NetRevealer: Uma Ferramenta Gráfica para a Análise do Tráfego de Redes

Saulo Queiroz da Fonseca[1], Henrique São Mamede [2]

[1] Estudante Univ. Aberta, [2] Univ. Aberta, INESC-TEC
email@saulofonseca.de, jose.mamede@uab.pt

Resumo

O problema da falta de uma ferramenta gráfica de análise de redes em tempo real levou ao trabalho de desenvolvimento de uma aplicação que colmatasse esta lacuna. Desta forma, desenvolveu-se o NetRevealer, uma ferramenta multi-plataforma que tem como objetivo analisar o tráfego numa rede de computadores. Para isso, a aplicação acede às interfaces de rede existentes num computador e representa através de um ícone cada equipamento que envia ou recebe pacotes nessa rede. Obtém-se assim um mapa que exibe o tráfego em tempo real e permite detectar atividades nessa rede, nomeadamente atividades indesejadas como portas clandestinas e utilizadores não autorizados a utilizar a rede.

palavras-chave: Rede de Computadores, Tráfego de Rede, Análise de Pacotes, Mapa da Rede

Title: NetRevealer: A graphical tools for analysing network traffic

Abstract

The need of a graphical tool for network analysis in real time is the reason of the development of an application to fulfil this gap. Thus, we developed the NetRevealer, a cross-platform tool that aims to analyze traffic on a computer network. The application accesses the existing network interfaces on a computer and uses an icon to represent each device that sends or receives packets on this network. The application tool generates a map that display in real-time the data traffic and allows to detect the activity in this network, including undesirable activities such backdoors and users allowed to use the network.

keywords: Computer Network, Netwrok Traffic, Packet Analysis, Network Mapping

1. Introdução

Numa rede de computadores, é difícil determinar com rigor como está a decorrer, em cada instante, o fluxo de dados entre os equipamentos envolvidos. Esta dificuldade deve-se ao fato de que o nível técnico destas informações é alto e pouco acessível ao público não especializado [1]. No momento em que surge uma anomalia, é fundamental ter disponível uma ferramenta que revele estas atividades, de forma a indicar quem está a enviar pacotes a quem.

Existem algumas ferramentas no mercado que prometem suprir esta carência, mas todas falham na concretização desse objetivo por restringir a visualização dos dados, concentrando-se exclusivamente em exibir o conteúdo dos pacotes ou em exibir um mapa da rede independente do tráfego local.

A solução encontrada passou pela construção de um produto de *software* capaz de ler os pacotes que fluem nas interfaces de rede a partir de um determinado computador, construindo um mapa da mesma a partir desse ponto. As informações presentes nos cabeçalhos dos pacotes, como endereço de origem e de destino, são utilizadas para criar um ícone para cada *host*. Uma linha é desenhada entre os ícones correspondentes, de forma a representar o tipo de pacote enviado. Com o passar do tempo, o desenho irá refletir todos os *hosts* ativos na rede, formando-se um mapa da mesma.

Estas informações permitem um diagnóstico em tempo real das atividades presentes nessa rede, oferecendo os elementos para uma deteção rápida de qualquer anomalia e contribuindo para aumentar a produtividade de quem estiver envolvido na resolução do problema.

O artigo está organizado em quatro secções. Na Seccão 2 é apresentado o enquadramento do trabalho relativo a interfaces, API e diagrama de classes. Na Secção 3, o trabalho é desenvolvido tendo em consideração, a leitura e interpretação dos pacotes, bem como a definição e mapeamento dos *hosts*. Finalmente, na Secção 4 são apresentadas as conclusões.

2. Enquadramento

2.1 Definição da interface com o utilizador

O primeiro passo foi decidir o ambiente de interface para a aplicação, tendo-se definido que a mesma seria exibida num ambiente de janelas, com a janela principal a ser dividida em duas partes. A parte superior contém o mapa da rede e a parte inferior contém o *log* dos pacotes lidos. A linha que divide as duas partes pode ser movida, de forma a aumentar/diminuir o tamanho de cada uma delas. Na Figura 1 ilustra-se o aspeto gráfico da aplicação.

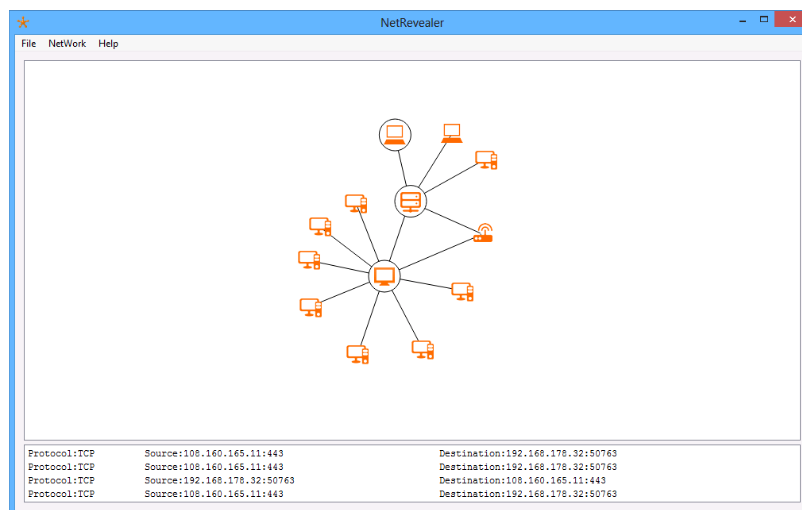


Figura 1 - Interface para o *NetRevealer*

2.2 Escolha da API para acesso ao *hardware*

A aplicação necessita de aceder às placas de rede no computador em que se estiver a executar, de forma a monitorizar os pacotes de dados que estão a circular pela mesma.

Diferentemente de um uso comum de uma placa de rede, onde um programa se conecta a um servidor e troca dados com o mesmo, o *NetRevealer* lê os pacotes criados pelos outros programas, sem ele mesmo gerar pacotes novos. Para isso, tem que modificar o modo como a placa de rede opera, com a necessidade de ativar o modo denominado de promíscuo [2] (*promiscuous mode*).

É importante salientar que a aplicação poderá apenas ler dados em pacotes que chegam à placa de rede em questão. Dois computadores presentes na rede local, que enviam pacotes entre si, sem passar pela referida placa, não serão lidos pelo programa. A vantagem deste modo promíscuo surge se ele estiver instalado num computador que faz o papel de roteador da rede, por onde passa o tráfego local ou ligado através de uma porta que faz o *mirror* de um roteador [3]. É nestes pontos onde normalmente o administrador de redes faz o seu trabalho de monitorização [4].

As principais APIs de comunicação com as placas de rede são *Winsocks* da *Microsoft* e *Libpcap* criado pelo grupo *TCPDump* [5]. Ambas permitem ativar o modo promíscuo das placas de rede. Estas APIs podem ser descritas de forma sumária, da seguinte forma:

- *Winsocks* é a API padrão para aceder ao ambiente de redes em aplicativos do sistema operativo *Windows*. Opera no *layer* 3 do modelo OSI [6], ou seja, permite ler pacotes que já foram processados pela placa de rede (*layer* 1) e pelo protocolo *Ethernet* (*layer* 2).
- O *Libpcap* foi desenvolvido por um grupo de *software* livre, originalmente para o sistema operativo *Linux*, mas hoje também disponível para *Windows* e *Mac* [7]. Esta API possui a vantagem de ler os pacotes ainda no *layer* 2, o que permite ver exatamente o que a placa de rede está a entregar. Com isso podem ler-se pacotes que utilizam o protocolo *Ethernet* e, portanto, os endereços de *hardware* (*MAC*

Addresses). Esta informação é importante para identificar a origem física do pacote. Por esta característica, esta API tornou-se o padrão entre quem desenvolve aplicativos de monitorização de redes de computadores [8].

Devido às características acima citadas, decidiu-se recorrer à utilização do *Libpcap* neste projeto.

2.3 Escolha da API para a interface gráfica

A decisão sobre a interface gráfica normalmente está ligada ao sistema operativo em que o programa irá funcionar. Na área de redes, existe um problema: apesar de o sistema operativo *Windows* dominar cerca de 80% dos computadores dos utilizadores comuns, o mundo dos servidores é muito mais diverso [9]. Eu gostaria de ter um programa que pudesse funcionar em qualquer um destes ambientes, inclusive no *MacOSX* da *Apple*.

Depois de pesquisar muito sobre este problema, eu descobri o *Qt-Project*. Ele é um *framework* disponível para os principais sistemas operativos, como *Windows, Linux e Mac*. A sua grande vantagem é que ele permite compilar o mesmo código C++ em vários sistemas operativos [10]. Com isso temos o melhor dos dois mundos: o código feito no *Qt* é compilado no sistema operativo local e se for enviado a outro sistema operativo também é compilado localmente.

Na Figura 2 podemos ver a IDE chamada *Qt-Creator*, onde é feito o desenvolvimento.

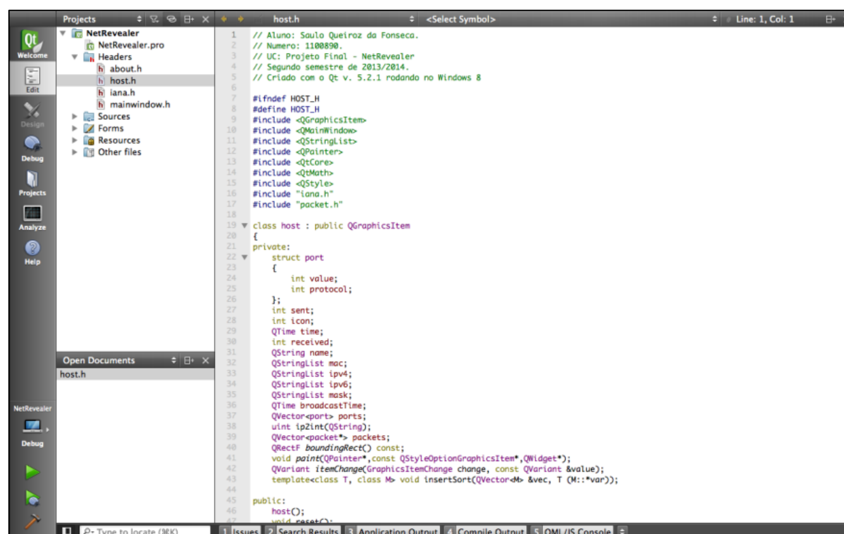


Figura 2 - Ambiente de programação do *Qt*

2.4 Definição das classes

A seguir foram definidos os atributos e métodos principais das classes que compõem o programa. A representação em UML pode ser vista na Figura 3.

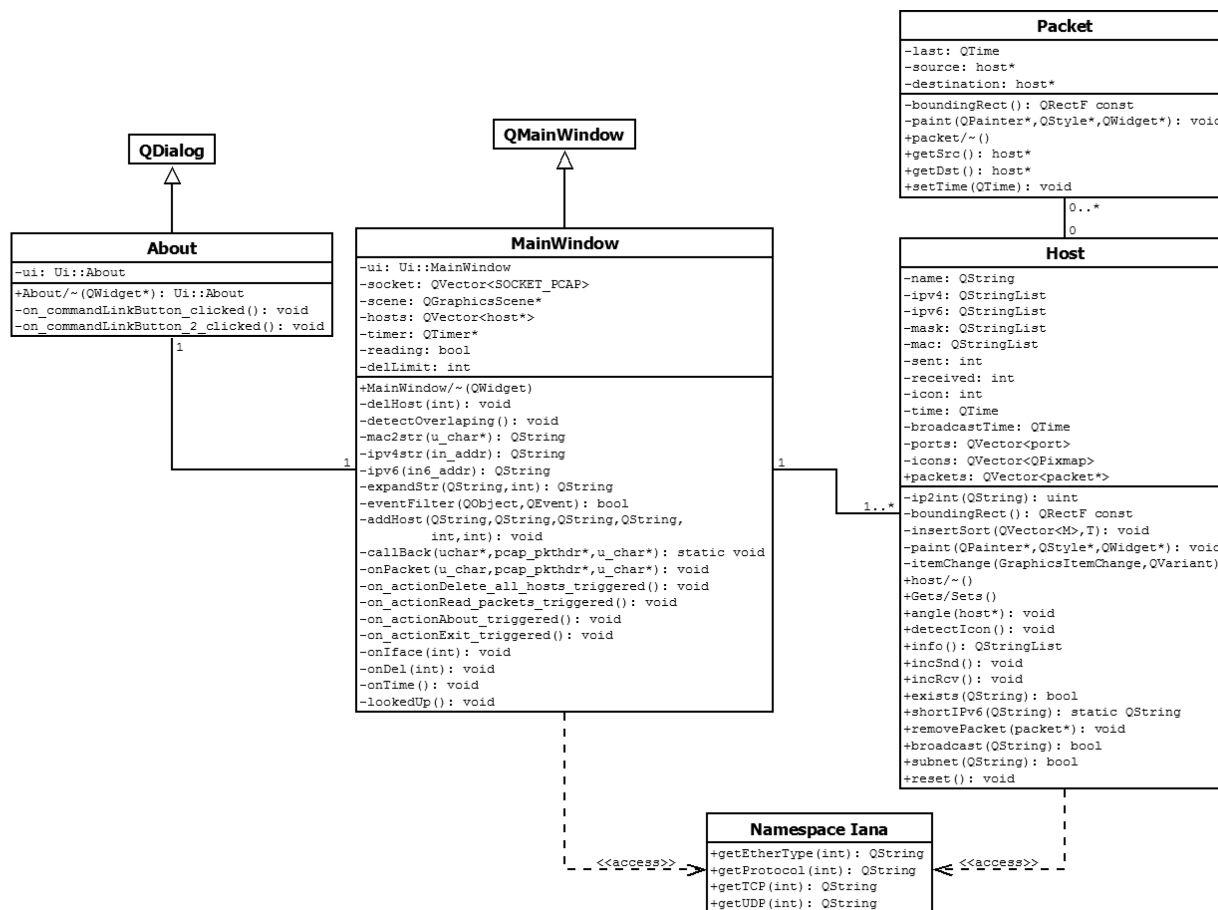


Figura 3 - Classes do programa

Para criar o ambiente de janelas, o *Qt* cria a classe *MainWindow*. Nesta classe eu defini as variáveis e métodos de uso geral.

Ao se aceder ao menu *Help* ⇒ *About*, surge uma pequena janela, definida pela classe de mesmo nome. Esta janela exibe a logomarca que eu criei para o programa, assim como o meu nome e dados para contato.

A classe *Host* é o coração do programa. É ela que irá representar os *hosts* detectados durante a leitura dos pacotes.

A classe *Packet* representa os pacotes enviados entre dois *hosts*.

Eu criei também um *namespace* chamado *Iana*. Esta é a sigla para *Internet Assigned Numbers Authority*, que é o órgão mundial para definição de endereços e protocolos usados na internet. Este *namespace* possui algumas funções onde, dado o número de um protocolo ou porta, retorna um pequeno texto descritivo sobre o mesmo. O ficheiro correspondente (*iana.cpp*) possui mais de 4500 linhas e seus dados foram obtidos a partir do endereço de internet da organização (http://www.iana.org). Os dados foram devidamente convertidos em uma *hashtable* usando o tipo *QHash* do *Qt*. Com isso, apesar de ser uma quantidade grande de informações, o acesso a um determinado índice é muito rápido.

3. Implementação

3.1 Leitura de pacotes

Este é o módulo que faz uso da biblioteca *Libpcap*, citada anteriormente. Depois de a instalação estar feita, precisamos apenas de adicionar "#include <pcap.h>" no *header* da classe *MainWindow*.

No construtor desta classe declaramos os comandos para iniciar o *Libpcap*. O primeiro passo é detectar as placas de rede presentes no computador (já que podemos ter mais de uma).

```
        pcap_if_t *alldevs;
        if (pcap_findalldevs(&alldevs,errbuf) == -1)
                QMessageBox::warning(this,"Error",
                        QString("Error in
pcap_findalldevs<br>")+errbuf);
```

A variável *alldevs* é definida como do tipo *pcap_if_t*, que é o formato usado para placas de rede pelo *Libpcap*. O comando *pcap_findalldevs* coloca em *alldevs* uma lista ligada contendo todas as placas encontradas. Se houver um erro, o *Qt* exibe uma caixa de mensagem.

Em seguida coloca-se cada placa encontrada em modo promíscuo.

```
        for (pcap_if_t *dev=alldevs; dev!=NULL; dev=dev->next)
        {
                // Open the NIC in promiscuous mode
                SOCKET_PCAP socket_temp;
                socket_temp.handle = pcap_open_live(dev-
>name,1518,true,
                        1000,errbuf);
                if (socket_temp.handle == NULL)
                        continue;
        ...
```

O tipo *SOCKET_PCAP* foi definido como um *struct* que contém duas variáveis. A primeira é do tipo *pcap_t* que define o ponteiro de ligação com o endereço de memória para aceder à placa. A segunda é do tipo booleano e será usada para permitir ligar ou desligar os pacotes vindos da mesma.

O comando *pcap_open_live* ativa a leitura da placa em questão. O atributo *name* é o nome detectado por *findalldevs*, 1518 é o tamanho máximo em bytes de um pacote no *layer* 2, *true* ativa o modo promíscuo, 1000 é tempo máximo em milissegundos que deve-se esperar por uma confirmação e *errbuf* é uma *string* onde será colocada uma mensagem de erro, caso a operação falhe. Caso isso ocorra, ignora-se o erro e usa-se o comando *continue* para tentar ativar a próxima placa de rede.

É importante salientar que algumas placas não permitem ativar o modo promíscuo [11]. Isto também pode ocorrer se o *Libpcap* não estiver corretamente instalado. Se o programa não conseguir ativar nenhuma placa, será exibida uma caixa de mensagem de alerta e o programa será encerrado.

Após os passos anteriores estarem concluídos, usa-se o seguinte comando, com os argumentos aqui em pseudo-código:

```
pcap_dispatch(ponteiro,quantidade,função,argumento);
```

O *ponteiro* é o que indica qual placa de rede deve ser usada, a *quantidade* é o número de pacotes a serem capturados, *função* é o nome da função definida pelo utilizador que deve processar os pacotes, e *argumento* é um argumento opcional que o utilizador pode passar à função informada, caso deseje.

Podemos perceber um problema. O *Libpcap* é feito em C, e por isso usa a lógica da programação funcional [12]. Ele não está preparado para lidar com objetos. Ele vai chamar uma função e ficar nela até processar todos os pacotes solicitados.

Este tipo de processamento não é amigável a um ambiente com janelas [13], pois o programa ficará "congelado" até o processamento estar pronto. Acrescento que o número de pacotes que eu desejo capturar não é fixo, e portanto colocar este comando em um *loop* infinito iria inviabilizar o projeto.

Os ambientes de janela funcionam através do processamento de sinais [14]. São declarados vários métodos, que respondem aos sinais emitidos. O ideal neste caso seria que o *Libpcap* emitisse um sinal quando recebe um pacote, que seria processado pelo método correspondente, mas isto não é possível.

A solução que eu encontrei foi usar um *timer*. No *Qt*, é possível criar um relógio que dispara um sinal com a frequência desejada. Isto é feito através do seguinte código, quando acrescentado ao construtor da classe *MainWindow*:

```
        timer = new QTimer(this);
        connect(timer,SIGNAL(timeout()),this,SLOT(onTime()));
        timer->start(50);
```

O primeiro comando inicializa a variável *timer*, que é declarada no *header* da classe *MainWindow*. Ela recebe como argumento a instância *MainWindow* onde se encontra (*this*). O segundo comando conecta esta variável de forma que, quando emitir seu sinal, será chamado o método *onTime()*. O terceiro comando informa que o relógio será disparado a cada 50 milissegundos.

Com isso eu pude criar o método *onTime()*, para chamar *pcap_dispatch*. Mas existe outro problema: a função declarada em *pcap_dispatch* deve ser comum ou estática, e não um método de uma classe. A solução que encontrei foi criar uma função *callback* que chama o método dinâmico a partir do estático:

```
static void callBack(u_char *args, const struct pcap_pkthdr *header,
                const u_char *packet)
        {
                ((MainWindow*)args)->onPacket(args,header,packet);
        }
```

Com isto o método *onTime()* passa a ter o seguinte formato:

```
void MainWindow::onTime()
{
        if (reading)
                for (int i=0; i<socket.size(); i++)
                        if (socket[i].active == true)
                                pcap_dispatch(socket[i].handle,1,

        MainWindow::callBack,(u_char*)this);
        ...
```

A variável *reading* é do tipo booleano que permite ou não a leitura dos pacotes. Ela é controlada por uma opção no menu. A variável *socket[i].active* informa se uma determinada placa de rede deve ser lida. Esta opção também é controlada por outro menu, que permite ao utilizador escolher quais placas devem estar ativas. O primeiro argumento do comando *pcap_dispatch* é o apontador para uma das placas de rede. O segundo é o número de pacotes que será lido, ou seja, apenas um. O terceiro é a função *callback*. O quarto é o endereço da instância de *MainWindow*, que será repassado à função *callback* como argumento. Em resumo temos a sequência da Figura 4.

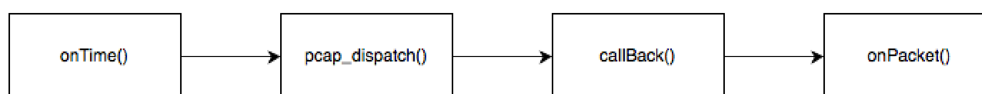


Figura 4 – Sequência de funções

Ou seja, o programa lê 1 pacote de cada placa de rede a cada 50 milisegundos. Isto permite o programa "respirar" nos intervalos de leitura entre pacotes e processar outras atividades, como menus, exibir *logs*, etc.

3.2 Interpretação dos pacotes

O *Libpcap* captura os dados no *layer* 2 (*data link*). Portanto, estes estão com todos os cabeçalhos presentes pertencentes aos *layers* superiores. A Figura 5 mostra os vários cabeçalhos que precisam ser removidos até se chegar aos dados originais.

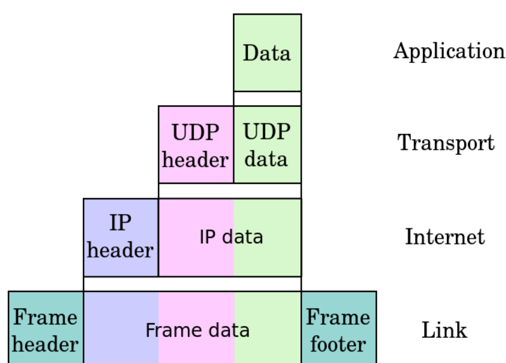


Figura 5 - Cabeçalhos presentes no layer 2

No caso do *NetRevealer*, o que nos interessa são justamente os cabeçalhos. Eles é que informam qual o endereço de origem e destino de cada pacote [15]. Os dados presentes dentro dos pacotes são portanto descartados pelo programa, já que são apenas uma cópia obtida através do modo promíscuo da placa de rede.

O método *onPacket()* é chamado quando o *Libpcap* entra em ação. Ele se encontra no ficheiro *capture.cpp* e tem os seguintes argumentos:

```
void MainWindow::onPacket(u_char*, const struct pcap_pkthdr *header,
        const u_char *packet)
```

O primeiro argumento não é usado. O segundo é um cabeçalho criado pelo *Libpcap* que indica, entre outras coisas, o comprimento do pacote. O terceiro é um apontador para a sequência de bytes capturados no *layer* 2, também chamado de *frame*.

Todo *frame* tem em seu início um cabeçalho *ethernet* [16], com o formato descrito na Figura 6:

802.3 Ethernet packet and frame structure

Layer	Preamble	Start of frame delimiter	MAC destination	MAC source	802.1Q tag (optional)	Ethertype (Ethernet II) or length (IEEE 802.3)	Payload	Frame check sequence (32-bit CRC)	Interpacket gap
	7 octets	1 octet	6 octets	6 octets	(4 octets)	2 octets	46(42)[b]–1500 octets	4 octets	12 octets
Layer 2 Ethernet frame			← 64–1518(1522) octets →						
Layer 1 Ethernet packet	← 72–1526(1530) octets →								

Figura 6 - Cabeçalho de um *frame ethernet*

O *frame* propriamente dito começa depois do SFD (*start of frame delimiter*). Precisamos então de 6 bytes para cada um dos *Mac Addresses* de origem e destino. Os 4 bytes seguintes do protocolo 802.1Q (VLAN) são opcionais, normalmente ausentes. Os próximos 2 bytes referne-se ao *EtherType*, que informa o tipo de protocolo encapsulado.

Para poder interpretá-lo, precisamos de um *struct* em C++ com tipos que tenham o mesmo comprimento em bytes do cabeçalho. O *struct* que eu criei é o seguinte:

```
struct ETHERNET
{
        u_char  destination[6]; // u_char tem 1 byte de comprimento
        u_char  source[6];
        u_short type;           // u_short tem 2 bytes de comprimento
};
```

Observe que não precisamos de declarar uma estrutura que vá até o final do pacote. Precisamos de capturar apenas do início até à parte que nos interessa.

A seguir está o código responsável pela captura do *frame header*:

```
        if (header->caplen < sizeof(ETHERNET))
                return;
        ETHERNET *frame = (ETHERNET*)(packet);
```

Primeiro verificamos se o pacote contém bytes suficientes para fazer a captura. Existem erros de comunicação em rede, e não fazer esta verificação pode travar o programa [17]. Em seguida criamos a variável *frame* que faz um *cast* do pacote fornecido pelo *Libpcap*.

Com isto podemos já capturar os *Mac Addresses* do pacote:

```
QString macSrc = mac2str(frame->source);
QString macDst = mac2str(frame->destination);
```

Todo pacote tem estes *Mac Addresses* [18]. Ele é usado para transferir informações entre equipamentos de hardware. Se o pacote tiver origem em um dos *hosts* da rede local, então este é o *Mac Address* deste *host*. Caso contrário, o endereço pertence ao roteador. Posteriormente é feita esta verificação para associar este pacote ao *host* correto.

Eu criei a função *mac2str()*, que converte os bytes do *Mac Address* em uma *string*:

```
QString MainWindow::mac2str(u_char *mac)
{
        QString str;
        for (int i=0; i<6; i++)
        {
                str += QString().sprintf("%02x",mac[i]);
                if (i!=5)
                        str +="-";
        }
        return str;
}
```

O método *sprintf* do tipo *QString* simula o conhecido comando do C++ padrão.

Outra informação que já temos disponível é o tipo de protocolo presente no *Payload*, que é a parte de dados que o *frame* carrega. O valor é codificado no formato *BigEndian*, e portanto precisa ser convertido para o formato do sistema operativo em questão.

Se o valor for 0x0800, então é uma pacote IPv4 [19].

```
if (qFromBigEndian(frame->type) == 0x0800) {
        // Capture header
        if (header->caplen < sizeof(ETHERNET)+sizeof(IPv4))
                return;
        IPv4 *ip = (IPv4*)(packet + sizeof(ETHERNET));

        // Get source and destination
        source = ipv4str(ip->source);
        destination = ipv4str(ip->destination);

        // If TCP or UDP, capture ports
        if (ip->protocol == 6 || ip->protocol == 17) {
                u_int tcp_size = (ip->vhl & 0x0f)*4;
                if (tcp_size < sizeof(IPv4)) return;
                if (header->caplen < sizeof(ETHERNET)+
                        tcp_size+sizeof(TCP_or_UDP))
                        return;
                TCP_or_UDP *tcp = (TCP_or_UDP*)
                        (packet + sizeof(ETHERNET) + tcp_size);
                portSrc = QString::number(qFromBigEndian(
                        tcp->source_port));
                port = qFromBigEndian(tcp->destination_port);
                portDst = QString::number(port);
```

```
            }

            // Add protocol
            toList += "Protocol:";
            protocol = ip->protocol;
            toList += iana::getProtocol(protocol);
    }
```

No primeiro bloco, realizamos uma verificação semelhante à anterior para assegurar que há uma quantidade de bytes suficientes para fazer a captura. Um pacote IPv4 [20] tem o cabeçalho descrito na Figura 7:

IPv4 Header Format

<table>
<tr><th>Offsets</th><th>Octet</th><th colspan="8">0</th><th colspan="8">1</th><th colspan="8">2</th><th colspan="8">3</th></tr>
<tr><th>Octet</th><th>Bit</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th><th>13</th><th>14</th><th>15</th><th>16</th><th>17</th><th>18</th><th>19</th><th>20</th><th>21</th><th>22</th><th>23</th><th>24</th><th>25</th><th>26</th><th>27</th><th>28</th><th>29</th><th>30</th><th>31</th></tr>
<tr><td>0</td><td>0</td><td colspan="4">Version</td><td colspan="4">IHL</td><td colspan="6">DSCP</td><td colspan="2">ECN</td><td colspan="16">Total Length</td></tr>
<tr><td>4</td><td>32</td><td colspan="16">Identification</td><td colspan="3">Flags</td><td colspan="13">Fragment Offset</td></tr>
<tr><td>8</td><td>64</td><td colspan="8">Time To Live</td><td colspan="8">Protocol</td><td colspan="16">Header Checksum</td></tr>
<tr><td>12</td><td>96</td><td colspan="32">Source IP Address</td></tr>
<tr><td>16</td><td>128</td><td colspan="32">Destination IP Address</td></tr>
<tr><td>20</td><td>160</td><td colspan="32">Options (if IHL > 5)</td></tr>
</table>

Figura 7 - Cabeçalho de um pacote IPv4

Ele é capturado com o seguinte *struct*:

```
    struct IPv4
    {
            u_char  vhl;
            u_char  type_of_service;
            u_short lenght;
            u_short id;
            u_short offset;
            u_char  ttl;
            u_char  protocol;
            u_short checksum;
            in_addr source;
            in_addr destination;
    };
```

Queremos aqui obter os endereços de origem e destino, por isso temos que especificar todos os dados até chegar a estes. Os endereços são convertidos com o método *ipv4str()*, que é semelhante ao explicado anteriormente na conversão do *Mac Address*.

Este é um pacote IP. Há vários protocolos que podem ser codificados dentro deste tipo de pacote [21]. O campo *protocol* nos fornece esta informação. Se o valor de *protocol* for 6, temos um pacote TCP. Se for 17, temos UDP.

Estes dois tipos de protocolo têm a particularidade de informar o número de uma porta de origem e destino. Esta também é uma informação que nos interessa. Para captura-las, temos que primeiro calcular o valor do tamanho do pacote IPv4, pois ele é variável. Isto é feito através da variável *vhl*, que capturou do cabeçalho os primeiros 8 bits de informação, correspondente à versão e ao IHL. Para isolar apenas o IHL, fazemos uma operação AND com o valor 0x0f. O valor IHL informa o número de blocos de 32 bits presentes no pacote. Por isso, para obter o valor em bytes, temos que multiplicar por 4.

Com isso aplicamos mais uma vez o mesmo raciocínio. Verificamos se há uma quantidade de bytes suficientes para fazer a captura e obtemos o cabeçalho do pacote UDP [22] ou TCP [23] correspondente. Nas Figuras 8 e 9 temos a descrição dos mesmos.

UDP Header

Offsets Octet		0	1	2	3
Octet	Bit	0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23	24 25 26 27 28 29 30 31
0	0	Source port		Destination port	
4	32	Length		Checksum	

Figura 8 - Cabeçalho de um pacote UDP

TCP Header

Offsets Octet		0	1	2	3
Octet	Bit	0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23	24 25 26 27 28 29 30 31
0	0	Source port		Destination port	
4	32	Sequence number			
8	64	Acknowledgment number (if ACK set)			
12	96	Data offset / Reserved 0 0 0 / NS	CWR ECE URG ACK PSH RST SYN FIN	Window Size	
16	128	Checksum		Urgent pointer (if URG set)	
20	160	Options (if *data offset* > 5. Padded at the end with "0" bytes if necessary.)			
...	...	...			

Figura 9 - Cabeçalho de um pacote TCP

Observe que em ambos os tipos de pacote, os endereços das portas de origem e destino encontram-se logo no início. Com isso podemos usar o mesmo *struct* para capturar esta informação de ambos os tipos de protocolo:

```
struct TCP_or_UDP
{
        u_short source_port;
        u_short destination_port;
};
```

As outras informações não nos interessam, sendo portanto ignoradas. Com isso as variáveis *portSrc* e *portDst* recebem os respectivos valores.

O mesmo processo se aplica caso o pacote seja IPv6 [24], onde o valor de *frame->type* é igual a 0x86dd. O código é muito semelhante, não sendo portanto necessário repeti-lo aqui. Uma vantagem do cabeçalho IPv6 é que o seu comprimento é fixo, como vemos na Figura 10.

<table>
<caption>Fixed header format</caption>
<tr><th>Offsets</th><th>Octet</th><th colspan="8">0</th><th colspan="8">1</th><th colspan="8">2</th><th colspan="8">3</th></tr>
<tr><th>Octet</th><th>Bit</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th><th>13</th><th>14</th><th>15</th><th>16</th><th>17</th><th>18</th><th>19</th><th>20</th><th>21</th><th>22</th><th>23</th><th>24</th><th>25</th><th>26</th><th>27</th><th>28</th><th>29</th><th>30</th><th>31</th></tr>
<tr><th>0</th><th>0</th><td colspan="4">Version</td><td colspan="8">Traffic Class</td><td colspan="20">Flow Label</td></tr>
<tr><th>4</th><th>32</th><td colspan="16">Payload Length</td><td colspan="8">Next Header</td><td colspan="8">Hop Limit</td></tr>
<tr><th>8</th><th>64</th><td colspan="32" rowspan="4">Source Address</td></tr>
<tr><th>12</th><th>96</th></tr>
<tr><th>16</th><th>128</th></tr>
<tr><th>20</th><th>160</th></tr>
<tr><th>24</th><th>192</th><td colspan="32" rowspan="4">Destination Address</td></tr>
<tr><th>28</th><th>224</th></tr>
<tr><th>32</th><th>256</th></tr>
<tr><th>36</th><th>288</th></tr>
</table>

Figura 10 - Cabeçalho de um pacote IPv6

Ele é interpretado com o seguinte *struct*:

```
struct IPv6
{
        u_int    vtcfl;
        u_short  lenght;
        u_char   next_header;
        u_char   ttl;
        in6_addr source;
        in6_addr destination;
};
```

Se o pacote lido não for IPv4 ou IPv6, a única informação que temos é o *Mac Address*. Neste caso, o conteúdo das variáveis *source* e *destination* irão receber esta informação. Isto ocorre, por exemplo, com os pacotes do protocolo ARP, que são usados para fazer a conversão de endereços de hardware para IP.

Os dados são então repassados ao método *addHost()*, que irá criar as instâncias da classe *host*. O método recebe os seguintes argumentos:

```
addHost(source,destination,macSrc,macDst,port,protocol);
```

Observe que a variável *port* contém o valor numérico da porta de destino. Os valores das portas de origem e destino são exibidos no *log*, mas na criação do *host* precisamos apenas desta última. Isso se deve ao fato de que a porta de origem é apenas um endereço de resposta ao pacote solicitado, não tendo grande importância. Já a porta de destino informa o serviço que está sendo acedido, como 80, 443 ou 21 (que indicam respectivamente *http, https* e *ftp*).

3.3 Instâncias dos *hosts*

Nesta etapa, os pacotes já foram lidos e as informações desejadas, como endereço IPv4, IPv6, *Mac Address* e portas, são repassados como argumento. Este método também se encontra no ficheiro *capture.cpp*:

```
void MainWindow::addHost(QString source, QString destination, QString
        macSrc, QString macDst, int port=-1, int protocol=-1)
```

A porta e protocolo recebem os valores -1 por defeito. Estes dados só estão presentes se for um pacote TCP ou UDP. Nos outros casos, eles são ignorados. A primeira verificação que devemos fazer é se o endereço de origem é composto por zeros. Isto ocorre se o *host* estiver à procura de um servidor DHCP:

```
        if (source == "0.0.0.0" ||
            source == "0000:0000:0000:0000:0000:0000:0000:0000")
            source = macSrc;
```

Neste caso, o endereço de origem passa a ser o mesmo do *Mac Address*. É importante lembrar que todo pacote tem um *Mac Address*. Mas essa informação será usada apenas se o *host* estiver na rede local.

No *header* da classe *MainWindow* foi criado o vector *hosts*. Este é o local onde serão armazenados todos os *hosts* encontrados. A próxima verificação que o método *addHost()* tem que fazer é se já existe um *host* com os dados existentes:

```
        int lastSrc = 0;
        int lastDst = 0;
        bool srcPresent = false;
        bool dstPresent = false;
        for (int i=0; i<hosts.size(); i++) {
                if  (hosts[i]->exists(source) ||
                    (hosts[i]->exists(macSrc) && hosts[0]-
>subnet(source))){
                        srcPresent = true;
                        if (source.contains("."))
                                hosts[i]->addIPv4(source);
                        else if (source.contains(":"))
                                hosts[i]->addIPv6(source);
                        if (hosts[0]->subnet(source))
                                hosts[i]->addMac(macSrc);
                        if (hosts[i]->getName() == "")
                                QHostInfo::lookupHost(source,
                                this,SLOT(lookedUp(QHostInfo)));
                        hosts[i]->setTime(QTime::currentTime());
                        hosts[i]->incSnd();
                        lastSrc = i;
                } ...
```

A variável *source* pode conter três tipos de informação: IPv4, IPv6 ou *Mac Address*. No comando *if*, verificamos se esta informação já existe. Além disso, pode haver o caso em que a variável *source* informa um IPv4, mas o *host* só possui o *Mac Address* registado. A segunda parte da operação booleana permite detectar casos como este.

Se for encontrado um *host* no vector, as suas informações são atualizadas. Se o *host* ainda não tiver um nome, chamamos o método *HostInfo::lookupHost()*, que possui uma característica interessante: ele cria uma tarefa (*thread*) em paralelo que tenta fazer a conversão do endereço de origem para um nome. Este é o único momento em que o *NetRevealer* envia um pacote à rede. O processo é feito com uma consulta reversa de DNS.

O fato de ser criada uma tarefa extra para isso é muito importante, pois este processo de consulta pode demorar [25], e não podemos deixar o programa parado aguardando-a. Quando o sinal de retorno chega, será recebido pelo seguinte método:

```
void MainWindow::lookedUp(const QHostInfo &host) {
        if (host.error() != QHostInfo::NoError)
                return;
        for (int i=0; i<hosts.size(); i++) {
                QString adr;
                if (hosts[i]->getIPv4(0) != "")
                        adr = hosts[i]->getIPv4(0);
                else if (hosts[i]->getIPv6(0) != "")
                        adr = hosts[i]->getIPv6(0);
                if (adr != "")
                      foreach (const QHostAddress
&address,host.addresses())
                                if (adr.left(5) !=
host.hostName().left(5)&&
                                      adr == address.toString()) {
                                      hosts[i]-
>setName(host.hostName());
                                      hosts[i]->detectIcon();
                                }
        }
}
```

O método não sabe a qual *host* a consulta se refere. Portanto, ele precisa de realizar uma busca para encontrá-lo. A variável do tipo *HostInfo* dada como argumento pode também retornar com informações sobre vários *hosts*. O método precisa portanto realizar mais um *loop* interno para fazer a comparação.

Voltando a falar do método *addHost()*, o código exibido refere-se apenas ao endereço de origem. O mesmo processo é feito com o endereço de destino, com o acréscimo da porta:

```
        if (port != -1)
                hosts[i]->addPort(port,protocol);
```

Se o *host* não estiver presente, cria-se um novo. Isto é feito da seguinte forma:

```
        if (!srcPresent)
        {
               // Create a new host
               host *newHost = new host();
               newHost->setIcon(1);
               if (source.contains("."))
                        newHost->addIPv4(source);
               else if (source.contains(":"))
                        newHost->addIPv6(source);
               if (hosts[0]->subnet(source))
               {
                        newHost->addMac(macSrc);
                        newHost->setIcon(2);
               }
               newHost->setTime(QTime::currentTime());
               newHost->incSnd();

        QHostInfo::lookupHost(source,this,SLOT(lookedUp(QHostInfo)));
               hosts[0]->angle(newHost);

               // Add host to view
               scene->addItem(newHost);
```

```
                hosts.append(newHost);
                lastSrc = hosts.size()-1;
                detectOverlaping();
        }
```

O *host* é adicionado à cena (explicada posteriormente) e ao vector.

O mesmo se faz para o endereço de destino. Mas antes disso, verificamos se o destino não é um *broadcast*:

```
        if (hosts[0]->broadcast(destination))
                hosts[lastSrc]->setBroadcastTime(QTime::currentTime());
```

Observe que faz-se a atualização do horário quanto um pacote é associado a um *host*. Isto é importante para poder usar a opção do menu que remove *hosts* inativos a "x" segundos. A variável *delLimit* informa o número de segundos que se deve esperar. Se ela estiver com o valor zero, o recurso estará desligado.

3.4 Ícones dos hosts

Este método faz uso intenso dos recursos gráficos do *Qt*. Para isso, adicionou-se um objeto do tipo *graphicsView* à parte superior da janela principal. Este tipo de objeto cria um ambiente gráfico, onde cada item adicionado possui coordenadas *x* e *y* associadas, como em um eixo cartesiano bidimensional.

Para se usar os recursos de forma plena, deve-se acrescentar uma cena a este objeto [26]. Aqui temos as definições das propriedades iniciais:

```
scene = new QGraphicsScene(this);
scene->setSceneRect(-5000,-5000,10000,10000); // Set canvas area
ui->graphicsView->setScene(scene);
ui->graphicsView->viewport()->setMouseTracking(true);
ui->graphicsView->viewport()->installEventFilter(this);
ui->graphicsView->setRenderHint(QPainter::Antialiasing);
ui->graphicsView->setDragMode(QGraphicsView::ScrollHandDrag);
ui->graphicsView->setCacheMode(QGraphicsView::CacheBackground);
ui->graphicsView->setVerticalScrollBarPolicy(Qt::ScrollBarAlwaysOff);
ui->graphicsView-
>setHorizontalScrollBarPolicy(Qt::ScrollBarAlwaysOff);
ui->graphicsView-
>setViewportUpdateMode(...::BoundingRectViewportUpdate);
```

As primeiras três linhas criam a cena, que possui uma área de trabalho grande para evitar que o *Qt* modifique o tamanho da janela para se ajustar à área criada. A janela é exibida inicialmente com a posição 0,0 no centro. As duas linhas seguintes ativam a emissão de sinais quando se usa o rato (explicada posteriormente). É ativada então a propriedade *antialiasing* para melhorar a qualidade da imagem. A opção *setDragMode* permite mover a posição vista na janela clicando-se com o rato em uma área vazia e arrastando-o. As linhas seguintes desligam as barras de rolagem laterais (*scrollbars*) e definem algumas propriedades que melhoram a performance do programa.

Cada objeto adicionado à cena deve herdar as propriedade da classe *QGraphicsItem*. Isto foi feito com os objectos da classe *Host* e *Packet*. Os objetos adicionados à cena (chamados aqui de itens) devem ter os seguintes métodos definidos:

- **boundingRect()** - Define os limites do item a ser exibido. Deve-se retornar um objeto do tipo *QRect*, que representa um retângulo no *Qt*. Os valores informados referem-se ao centro do objeto e não à cena. Aqui temos o código usado na classe *Host*:

```
        QRectF host::boundingRect() const {
                return QRectF(-20,-20,40,40);
        }
```

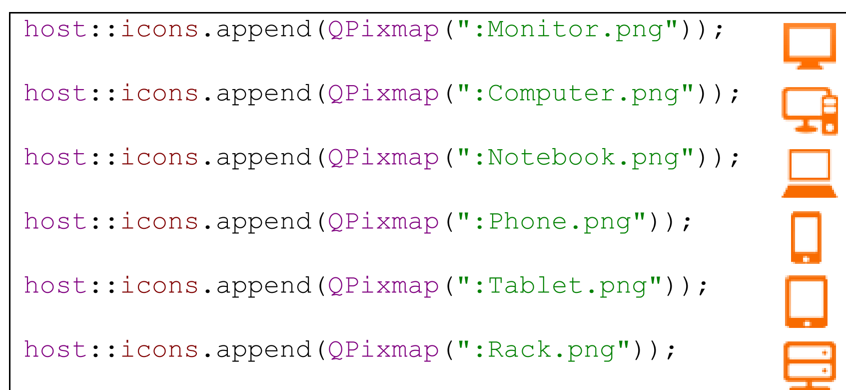
- **paint()** - Desenha o objeto propriamente dito. Temos aqui o exemplo para a classe *Host*:

```
void host::paint(QPainter *painter, const QStyleOptionGraphicsItem *,
QWidget*)
{
        // Draw broadcast circle
        if (broadcastTime != QTime(0,0,0))
        {
                if (broadcastTime.msecsTo(QTime::currentTime()) < 200)
                        painter->setPen(QColor(255,133,0)); // Orange
                else
                        painter->setPen(Qt::black);
                painter->setBrush(QBrush(Qt::white));
                painter->drawEllipse(boundingRect());
        }

        // Draw icon
        QRectF target(-16,-16,32,32);
        QRectF source(0,0,32,32);
        painter->drawPixmap(target,icons[icon],source);
}
```

Neste caso desenha-se um círculo ao redor do ícone, caso este tenha emitido um pacote *broadcast* ou *multicast*. A cor do círculo depende de há quanto tempo o pacote foi enviado. Se tiver sido a menos de 200 milissegundos, desenham-se com a cor laranja.

Desenha-se então o ícone associado ao *host*. Obtem-se através da variável *icon* a figura correta. O vector *icons* foi definido no construtor da classe *MainWindow*:



```
host::icons.append(QPixmap(":Monitor.png"));

host::icons.append(QPixmap(":Computer.png"));

host::icons.append(QPixmap(":Notebook.png"));

host::icons.append(QPixmap(":Phone.png"));

host::icons.append(QPixmap(":Tablet.png"));

host::icons.append(QPixmap(":Rack.png"));
```

```
host::icons.append(QPixmap(":Router.png"));

host::icons.append(QPixmap(":wRouter.png"));
```

A escolha do ícone é feita pelo método *detectIcon()* presente na classe *Host*. Ele faz a escolha de acordo com o nome associado ao mesmo, recebido pela função *lookedUp()*.

- **itemChange()** - Pode-se definir também este método da classe *QGraphicsItem*, que recebe um sinal quando há uma mudança de posição do item, redesenhando-o.

3.5 Posição no mapa

Eu defini também um chamado *angle()*. Ele é chamado durante a criação de um *host*:

```
void host::angle(host *h) {
        double ang, x, y, dist = this->x()*this->x() + this->y()*this-
>y();
        do {
                ang = qrand()%628;
                x = this->x()+sin(ang)*100;
                y = this->y()+cos(ang)*100;
        } while (x*x + y*y < dist+84);
        h->setPos(x,y);
}
```

Com ele define-se a posição que o ícone terá inicialmente. O método é chamado a partir do *host* "pai" no gráfico. Se o primeiro pacote que define este *host* for declarado no cabeçalho capturado pelo *Libpcap* como origem, a posição será em relação às coordenadas do *host* inicial (que representa o computador onde o programa está a rodar). Se for destino, será em relação ao *host* de origem do pacote.

O método calcula inicialmente a distância do *host* pai até a origem 0,0, onde normalmente encontra-se o ícone inicial. Obtém-se então uma posição aleatória para o filho em relação ao pai, que vai estar em um raio de 100 pixels. Se esta posição for menor que a distância calculada inicialmente mais 84, procura-se outra posição. O objetivo é evitar que o filho fique dentro do raio de atuação do ícone inicial (seu avô). A posição do filho deve ser então a uma distância de no mínimo 100+84 pixels, como pode ser vista na Figura 11. Estes valores fazem com que os três ícones formem um ângulo de no mínimo 120º.

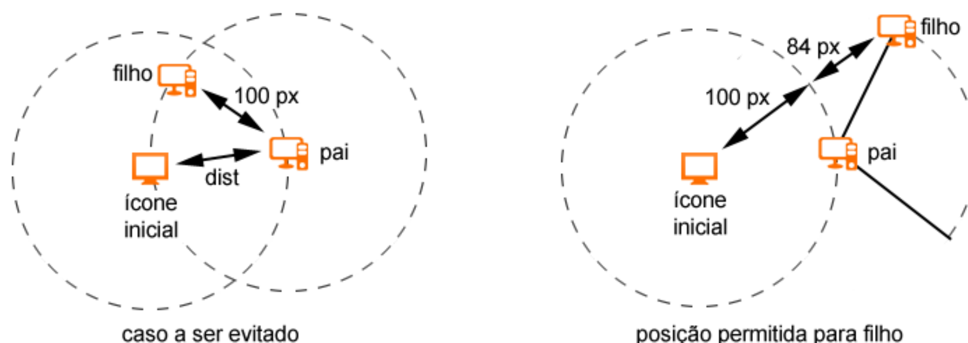


Figura 11 - Definição da posição do ícone

Se o ícone for calculado em relação ao inicial, então ele pode ficar em qualquer posição em um raio de 100 pixels, como ocorre com o ícone pai na Figura 11.

3.6 Eventos

Para permitir interpretar os sinais do rato sobre o *graphicsView*, foi declarada (como já dito) a opção *installEventFilter* à cena. Com isto, todos os sinais emitidos pelo *graphicsView* são repassados à *MainWindow*. Os sinais são recebidos pelo método *eventFilter()* declarado nesta classe e mostrado aqui em pseudo-código (pois o método é um pouco longo):

```
bool MainWindow::eventFilter(QObject *, QEvent *event)
{
      Define valores para caixa de informação (toolTip)
      Se sinal = MouseMove
            Se rato está sobre um host
                  Exibe texto obtido por info()
                  Exibe retângulo abaixo do texto
      Se sinal = MouseButton && RightButton
            Se rato está sobre um host && não for host inicial
                  Mostra menu para deletar
            Se estiver sobre área vazia
                  Mostra menu de zoom
}
```

Inicialmente adiciona-se à cena um item de texto e um retângulo, ambos com a visualização desligada (*setVisible*(*false*)). A declaração como estático permite aos mesmos persistirem entra as chamadas ao método. Estes itens formam a caixa de informação (*tooltip*).

Quando o rato é movido, verifica-se se o mesmo está sobre um *host*. Se for o caso, obtêm-se as informações sobre o mesmo com o método *info()* da classe *Host*. As informações são então colocadas no item de texto e o retângulo é redimensionado para o tamanho deste. Ambos os item são então ligados, de forma a ficarem visíveis na *graphicsView*. Se o cursor sair da área do *host*, desliga-se a visualização novamente. Portanto os mesmos itens são usados para exibir as informações de todos os *hosts*. O efeito pode ser visto na Figura 12.

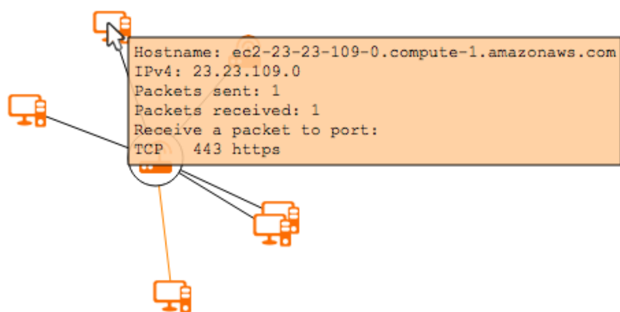


Figura 12 - Caixa de informação

Quando o botão direito do rato é pressionado, verifica-se se o mesmo encontra-se sobre um *host* ou uma área vazia. No primeiro caso exibe-se a opção que permite deletar o mesmo (uma janela pede pela confirmação do ato). No segundo caso, exibe-se as

opções para realizar o *zoom*. Esta opção usa o método *scale()* do *graphicsView* e multiplica ou divide o valor atual por 1,5 de acordo com o *zoom* escolhido. O efeito pode ser visto na Figura 13.

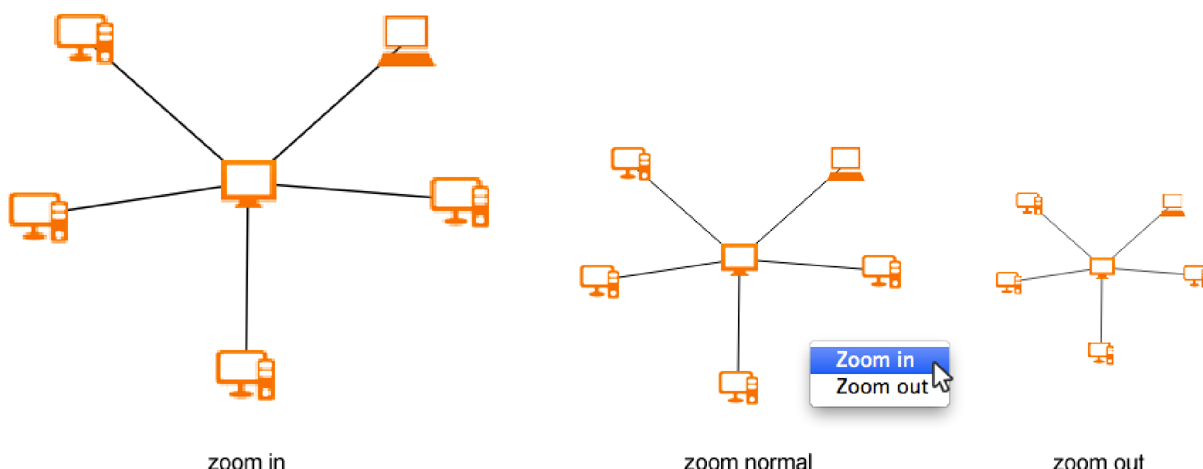


Figura 13 - Zoom

Quando o botão esquerdo do rato é pressionado, o método *eventFilter()* não modifica o comportamento por defeito que o rato tem sobre o *graphicsView*. Neste caso, quando se está sobre uma área vazia, é possível mover a área de visualização atual (*pan*). Quando se está sobre um *host*, é possível movê-lo. Isso é habilitado através do comando "*setFlag(ItemIsMovable)*", acrescentado no construtor da classe *Host*.

3.7 Ligações entre hosts

Para concluir o *NetRevealer*, foi preciso criar mais uma classe chamada *Packet*. Ela representa um pacote que é enviado entre dois *hosts*. Cada instância desta classe possui um ponteiro para um *host* chamado *source* e outro chamado *destination*.

As instâncias desta classe são criadas no final do método *addHost()*, como pode ser visto no código a seguir:

```
// Search if packet from source to destination was already sent
bool find = false;
for (int i=0; i<hosts[lastSrc]->getPackets().size(); i++)
{
	if (hosts[lastSrc]->getPackets()[i]->getSrc() == hosts[lastDst] ||
		hosts[lastSrc]->getPackets()[i]->getDst() == hosts[lastDst])
	{
		hosts[lastSrc]->getPackets()[i]-
>setTime(QTime::currentTime());
		find = true;
		break;
	}
}

// Search if packet from destination to source was already sent
if (!find)
   ...

// If it's new
```

```
if (!find)
{
     packet *p = new packet(hosts[lastSrc],hosts[lastDst]);
     scene->addItem(p);
}
```

O código verifica inicialmente se já foi enviado um pacote entre os *hosts* em questão. Cada *host* possui um vector com uma lista de pacotes à qual está ligado. Se a ligação entre os *hosts* não existe, cria-se uma nova e adiciona-se à cena.

O construtor da classe *Packet* adiciona a instância aos respectivos vectores dos *hosts* de origem e destino.

```
packet::packet(host *src, host *dst)
{
        source = src;
        destination = dst;
        source->addPacket(this);
        destination->addPacket(this);
        setTime(QTime::currentTime());
}
```

Em ambos os casos, salva-se o momento da criação ou atualização com o método *setTime()*, através do qual é possível definir se a linha será desenhada na cor laranja ou preta. As linhas de cor laranja representam os pacotes enviados há pouco tempo, aumentando o efeito de atividade em tempo real que o *NetRevealer* oferece.

Os métodos *paint()* e *boundingRect()* da classe *Packet* atualizam seus valores de acordo com a posição dos *hosts* de origem e destino, de forma que quando os mesmos são movidos, as linhas são redesenhadas da maneira correta.

4. Conclusões

O *NetRevealer* permite ao técnico de redes de computadores presente em campo realizar um diagnóstico preciso das atividades correntes nos equipamentos em tempo real. O aplicativo causa pouco impacto no computador onde será instalado, pois ocupa pouco espaço tanto na memória quanto no disco e causa pouca sobrecarga sobre o processamento local. A interface é de fácil uso, sendo intuitiva e simples. As informações fornecidas são tecnicamente relevantes para o uso por profissionais médios / avançados.

O programa permitiu experimentar a criação de um aplicativo em interface gráfica, adicionando a capacidade de poder ser compilado em diversos sistemas operativos. Ele segue a tendência presente no mercado de integração de várias tecnologias, contribuindo para eliminar as diferenças presentes entre diversos ambientes computacionais.

Bibliografia

1. C. Chen, “Top 10 Unsolved Information Visualization Problems”, IEEE Computer Graphics and Applications, vol. 25, no. 4, pp. 12-16, July/Aug. 2005.
2. S. Ansari, Rajeev S.G. and Chandrasekhar H.S, "Packet Sniffing: A brief Introduction", IEEE Potentials, Dec 2002 Jan2003, vol. 21, issue 5, pp: 17-19.
3. Zhiqiang Zhu, Jing Yan, Zhuopeng Wang, Guangyin Xu, “Data flow monitoring and control of LAN based on strategy”, IEEE Conference Publications, vol. 2, pp: 225-228, May/2010.
4. J. Zhang, A. Moore, "Traffic trace artifacts due to monitoring via port mirroring," in Workshop on End-to-End Monitoring Techniques and Services, May 2007, pp. 1-8.
5. D. Ficara, S. Giordano, F. Oppedisano, G. Procissi, F. Vitucci, "A cooperative PC/Network-Processor architecture for multi gigabit traffic analysis," 4th International Telecommunication Networking Workshop on QoS in IP Networks, 2008.
6. A. S. Tanenbaum, “Computer Networks”, Ed. 4, pp 45-47, 1981, Prentice Hall.
7. Duarte Vitor, Farruca Numo, "Using libpcap for monitoring distributed applications", 2010 International Conference on High Performance Computing and Simulation, HPCS 2010, Caen France, June 28, 2010 -July 2, 2010.
8. Mohammed Abdul Qadeer, Mohammad Zahid, Arshad Iqbal, "Network Traffic Analysis and Intrusion Detection using Packet Sniffer", Proceeding of the Second International Conference on Communication Software and Networks 2010.
9. M.F. Kaashoek, D.R. Engler, G.R. Ganger, D.A. Wallach, “Server operating systems”, Proceedings of the 7th workshop on ACM SIGOPS European workshop: Systems support for worldwide applications, September 09-11, 1996, Connemara, Ireland.
10. D. Xu, Z. Tan, Y. Gao, “Developing application and realizing multiplatform based on Qt framework”, Journal of Northeast Agricultural University, March/2006.
11. Brant D. Thomsen , “Method for detecting unauthorized network access by having a NIC monitor for packets purporting to be from itself”, 3Com Corporation, Jan/2002, Patent: US 6745333 B1.
12. L.M.Garcia, “Programming with Libpcap – Sniffing the Network from Our Own Application”, Hacking Magazine, Feb/2008.
13. T. S. Perry, J. Voelcker, "Of Mice and Menus: Designing the User-Friendly Interface", IEEE Spectrum, vol. 26, no. 9, pp.46 -51, 1990.
14. Yu Ren, Jun Qiu, “Embedded GUI Design Using Signal-Slot Communication Mechanism”, 2009. WCSE '09. WRI World Congress on Software Engineering, vol. 1, pp: 159-162, May/2009.
15. W.F. Jolitz, M.T. Lawson, L.G. Jolitz, “TCP/IP network accelerator system and method which identifies classes of packet traffic for predictable protocols”, Interprophet Corporation, Jan/2001, Patent: US 6173333 B1.
16. David C. Plummer, “An Ethernet Address Resolution Protocol”, Network Working Group, Nov/1982, RFC 826.
17. R. Albert, H. Jeong, A.L. Barabási, “Error and attack tolerance of complex networks”, Nature Magazine, no. 406, pp. 378-382, Jul/2000.
18. S. Whalen, S. Engle, D. Romeo, “An Introduction to ARP Spoofing”, April/2001.

19. K. Liu, C. Hu, C. Yuan, C.L. Pok, “Ethernet architecture with data packet encapsulation”, At&T Intellectual Property L, L.P., Jan/2010, Patent: US 7643424 B2.
20. J. Postel, “Internet Protocol”, Department of Defense Standard, Jan/1980, RFC: 760.
21. J. Postel, “Assigned Numbers”, Network Working Group, Jan/1980, RFC 762.
22. J. Postel, “User Datagram Protocol”, Internet Standard, Aug/1980, RFC 768.
23. J. Postel, “Transmission Control Protocol”, Department of Defense Standard, Jan/1980, RFC: 761.
24. S. Deering, R. Hinden, “Internet Protocol, Version 6 (IPv6) Specification”, Network Working Group, Dec/1998, RFC 2460.
25. A. Sears, J.A. Jacko, M.S. Borella, “Internet delay effects: how users perceive quality, organization, and ease of use of information”, CHI '97 extended abstracts on Human factors in computing systems: looking to the future, March 22-27, 1997.
26. M Dalheimer, “Programming with QT: Writing portable GUI applications on Unix and Win32”, O’Reilly Velag GmbH & Co KG, ed: 2, 2002.

Saulo Fonseca, Formado em Eletrónica pela Escola Técnica Federal de Pernambuco do Brasil e aluno de Licenciatura em Informática da Universidade Aberta de Portugal. Iniciou sua vida profissional como técnico de redes de computadores e dedica-se a concluir as poucas unidades curriculares que lhe faltam para obter o grau de licenciado.

Henrique São Mamede, Doutorado em Sistemas e Tecnologias de Informação pela Universidade do Minho, com interesse nas áreas de investigação em engenharia e gestão de sistemas de informação. Professor da Universidade Aberta, no Departamento de Ciências e Tecnologia, onde exerce há 14 anos. Colaborador do INESC-TEC.

(esta página par está propositadamente em branco)