usar a força bruta. Supondo que o vetor de chamadas *v* está localizado entre o endereço de duas funções públicas *a* e *b*, sabendo o índice *i* de uma das chamadas do vetor que aponte para uma função pública *f*, podemos iterar a memória entre *a* e *b,* até achar uma posição para v que indexado do índice i aponte para a função *f,* onde *v[i] = f*. Convertendo esse raciocínio em pseudocódigo, temos o equivalente à figura 3.9.

```
for v = a to b
do
    if (v[i] = f) then break
done
if (v[i] != f) then “Fail!”
```

Figura 3.9 Pseudocódigo do método da força bruta

Porém, a cada versão nova do núcleo, alterações realizadas no código fonte podem alterar a posição do vetor de chamadas em relação às outras funções públicas, logo, não há garantias de que o vetor estará sempre entre as mesmas funções.

Sendo assim, esse método não é eficaz por funcionar em algumas versões, mas não em outras.

O segundo método utiliza uma abordagem mais robusta a alterações na posição do vetor de chamadas. Entretanto, requer a existência de um arquivo, gerado no momento da compilação, contendo o mapa de todos os métodos e atributos disponíveis no núcleo. Tal mapa, representado pelo arquivo *System.map*, contêm todas as funções e variáveis existente no núcleo, independente de serem, ou não, exportadas.

Assim, o vetor de chamadas pode ser encontrado realizando uma varredura no referido arquivo. Uma vez encontrado o vetor, as chamadas de leitura e escrita podem facilmente ser alteradas. Embora as distribuições Linux que priorizem a segurança não disponibilizem o arquivo mencionado, este arquivo, instalado por padrão no Red Hat Linux e na maioria das demais distribuições, compõe os arquivos que integram o pacote de instalação do núcleo do Linux. Logo, enquanto houver a disponibilidade do arquivo *System.map*, a localização do vetor de chamadas não será um problema.

Todavia, por estarmos usando como referência a família de processadores x64, não podemos nos esquecer de que, por conta da compatibilidade desses processadores com a família x86, existem dois vetores de chamadas de sistema, um para cada tipo de micro instrução. Logo, devemos fazer o redirecionamento das duas versões do vetor, como mostra a figura 3.10.

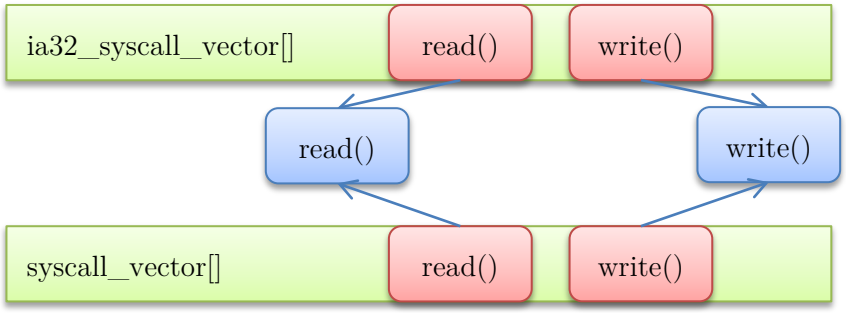


Figura 3.10 Os dois vetores de chamadas de sistema

Apesar de haverem chamadas de sistema em versões de 32 bits e 64 bits, elas apontam para uma mesma função base. A existência dos dois vetores é necessária para que o núcleo possa diferenciar as chamadas de 32 bits, que devem ter seus parâmetros ajustados, das de 64 bits. O núcleo trata os parâmetros de entrada das chamadas para que esta compatibilidade seja garantida. Dessa forma, não há necessidade de manter duas versões da função base.

Vale informar que os vetores em questão, com o objetivo de não sobrecarregar o sistema operacional desnecessariamente, são modificados apenas quando a interface núcleo-usuário, descrita na seção 3.3, estiver em uso. Sendo assim, somente quando o desvio estiver ativo que o algoritmo descrito na figura 3.11 é aplicado.

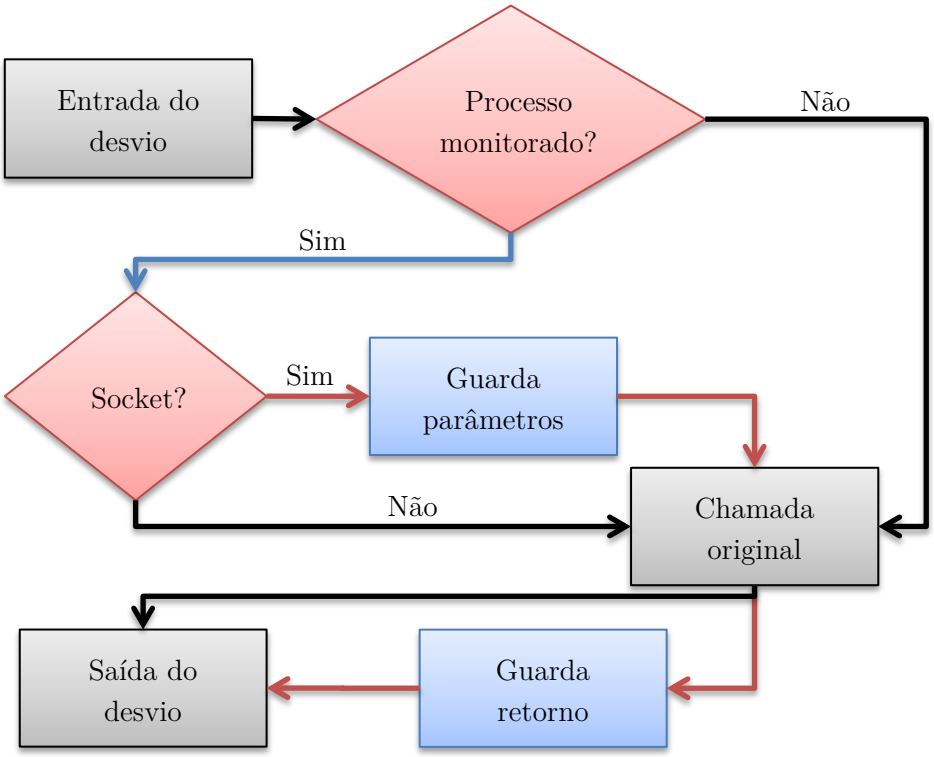


Figura 3.11 O algoritmo de interceptação

Ao entrar no desvio, é verificado, através da localização em uma tabela de dispersão *(hash table)*, se o processo está marcado para ser monitorado. Caso positivo, verifica-se se o tipo de comunicação é do tipo *socket*, pois nos interessa apenas a comunicação do processo com a rede. Caso positivo, a chamada original é invocada. Ao receber o retorno da chamada, o valor de retorno corresponde à quantidade de bytes efetivamente lidos ou escritos. Esse conjunto de bytes será armazenado do parâmetro de entrada e salva no buffer circular, bem como os demais parâmetros de entrada. Caso qualquer condição apresentada não seja satisfeita, a chamada original é invocada, sem qualquer tratamento pelo módulo.

3.2.2 Dispositivo de comunicação com o modo usuário

O monitoramento, descrito na seção 3.2.1, promove o desvio das chamadas de sistema e faz com que as funções base das chamadas sejam trocadas por funções de intermediação, que interceptam os parâmetros de entrada e saída, e em seguida, seguem o fluxo normal de processamento, executando as funções que originalmente ocupavam aquelas posições do vetor. O intuito do dispositivo de comunicação com o modo usuário é promover a comunicação do núcleo com o modo usuário, entregando as informações interceptadas, sem tratamentos, para um software em modo usuário, para que este faça os tratamentos devidos.

A proteção entre o modo privilegiado e não privilegiado impede a comunicação direta entre os dois modos, por esta razão, nos sistemas operacionais *Unix* e assemelhados, os controladores de dispositivos, ou *drivers*, que executam em

modo privilégiado, se comunicam com programas não privilegiados por intermédio de arquivos localizados no diretório */dev*.

Existem, neste diretório, dispositivos de bloco, de caractere, e *sockets* de domínio local. Os dispositivos de bloco são utilizados para leitura de blocos com tamanho fixo, como leitura em blocos de um disco rígido. Os dispositivos de caractere leem fluxos seriais de bytes, e são mais adequados para enviar e receber comandos de dispositivos de hardware em geral, como em modems. Já os *sockets* de domínio local são utilizados para comunicação local, entre processos.

A natureza da comunicação deste módulo é serial, por este motivo, o dispositivo de caractere é o mais indicado para a comunicação com o usuário. O módulo, ao ser carregado pelo núcleo, exibe na saída padrão, conforme a figura 3.12, o identificador do dispositivo, para que o usuario crie o nó de comunicação no diretório */dev* com o comando *mknod*[4].

```
interceptor[main.c:init:55]: Kernel interceptor compiled on Feb  2 2013@12:05:56
interceptor[cdev.c:createdevfsfile:134]: Registered character device major 251 minor 0
```

Figura 3.12 Mensagem de saída apresentada após a carga do módulo

Com o objetivo de evitar que o uso concorrente corrompa o funcionamento do módulo, este tipo de dispositivo recebe notificações quando há tentativas de abertura ou quando há eventos de leitura ou escrita para ele. Ao ser notificado que há uma tentativa de abertura do dispositivo, o módulo deve se certificar que o dispositivo não está em uso, em outras palavras, que não há uso concorrente do dispositivo. Se houver uso concorrente, a tentativa é rejeitada e um código erro é enviado ao usuário. Se a tentativa obtiver sucesso, os vetores de chamadas de sistema são alterados, desviando as chamadas de sistema para as funções monitôras, e consequentemente, alimentando o buffer circular, disponibilizando bytes para serem lidos, via dispositivo de caractere, pelo usuário.

O protocolo de comunicação esperado pela interface núcleo-usuário, que será apresentado na seção 3.3, é estruturado na tabela 3.1, seguido da carga útil produzida pelo evento, que corresponde ao produto da monitoração da chamada de sistema. Os primeiros 128 bits equivalem ao horário a qual o evento ocorreu, com precisão de nanosegundos. Seguido de 64 bits com os endereços IP de origem e destino, seguido de outros 64 bits com a identificação do processo, com mais 64 bits de identificação do processo pai[5], seguido dos 32 bits de identificação *socket*, com mais 32 bits contendo as portas de comunicação da origem e do destino, seguido de 8 bits, que identificam se o evento é de leitura *(r)* ou escrita *(w)*.

A formação dessa estrutura deve seguir as regras de alinhamento da arquitetura x64, de 64 bits, logo, deve ser arredondada para múltiplos de 64 bits. Por esse motivo, 40 bits são desperdiçados ao final da estrutura, por conta do preenchimento devido ao alinhamento *(padding)*, até completar o próximo múltiplo de 64 bits. Logo, para ler um evento do dispositivo de caractere, o software que utilizar a in-

[4] Comando para criar arquivos especiais para a comunicação com dispositivos.

[5] O processo que instanciou o processo atual.

terface deve ler os primeiros 56 bytes do dispositivo, verificar o tamanho da carga útil que está anexada ao evento, descartar os próximos 5 bytes, e ler, na sequência, toda a carga útil. O próximo evento virá na sequência deste, sendo necessário ler um novo cabeçalho de 56 bytes, ignorar os próximos 5 bytes, e ler uma nova carga útil.

<table>
<tr><td colspan="8">Disposição dos bits</td></tr>
<tr><td>8</td><td>16</td><td>24</td><td>32</td><td>40</td><td>48</td><td>56</td><td>64</td></tr>
<tr><td colspan="8">timeval->sec</td></tr>
<tr><td colspan="8">timecal->nsec</td></tr>
<tr><td colspan="4">Endereço IP de origem</td><td colspan="4">Endereço IP de destino</td></tr>
<tr><td colspan="4">Identificador do processo</td><td colspan="4">Thread group do processo</td></tr>
<tr><td colspan="4">Identificador do processo pai</td><td colspan="4">Thread group do processo pai</td></tr>
<tr><td colspan="4">Identificador do socket</td><td colspan="2">Porta de Origem</td><td colspan="2">Porta de Destino</td></tr>
<tr><td colspan="2">Tam. da msg.</td><td>r/w</td><td colspan="5">Preenchimento (padding)</td></tr>
<tr><td colspan="8">Carga útil</td></tr>
</table>

Tabela 3.1 Estrutura utilizada para a comunicação núcleo-usuário

3.2.3 Interface de gerência

Como indicado na seção 3.1, o módulo monitora apenas processos previamente selecionados. Para saber quais serão monitorados, é necessário um canal de comunicação entre o módulo e o usuário que está utilizando o módulo. Este canal deve permitir ao usuário selecionar quais processos devem ser monitorados. Por este motivo, o módulo disponibiliza no diretório */proc* o arquivo *interceptor*, que, ao ser lido, reporta o status do seu funcionamento, como demostrado na figura 3.13.

```
# cat /proc/interceptor
Flags: [0000000000000026] HASH_INIT FIFO_INIT SYSCALL_HOOKED
Monitored PIDs:
```

Figura 3.13 Status do funcionamento do módulo

Quando houver um evento de escrita no arquivo *interceptor*, o módulo interpreta a sequência de caracteres enviados a procura por comandos. Por exemplo, para requisitar que o módulo monitore determinado identificador de processo, devemos gravar no arquivo a sequência de caracteres "[número do identificador]" (identificador do processo precedido e terminado por colchetes), como demostrado na figura 3.14.

```
# echo '[1234]' > /proc/interceptor
# cat /proc/interceptor
Flags: [0000000000000026] HASH_INIT FIFO_INIT SYSCALL_HOOKED
Monitored PIDs: 1234
```

Figura 3.14 Comando para monitorar um processo

Ao enviar o comando para o módulo monitorar um processo, requisitando, na sequência, seu status de funcionamento, é possível verificar que o identificador deste processo aparece listado como um processo que deve a ser monitorado. O processo pode ser retirado da lista de monitoramento reenviado o mesmo comando de requisição de monitoramento, como demostrado na figura 3.15.

```
# echo '[1234]' > /proc/interceptor
# cat /proc/interceptor
Flags: [0000000000000026] HASH_INIT FIFO_INIT SYSCALL_HOOKED
Monitored PIDs:
```

Figura 3.15 Comando para remover monitoração de um processo

Além do comando de requisição de monitoramento, a interface de gerência reconhece os comandos listados a seguir:

- **parent**: Força a interceptação de todos os processos cujo pai esteja cadastrado na lista de monitoração
- **hijack**: Habilita o desvio das chamadas de leitura e escrita independente de haver um software lendo o dispositivo registrado no diretório */dev*, serve para preencher o buffer circular com o propósito de depuração.
- **debug**: Habilita mensagens extras de depuração.

Um exemplo do uso do comando "*parent*" aparece na figura 3.16, onde também é possível verificar que "*parent*" está habilitada, observando que a *flag* SEEK_PARENT está ativa.

```
# cat /proc/interceptor
Flags: [0000000000000026] HASH_INIT FIFO_INIT SYSCALL_HOOKED
Monitored PIDs: 1234

# echo 'parent' > /proc/interceptor
# cat /proc/interceptor
Flags: [000000000000002e] HASH_INIT FIFO_INIT SEEK_PARENT SYSCALL_HOOKED
Monitored PIDs: 1234
```

Figura 3.16 Incluindo processos parentes na monitoração

3.3 Formato das mensagens de instrumentação

Esta camada, diferente da mostrada na seção 3.2, é executada por um software de usuário. Este software é responsável por ler o dispositivo de caractere registrado pelo módulo, usando a estrutura descrita na tabela 3.1, e converter essa informação para o formato *libpcap,* padronizado pela biblioteca de mesmo nome. A motivação para esta conversão será apresentada na seção 3.4.

Originalmente utilizado pelo *tcpdump* [44], um analisador de pacotes de rede, a *libpcap* é a biblioteca que se tornou o padrão "de facto" para ferramentas de cap-

tura de pacotes de rede em ambientes *Unix* e assemelhados, e se tornou o "denominador comum" das ferramentas de captura no mundo do software livre [45]. O padrão segue um formato bem simples, como demonstrado na figura 3.17.

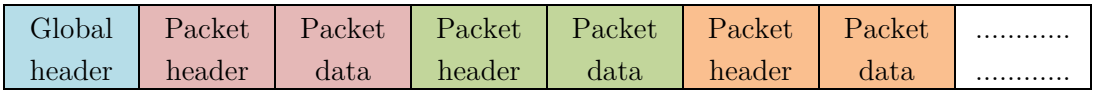


Figura 3.17 Cabeçalho utilizado pelo formato *libpcap*

O início do arquivo conta com um cabeçalho global *(global header)*, descrito na tabela 3.2, seguido da sequência de pacotes capturados. Cada pacote é precedido por um cabeçalho *(packet header)*, descrito na tabela 3.3, cujo formato é especificado no cabeçalho global, seguido da carga útil *(payload)* do pacote *(packet data)*.

Disposição dos bits			
8	16	24	32
Valor mágico do formato			
Valor superior da versão		Valor inferior da versão	
Correção de fuso horário			
Acurácia dos marcadores de tempo			
Tamanho da mensagem			
Protocolo utilizado pelas mensagens			

Tabela 3.2 Cabeçalho global utilizado pela *libpcap*

Disposição dos bits			
8	16	24	32
Timestamp (segundos)			
Timestamp (microsegundos)			
Quantidade de octetos salvos no arquivo			
Quantidade de octetos do pacote			

Tabela 3.3 Cabeçalho de cada pacote armazenado no arquivo *libpcap*

Os campos valor mágico, valor superior e inferior de versão, correspondem a 0xA1B2C3D4 (valor mágico[6] correspondente a um arquivo *libpcap*), 2 e 4 (última versão do formato), respectivamente. Foram atribuídos, para os campos correção de fuso horário e acurácia do marcador de tempo, o valor 0, por não haver distorções no horário corresponde ao momento em que o evento ocorreu e a interceptação do evento. Para o tamanho da mensagem, foi atribuído o valor máximo permitido pelo formato, que corresponde a 65535 bytes. Já o valor do protocolo corresponde a 101, a razão para este valor será fornecido mais adiante.

Por ser um formato utilizado por analisadores de pacotes, é esperado que os pacotes contassem com a camada de enlace de dados *(network layer)* e a camada física *(data link layer)*. Entretanto, a interceptação é realizada na camada de apli-

[6] Em *Unix*, é comum arquivos terem identificadores exclusivos nos primeiros 32-bits

cação *(application layer)*, onde as informações de tais camadas ainda não estão disponíveis, como pode ser avaliado no processo de encapsulamento demonstrado na figura 3.18.

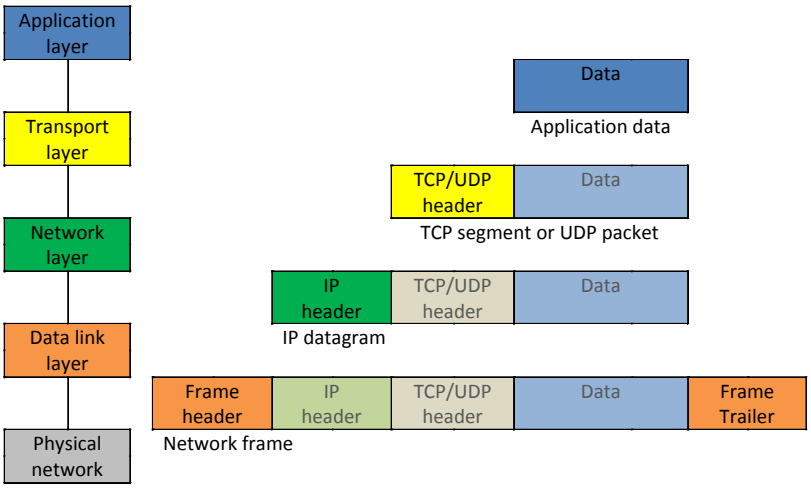


Figura 3.18 O processo de encapsulamento de um pacote TCP/IP

Assim, para manter a compatibilidade com a *libpcap*, faz-se necessário sintetizar, artificialmente, um pacote compatível. Entre os tipos de mensagem suportados pela *libpcap*, e com base nas informações disponibilizadas pela monitoração, o tipo mais indicado é o mesmo utilizado para pacotes IPv4 ou IPv6 brutos, sem o encapsulamento físico. A fundamentação desta decisão vem do fato de existir as informações necessárias para sintetizar o enlace de dados TCP/IP, mas não há informações suficientes para sintetizar a camada física. Por esse motivo o valor do tipo de mensagem no cabeçalho global é 101, por corresponder a pacotes IPv4 ou IPv6 brutos.

Dessa forma, descartando a camada física e recriando artificialmente o pacote TCP/IP com base em inferências e informações já existentes, podemos montar um pacote similar ao apresentado na tabela 3.4, onde os campos nulos são deixados em branco, gravando zeros nessas posições. Já os campos que podem ser inferidos, correspondem aos campos que podemos preencher com base nas inferências a seguir:

- **Versão**: A utilização da versão 4 do protocolo IP é a mais comum e a única versão compatível com endereços IP de 32-bits.
- **IHL**: É o tamanho do cabeçalho em palavras de 32-bits. O menor cabeçalho IP possível tem IHL igual a 5, crescendo conforme a utilização do campo opcional IP.
- **Tamanho total**: Informação obtida pela soma dos tamanhos da carga útil da mensagem, e dos cabeçalhos IP e TCP.
- **TTL**: Atribuído o valor 255, mas poderia ser qualquer valor diferente de 0.
- **Protocolo**: Atribuído o valor 6, que corresponde ao protocolo TCP. Este valor foi escolhido por serem poucos os programas que trocam informações usando um protocolo diferente do TCP.

- **Número de sequência**: Equivale à quantidade de bytes gravados, por descritor de arquivos, por processo. O número de sequência é utilizado para remontar a ordem dos pacotes.
- **Deslocamento**: Tamanho do cabeçalho TCP em palavras de 32-bits. O menor cabeçalho TCP possível tem deslocamento 5.
- **Tamanho da janela**: Atribuído o valor de 8192, mas poderia ser qualquer valor diferente de zero.
- **Verificação**: A verificação TCP deve ser calculada seguindo o algoritmo de *checksum* do protocolo TCP. Ao contrário da verificação TCP, a verificação IP pode ser preenchida por zeros, correspondendo ao cálculo do *checksum* realizado fora do núcleo, por equipamento especializado.

Disposição dos bits							
0-3	4-7	8-11	12-15	16-19	20-23	24-27	28-31
Versão	IHL	DSCP	ECN	Tamanho total			
Identificação				Sinal.	Deslocamento IP		
TTL		Protocolo		Verificação IP			
Endereço IP de origem							
Endereço IP de destino							
Opcionais (se IHL > 5)							
Porte de origem				Porta de destino			
Número de sequência							
Número de reconhecimento							
Desloc.	Reserv.	Mapa de bits		Tamanho da Janela			
Verificação				Urgencia (Habilitado se bit URG)			
Opcionais (Habilitado se deslocamento > 5)							
Carga útil							

Legenda: Disponível | Inferido | Nulo

Tabela 3.4 Estrutura de um pacote TCP/IP

A monitoração ainda nos fornece informações extras, como os identificadores do processo (atual e pai) e descritor do arquivo que gerou o evento, que não são previstas em um pacote IP ou TCP padrão. Por esse motivo, o cabeçalho IP opcional é utilizado para armazenar tais informações, lá é previsto que até 40 bytes de informações podem ser guardadas, seguindo a estrutura da tabela 3.5.

Disposição dos bits				
0	1-2	3-7	8-15	16-...
Copia	Classe	Número	Tamanho	Dados

Tabela 3.5 Estrutura de um cabeçalho IP opcional

Dessa forma, os identificadores de processos (corrente e pai) e descritor de arquivos são armazenados dentro do cabeçalho IP, completando a utilização de todas as informações obtidas pela monitoração, sem prejudicar a compatibilidade com a estrutura do pacote TCP/IP ou com o formato *libpcap*.

É importante observar que as informações gravadas no cabeçalho opcional corresponderiam a cinco inteiros de 32-bits, ou 20 bytes, gastos por pacote. Como na maioria das vezes esses inteiros são utilizados para representar valores pequenos, a utilização de um método de compressão de bits corresponde a uma boa prática para reduzir o espaço consumido por essa informação. A codificação BER (Representação de Codificação Binária ou *Binary Encoding Representation*) foi utilizada para reduzir o consumo do espaço utilizado para, em média, 9 bytes. A codificação BER procura diminuir a quantidade de bits necessários para representar um inteiro, reduzindo a redundância presente em valores de baixa ordem, onde apenas os bits menos significativos são utilizados. A figura 3.19 mostra a utilização do algoritmo.

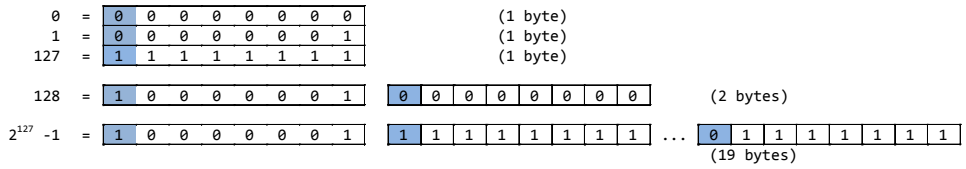


Figura 3.19 Binary Encoding Representation

O algoritmo diz que se o primeiro bit do byte lido for igual à zero, o inteiro pode ser representado por este byte apenas. Se o primeiro bit for de valor um, o valor do byte corrente deve ser deslocado de 7 bits e acrescido do byte subsequente. Se o primeiro bit do byte subsequente for igual a um, o primeiro byte deve ser deslocado de 14 bits, o byte seguinte de 7 bits e somado ao byte atual.

Seguindo esse raciocínio, se um inteiro de 32-bits fosse completamente preenchido com bits de valor 1, sua representação BER ocuparia 5 bytes ao invés de 4 bytes, que seria o espaço ocupado pela representação regular de um inteiro sem compressão. Por esse motivo, BER é recomendado apenas para inteiros que possam ser representados apenas por seus bits menos significativos. Dessa forma, com a utilização da compressão, a representação das informações pelo cabeçalho opcional IP seguem o formato descrito na tabela 3.6.

Disposição dos bits		
0-7	8-15	16-...
0xFF	Tamanho	Processo atual, pai, descritor, ...

Tabela 3.6 Campo opcional IP com dados de processo e descritor

3.4 GUI com o usuário

O uso de arquivos de captura no formato *libpcap*, como descrito na seção 3.3, nos permite utilizar ferramentas de análise de rede com o objetivo de analisar eventos de leitura e escrita, de agora em diante denominado eventos de E/S (entrada e saída). Como os eventos foram armazenados como pacotes, o mesmo conceito de analisar pacotes pode ser aplicado na análise dos eventos.

Ferramentas de análise de rede são ferramentas que decodificam pacotes contendo os protocolos mais comuns e os exibem de uma maneira legível, por isso são utilizadas para monitorar o fluxo de informações que trafegam por uma rede de computadores. A diferenciação entre os analisadores de rede está, principalmente, na quantidade de protocolos que são capazes de decodificar e por suas habilidades especiais. [46]

Entre os analisadores mais conhecidos, o *wireshark* [47] merece especial atenção por ser considerado o melhor da categoria. Seu reconhecimento se dá por ser capaz de inspecionar centenas de protocolos, ser desenvolvido como software livre e coberto pela licença GPL v2 [48], funcionar em *Unix* e assemelhados, Windows e Mac OS, ter interface gráfica intuitiva, e também por ser ativamente desenvolvido e mantido.

Além das características já existentes, é possível utilizar sua arquitetura de *plug-ins*, que são códigos escritos por terceiros, que tem como objetivo adicionar funcionalidades não disponíveis originalmente no software. A apresentação da interface pode ser avaliada na figura 3.20.

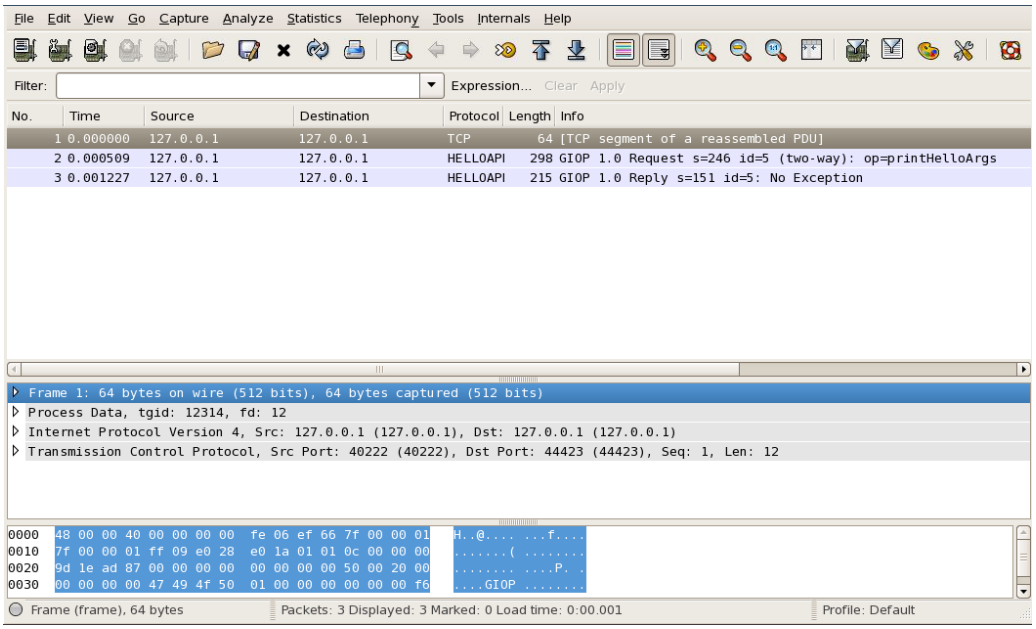


Figura 3.20 A interface do *wireshark*

O *wireshark* possui três janelas principais, a janela de resumo, onde são apresentados os pacotes em ordem cronológica, a janela contendo a árvore de decodifi-

cação do protocolo, onde o pacote selecionado na janela de resumo é decodificado e representado por uma árvore de protocolos. Nessa árvore, cada folha é atribuída para cada camada decodificada do pacote. Na janela de visualização de dados, é exibido o conteúdo bruto do item selecionado, em formato hexadecimal e em texto puro. Se, no pacote corrente, uma folha for selecionada, a representação das informações correspondentes a esta folha são realçadas na janela de visualização de dados.

Observando a figura 3.21, é possível verificar que as informações de identificação do processo e descritor de arquivos, embutidas no pacote TCP/IP, descrito na seção 3.3, estão com pouco destaque na interface, pois para visualizar tal informação é necessário: abrir a folha "Internet Procotol", em seguida, abrir a folha "Options", e observar, na janela de visualização de dados, a informação bruta, não legível, por estar sem a decodificação apropriada.

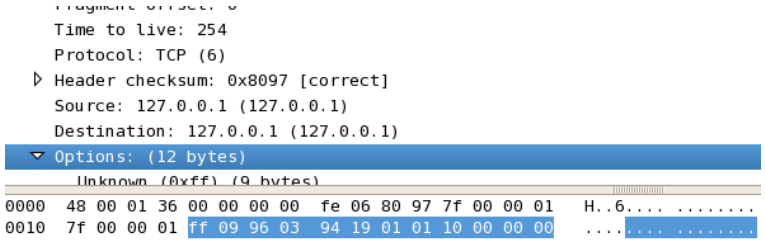


Figura 3.21 Campo opcional IP com as informações de processo

Para melhorar a apresentação dessa informação, foi desenvolvida, utilizando a arquitetura de *plug-ins*, uma extensão que altera o que é exibido na árvore de decodificação de protocolo. Esta, como consta na figura 3.22, realiza a decodificação BER necessária para a visualização das informações contidas no campo opcional IP e a desloca do para uma folha mais próxima a raiz da árvore.

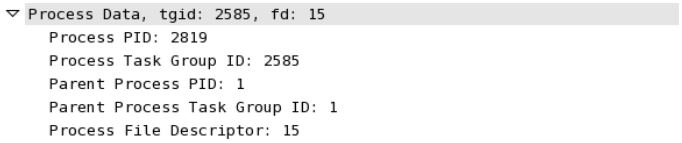


Figura 3.22 Campo opcional IP manipulado pela extensão

A respeito da procura por defeitos, o *wireshark* conta com uma ferramenta de filtros de pacotes capaz de reduzir a quantidade de informações que são exibidas na janela de resumo, com base em características dos pacotes que se enquadram nos critérios selecionados pelo filtro. Tal ferramenta, disponível no menu de acesso rápido, é acessível através da entrada "Follow TCP Stream", como consta na figura 3.23.

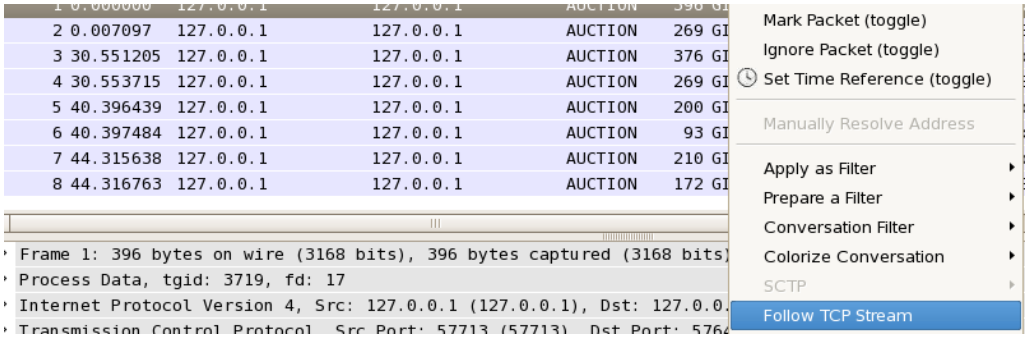


Figura 3.23 Acesso ao filtro rápido do Wireshark

Este acesso rápido aplica, automaticamente, um filtro que seleciona apenas os pacotes TCP que se relacionam com o pacote onde o acesso rápido foi chamado, exibindo em uma nova janela, similar a mostrada na figura 3.24, com as trocas de mensagens a qual o pacote selecionado faz parte.

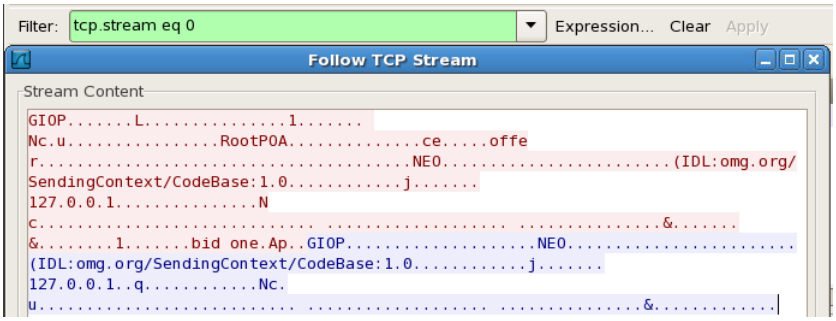


Figura 3.24 Resultado da aplicação do filtro

Este filtro tem especial importância por mostrar toda a interação entre os dois processos que fizeram parte da troca de mensagens, podendo dar indícios sobre o que ocorreu entre eles. Por exemplo, supondo que ao invocar um método de um componente hipotético, houvesse um problema ao passar um de seus parâmetros. Dessa forma, supondo que o componente responda ao invocador do método com uma exceção contendo como atributo um campo de texto descrevendo o erro, esse texto poderá ser observado pela ferramenta independende da forma como o invocador do método tratar tal excessão.

De forma a facilitar a visualização dos dados, o *wireshark*, através da árvore de decodificação, realiza uma decodificação por camadas. Primeiro realiza a decodificação da camada de rede, decodificando o protocolo IP e gerando uma folha, em seguida, a camada de transporte, decodificando o protocolo TCP e gerando uma nova folha, e por último, a camada de apresentação, decodificando o protocolo utilizado na comunicação entre os compoentes do software. Se o protocolo for reconhecido, novas folhas serão adicionadas. A figura 3.25 mostra a árvore de decodificação de um pacote GIOP (General Inter-ORB Protocol), advindo da comunicação de um software utilizando o ORB (Object Request Broker) do *middleware* CORBA.

Nativamente, o *wireshark* decodifica o protocolo GIOP, porém sem os detalhes específicos do software que montou o pacote, por exemplo, sem distinguir

os métodos e parâmetros do restante da representação hexadecimal do pacote, pois tais detalhes só podem ser obtidos se houver acesso a IDL[7] (*interface description language* ou linguagem descritora de interface) utilizada pelo software. A figura 3.26 mostra o mesmo pacote da figura 3.25, porém com auxílio da IDL, é possível ver os dados inseridos pela nova folha agregada a ávore de protocolos, com a representação mais compreensível do pacote GIOP.

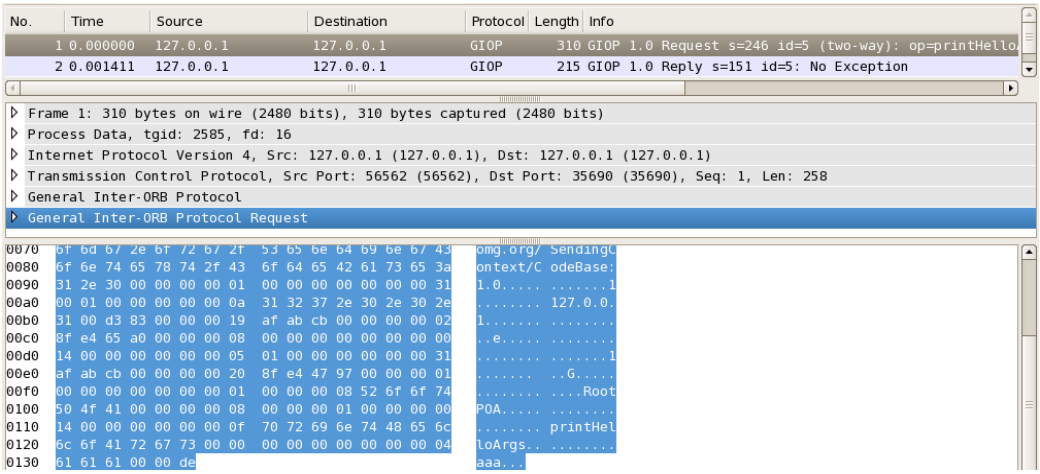


Figura 3.25 Interpretação nativa do protocolo GIOP

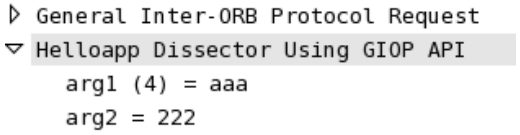


Figura 3.26 Uso da IDL para obter mais informações do pacote

De modo a carregar uma IDL no *wireshark*, com o auxílio de scripts disponibilizado pelo próprio, é necessário que antes esta passe por um processo de adaptação para o formato de extensão. A carga desta extensão faz com que informações extras possam ser exibidas em uma nova folha. Além disso, tal extensão habilita novas fontes de dados que podem ser utilizadas para elaborar filtros mais complexos, e assim, permitir uma melhor filtragem de pacotes.

A figura 3.27 mostra a utilização da fonte de dados *giop-helloapp*, que foi habilitada pela carga da IDL do software no *wireshark,* de forma a fazer uma seleção mais criteriosa do que é exibido na janela de resumo. Na figura, como exemplo, é relacionado o descritor de arquivos utilizado pelo processo que gerou o evento com um parâmetro de um método invocado pela software.

Em resumo, utilizando as características avançadas de decodificação de protocolos, aliado ao acompanhamento da troca de mensagens relacionada a um determinado pacote, e a utilização de filtros usando quaisquer características dos protocolos, tornam o *wireshark* uma ferramenta capaz relacionar eventos com condições de erros, mostrando ao programador qual interação, com qual componente,

[7] Linguagem que descreve a interface utilizada por um componente de software.

pode ter ocasionado o defeito. Assim, através da visualização da interação com o componente defeituoso, o programador pode ser capaz de recriar e reproduzir as condições que levaram o componente ao erro.

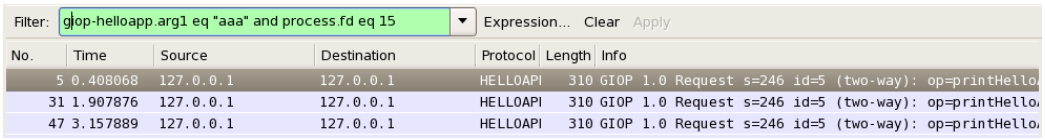


Figura 3.27 Acesso ao filtro rápido no Wireshark

3.5 Metodologias de utilização

As metologias de uso apresentadas nesta seção auxiliam a diminuição da quantidade de informações geradas, e consequentemente, o volume de dados que devem ser analisados para a identificação da causa de um defeito. Tais informações são ditas desnecessárias por terem sido geradas de forma redundante.

Para uma melhor compreensão da ferramenta descrita por este trabalho, sabe-se que esta é dividida em três camadas: a monitoração, a transformação e a análise. A monitoração e a transformação trabalham em conjunto e por esse motivo não podem ser separadas. A análise, diferentemente, pode ser realizada fora do contexto onde está sendo realizada a monitoração, não havendo restrições sobre o local onde a análise é feita. A monitoração dos processos, por ser realizada pelo núcleo do sistema operacional, deve ser instalada nos servidores onde os componentes estão sendo executados, com isso, a monitoração terá acesso a todas as mensagens ingressas e egressas destas máquinas. Assim, dependendo da topologia de comunicação utilizada entre os componentes, pode ser aplicada a metodologia de depuração *top-down* ou *bottom-up*.

A figura 3.28 mostra dois componentes monitorados pela ferramenta, sendo executados por servidores distintos. Supondo que exista uma troca de mensagens entre eles, e que no instante que o componente **A** enviar uma mensagem ao componente **B**, **B** responda a mensagem a **A**. Esta distribuição, por haver comunicação apenas entre dois componentes, é a ideal para a utilização do filtro de acesso rápido apresentado na seção 3.5. Nota-se que este filtro não foi idealizado para arquiteturas mais complexas, envolvendo a comunicação entre mais de dois componentes. Para estes casos, a elaboração de um filtro mais adequado à arquitetura deverá ser planejada pelo administrador da infraestrutura onde os componentes estiverem sendo executados, por este possuir os conhecimentos necessários de como obter os endereços e portas de onde os componentes estiverem instalados.

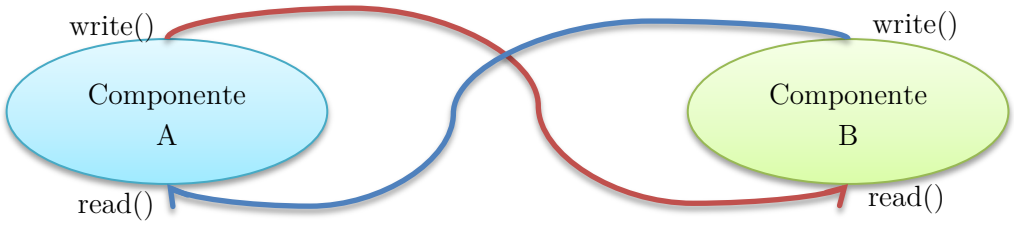


Figura 3.28 Elos de comunicação entre componentes A e B

A monitoração dos componentes **A** e **B** resultaria, respectivamente, nas tabelas 3.7 e 3.8. Podemos observar que as informações contidas nestas tabelas são análogas, apenas com o ponto de vista invertido. O componente **A** "enxerga" a operação de escrita de **B** como leitura e vice versa. Seria suficiente, neste caso, a monitoração estar ativa em apenas um dos componentes. A posição de qualquer um destes componente na topologia de comunicação do software é suficiente para observar toda comunicação realizada entre os componetes **A** e **B**.

IP e porta de origem	IP e porta de destino	OP	Mensagem
10.0.0.1:1001	10.0.0.2:1002	W	Mensagem A→B
10.0.0.2:1002	10.0.0.1:1001	R	Mensagem B→A

Tabela 3.7 Mensagens monitoradas pelo componente A

IP e porta de origem	IP e porta de destino	OP	Mensagem
10.0.0.2:1002	10.0.0.1:1001	R	Mensagem A→B
10.0.0.1:1001	10.0.0.2:1002	W	Mensagem B→A

Tabela 3.8 Mensagens monitoradas pelo componente B

Dentre as topologias de comunicação conhecidas (ponto a ponto, barramento, estrela, anel, malha conectada, malha parcialmente conectada, híbrida e conexão em cadeia [49]), destacamos, na figura 3.29, as aplicáveis a interconexão entre componentes. Em uma topologia estrela, o componente central da estrela centraliza toda a comunicação, sendo possível a instalação da monitoração apenas no servidor onde está instalado este componente, categorizando a metodologia *top-down* para a depuração do software distribuído. Em uma topologia de barramento ou malha completamente conectada, não existe componente centralizador, logo, é necessário a instalação da monitoração em todos os servidores que hospedem componentes, categorizando a metodologia *bottom-up*.

Já em uma topologia em anel, árvore ou malha parcialmente conectada, por não ser uma topologia nem tão simples quanto a topologia em estrela e nem tão complica quanto uma malha completamente conectada, requer um maior esforço na identificação de qual posição seria mais adequada a instalação da ferramenta. Devido a natureza híbrida dessas topologias, a comunicação entre componentes pode transcorrer por caminhos distintos e nem sempre determinísticos, podendo, por exemplo, haver um balanceador de carga e a requisição percorrer o caminho onde há menor utilização dos componentes. Para o caso descrito, a instalação da

ferramenta em todos os nós seria o procedimento adequando, sendo assim necessário lidar com a redundância posteriormente.

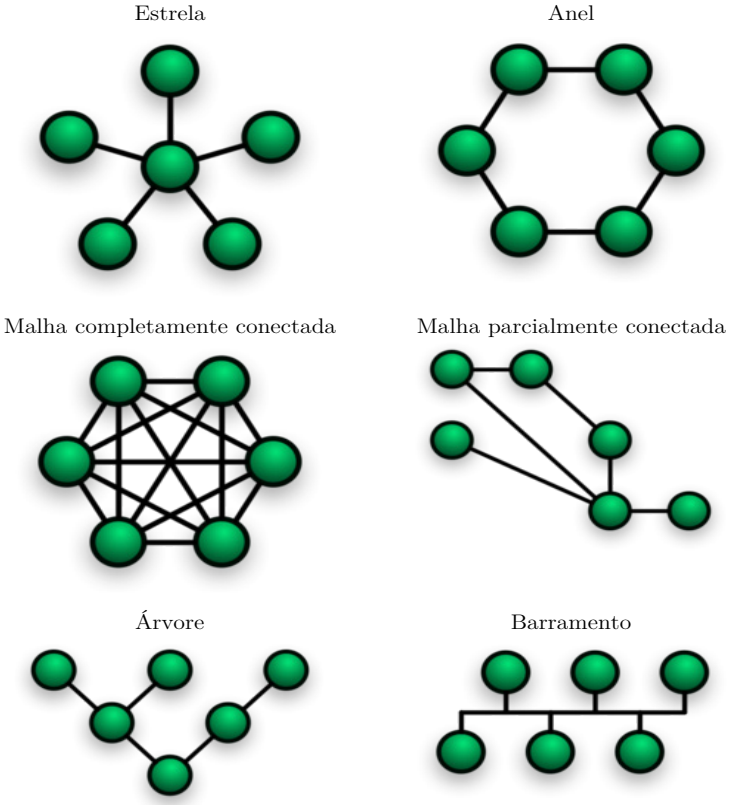


Figura 3.29 Topologias de comunicação

Ao espalhar a instalação da ferramenta em mais de um servidor, característica da metodologia *bottom-up*, os arquivos *libpcap* contendo a troca de mensagens entre componentes ficam hospedados nos sistemas de arquivos desses servidores, e consequentemente, espalhados. Mesmo sendo possível analisar os arquivos um a um, talvez este tipo de análise não seja suficiente para o objetivo esperado, por ser necessário visualizar o funcionamento do software como um todo.

Apesar da ferramenta descrita por este trabalho ter sido inicialmente projetada para a utilização por pares cliente-servidor, esta não foi projetada para uso por mais que um par máquinas simultaneamente (quando cliente e servidor estiverem em máquinas distintas), porém, tal limitação não é impedimento para sua utilização no caso descrito no parágrafo anterior. Há, no *wireshark*, ferramentas que podem ser utilizadas para juntar esses arquivos de modo a contornar tal limitação, porém, sem a reordenação causal necessária para a correta interpretação da troca de mensagens que foram armazenadas pelas múltiplas máquinas. A reordenação causal dessas mensagens só seria possível se houvesse uma melhor avaliação do relacionamento das informações contidas dentro dos pacotes armazenados por esses arquivos.

A motivação para esse cuidado se deve ao fato de ser necessário garantir que a requisição enviada por um componente, na reordenação, apareça antes da resposta a esta requisição. O não cumprimento dessa sequência implicaria na quebra da causalidade, e consequentemente, na impossibilidade do analisador de pacotes de decodificar as trocas de mensagens subsequentes. Por este motivo, pode não ser suficiente, para a reordenação causal dessas mensagens, se basear apenas no horário quando o evento foi realizado.

Dessa forma, de maneira análoga ao trabalho realizado por França [13] em sua dissertação de mestrado, a utilização das informações adicionais contidas nos pacotes (identificadores do processo, *threads* e descritor do arquivo que gerou o evento) devem ser consideradas para a reconstrução da causalidade das mensagens. Acredita-se que só assim seja possível recompor a ordem de troca das mensagens que foram fracionadas, devido ao uso da metodologia *bottom-up*, em arquivos separados.