HEBREW UNIVERSITY

NetUtils Java Library Toturial

Author:
Roni BAR-YANAI
roni.bar.yanai@gmail.com

Supervisor:
Prof. Danny DOLEV

May 10, 2011

Contents

Chapter 1

Introduction

1.1 Preface

'netuils' is a Java library that combines several useful functionalities for various network traffic research tasks. It is written mostly in Java and therefore easy and fast to develop compared to C/C++ (although C/C++ fans might not think so). The library also has the advantage of multi-platform support. The 'netutils' package gives a low level java network library. It contains extensive infrastructure for sniffing, injecting, building and parsing Ethernet/IP/TCP/UDP/ICMP packets. The library is based (using JNI - Java Native Interface) on two multi platform well known c libraries: libnet and libpcap. libnet is used for injection while libpcap is used for recording/snifing. The package also includes pure java packages for parsing and reading/writing to/from capture files using the libpcap format (and some other formats). There are several similar projects, such as JPcap and JEthereal, which are available at the Internet .However, JPcap is just a wrapper that provides direct access to the C functionality and has no high level object oriented structure (last maintenance was done on 2007). JEthereal only provides recording and sniffing functionality. Other free projects do not have a reasonable and easy API. Another problem is the lack of basic support for IPv6. netutils was written during my thesis, mainly for traffic analyze and partly for running simple tests which included traffic injection. 'netutils' injection and sniffing performance capability is limited and could not be used with high line rate, but it is good enough for small to medium tests (when recording the real bootle neck is usually the disk).

1.2 Overview

In this section we describe the library content. The netuitls library can be split into the following components:

Traffic parsing Collection of classes for parsing TCP/UDP/IP packets data. The package provides an API for easy and abstract access to all network layers header fields and to the layers payload. The library contains extensive printing functionality for headers and payloads.

Flow abstraction Collection of classes for analyzing network traffic at flow level (see 2.2.3 for flow definition). When analyzing network traffic we may want to get statistics about flows

and not statistics about independent packets. For example, we may would like to know the average number of packets for flow, longest flow, flow duration...etc. The package expose interfaces for getting flows and analyze them.

Traffic generation API for easy creation of TCP segments, UDP segments and other IP based packets. The API can be used for creating flows, for example, converting capture file consisted of TCP and UDP flows carried over IPv4 into capture file consisted of the same flows (TCP and UDP) but carried over IPv6.

Capture file utilities Set of capture file utilities for common functionally, such as concatenating capture file, extracting flows from capture file and so on.

Traffic recoding and injection Utilities for injecting network traffic and recording network traffic.

Examples A collection of useful examples, which some are explained in this tutorial in detailes.

All components are integrated into a single library which is useful for many network research tasks. In the following sections, we first go over the packet parsing and packet building API, which are the basic building blocks for the other library components. We then go through the available capture file utilities. We end by describing the network recoding API and network injecting API. All packages are explained by following basic code examples.

Code Conventions:
In many of the examples we omitted some or all of the validation code and it is assumed that the input is always correct. This is done for breviary and a functional code should include those validations.
Occasionally we use the specific network naming for a network message. For example, UDP message (and other unreliable services) is refereed by *UDP datagram*, TCP is a refereed by *TCP segment*...etc, while occasionally we use the term *packet* for any network message.
All the code were compiled and run using java 6.0.

Chapter 2

Parsing packets

Parsing packet is the basic functionality needed when analyzing capture files or when building network real-time utility, e.g. **ping**. We first show how to read a capture file (online functionally is explained later when we describe the recording API 5.2.2) which is the basic functionality for each utility used. This is the only place where we will explain the capture file iteration code. In the rest of the code examples in this tutorial we will skip the capture file iteration code as it remains exactly the same.

2.1 Reading capture file

In order to read capture file, we use *edu.huji.cs.netutils.files.CaptureFileReader* which is an interface for reading capture files. We open the reader through a factory function as there are many types of possible capture formats. The library supports three types ,libpcap, nas sniffer and xcp, but other formats can be easily added in the future. The class edu.huji.cs.netutils.files.CaptureIterator implements java iterator interface and make the use of edu.huji.cs.netutils.files.CaptureFileReader interface more simple. The following example shows how to open a file and count the number of packets.

Listing 2.1: "Reading capture files"

```
public static void main(String[] args) throws IOException, NetUtilsException
{
    if(args.length<1)
    {
        System.out.println("missing file name parameter");
        System.exit(-1);
    }

    CaptureIterator it = new CaptureIterator(CaptureFileFactory.
       createCaptureFileReader(args[0]));
    int n = 0;

    while( it.hasNext())
    {
        it.next();
        n++;
```

```
    }
    System.out.println("Total number of packets : "+n);
}
```

hasNext() method return true if we still have packets to read. The next() method returns the next read packet as a byte array. The iterator provides additional method which is not part of the java iterate interface for returning the last read packet timestamp (not shown and the example). Timestamps are useful when we want to analyze time parameters of flows, for exmaple, flow total time.

2.2 Parsing Packets

The library contains network parse package and network build package, where the parse package *edu.huji.cs.netutils.parse* is used to parse the packets and get access for specific fields, for example, the TCP packet sequence number, TCP flags (SYN, RST...etc), and so on. The build package which is discussed on the next section is used to build TCP/UDP and other packets from scratch.

When reading a capture file, or when sniffing packets on an interface in its simplest form, our input is a simple byte array. The byte array is the full layer 4 packet including the underline layer 3 and the underline layer 2 (In fact, we are getting the frame). The frame can carry a TCP packet or other IP based protocol. On the other hand, it can carry an ARP ,which is not an IP based protocol. Obviously, before we can create the matching java object for the packet, we must check layer by layer for the their type, until we get the packet type. When we have the type, we can create its java object. In other words, when we get a packet as a byte array, we may first check if this an IP. If the frame is indeed an IP, we check what is the carried protocol, for example TCP,UDP...etc. When we get the protocol, we can create the correct java object, for example TCPPacket object. The library supply some factory functions for common cases.

2.2.1 IP Packets

In the following example we count the number if ipv4 packets and the number of fragment packets.

Listing 2.2: "Counting IPv4 fragments"

```
while( it.hasNext())
{
    byte data[] = it.next();
    if (EthernetFrame.statIsIpv4Packet(data))
    {
        n++;
        IPv4Packet ippacket = new IPv4Packet(data);
        if (ippacket.isFragmented()) {
            frag++;
        }
    }
}
```

We first use static function to check that the packet is ipv4 and only then we create IPvPacket object. Through the IPv4Packet object we can access the IP header fields or data. The class also

provides methods for printing the packet to buffer or stdout. Note that if we do not verify that the byte array is indeed IPv4 then we may get unexpected results when creating IPv4Packet as there is not internal check.

Example of printed IPv4 header to stdout using IPPacket.toReadableText(sb).

```
Internet Protocol
Ver: 4
Hdr Length: 20
TOS: 0
Total Length: 52
Identification: 57989
Flags: 2
Fragmnet Offset: 0
TTL: 61
Protocol: 6
Checksum: 53636
Source: 10.56.193.228
Destination: 64.103.125.22
```

Example of printed IPv6 header to stdout using IPPacket.toReadableText(sb).

```
Internet Protocol
Ver: 6
Traffic Class : 0
Flow Label: 0
Payload Length: 60
Next Header: 6
Hop Limit: 64
Source: e4c1::3800::e4c1::3800::e4c1::3800::e4c1::3800
Destination: 167d::6740::167d::6740::167d::6740::167d::6740
```

2.2.2 TCP/UDP packets

For TCP and UDP, we must add another check, this time for layer three, namely checking the protocol type that is carried by the IP layer (of course the IP packet can be IPv6 or it can be IPv4. we will address IPv6 later). In the following example, we count the number of TCP packets which has SYN flag and the number of TCP packets which has RST (reset) flag.

Listing 2.3: "Counting Syn and Rst"

```
int syn = 0;
int rst = 0;
while( it.hasNext())
{
    byte data[] = it.next();
    if (EthernetFrame.statIsIpv4Packet(data))
    {
```

```
        if (IPFactory.isTCPPacket(data))
        {
            TCPPacket tcppkt = new TCPPacket(data);
            if(tcppkt.isSyn())
            {
                syn++;
            }
            if(tcppkt.isRst())
            {
                rst++;
            }
        }
    }
}

System.out.println("Total Syn:"+syn);
System.out.println("Total Rst:"+rst);
```

As shown in the example, we first check if the packet is an IP packet. If it is an IP packet, then we check, using IPFactory, if the carried protocol is TCP. Only then, we can create TCPPacket object and access its fields. TCPPacket provides access method for all TCP header fields. It also provides access to the underline protocol which can be IPv4 or IPv6, although in the example we specifically create TCPPacket which uses IPv4. If the underline packet is of IPv6 then there is a chance that our program will crash since we just assume that layer three protocol is IPv4.

In order to get the correct TCPPacket (carried over IPv4 or IPv6) we should use the factory function IPFactory.createTCPPacket(data)
Another option is to explicitly check the IP type of the packet in hand and call TCPPacketv6 or TCPPcaketIPv4 respectively.

The following figure shows an example of printed TCP header.

```
TCP
Source Port : 58334
Destination Port : 3868
Sequence : 460251266
Acknowledgemet : 4228722687
Header Length: 32
SYN : false
ACK : true
RST : false
FIN : false
Window Size: 2532
Checksum: 23131
```

2.2.3 Five Tuple and Flows

The term *flow* refers to a single data flow connection between two hosts, defined uniquely by its five-tuple (source IP address, source port, destination IP address, destination port, protocol type TCP/UDP). To be more precise, this five-tuple uniquely defines a *flow* only at a given time, i.e.,

two concurrent flows will never use the same five-tuple. TCP flows have a well-defined start and end by the protocol definition. For UDP, we usually define the *flow start* as the first packet we see. The *flow end* is characterized by *aging*, defined as a (constant) time period that elapses with no data exchange on the flow. W

As said, Five Tuple is the key for identifying flow. *edu.huji.cs.netutils.parse.FiveTuple* class is used in the library to identify flows and it is the key for flow abstraction. It implements hashCode() and equals(), so it can be used as a key for java hash tables. The five tuple is symmetric, that is if we have a flow between two hosts, the packets of both directions will have the same five tuple. We do so by normalizing the ip's by their values, the bigger IP (unsigned integer 32 bits) is the first and the smaller is the second. We demonstrate the use of FiveTuple by showing an example of how to count the number of TCP and UDP flows in a capture file.

Listing 2.4: "Counting Flows"

```
public static void main(String[] args) throws IOException, NetUtilsException
{
    if(args.length<1)
    {
        System.out.println("missing file name parameter");
        System.exit(-1);
    }

    CaptureIterator it = new CaptureIterator(CaptureFileFactory.
       createCaptureFileReader(args[0]));
    HashSet<FiveTuple> udps = new HashSet<FiveTuple>();
    HashSet<FiveTuple> tcps = new HashSet<FiveTuple>();

    while( it.hasNext())
    {
        byte data[] = it.next();
        if (EthernetFrame.statIsIpv4Packet(data))
        {
            if (IPFactory.isTCPPacket(data))
            {
                tcps.add(new FiveTuple(data));
            }
            else if (IPFactory.isUDPPacket(data))
            {
                udps.add(new FiveTuple(data));
            }

        }
    }

    System.out.println("Total TCP Flows:"+tcps.size());
    System.out.println("Total UDP Flows:"+udps.size());

}
```

The FiveTuple automatically identifies layer three and later four to generate the data. In case the packet is not IP or it is an IP fragment an exception will be thrown.

2.2.4 IPv6

In all the example so far we checked if the frame is of IPv4 and then created the matching object, IPv4Packet,TCPPacket...etc. However, in some cases the packet is TCP, but the underline IP packet is not IPv4, but IPv6. In this case the code we showed so far will not work as we treat it always as IPv4. We can add support for IPv6 in two ways: we can add explicit check for the IP type and then call the correct constructor. Second, we can use IPFactory to create to correct instance for us. We demonstrate the two approaches in the following code segment

```
if (IPFactory.isTCPPacket(data))
{
    if(IPv4Packet.statIsIpv4Packet(data))
    {
        TCPPacketIpv4 tcppkt = new TCPPacketIpv4(data);
    }
    else
    {
        TCPPacketIpv6 tcppkt = new TCPPacketIpv6(data);
    }
    TCPPacket tcppkt = IPFactory.createTCPPacket(data);
}
```

In in the if statement we check the type of the IP. If it is IPv4 we create an TCPPacketIpv4 and if not we create TCPPacketIpv6. A much better way is to use the factory and create TCPPacket instance. Needless to say, the TCP packet is indifference to the underlying IP version (except for checksum calculation). UDP parsing with IPv6 works in a similar way.

Chapter 3

Building packets

The builder package edu.huji.cs.netutils.build is useful when we want to create or modify capture files. We can build TCP or UDP flows from scratch or we can change existing flows that read from capture file or recorded from live network. The builders are divided to the three layers, data, network and transport. We must build all the layers separately and then connect them by adding each layer to its upper layer. For example, add the TCPBuilder to IPv4Buidler which we add to FrameBuilder. After all the layers were connected we can call the function which create the packet (we show an example later in this section). The created packet is an instance of parse packet (see previous chapter). We can configure any field in the packet which is not dependent on the packet. For example, we can configure all the TCP flags, sequence number,...etc, but we can not configure the checksum or the packet length field which are automatically being calculated and set.

We show the use of the builders by building a very useful utility which converts capture file with only IPv4 packets to capture file with only IPv6 packets while preserving the flows. For brevity, we give only the parts of the code that are relevant for the discussion and leave some of the details out. For the complete code look at *edu.huji.cs.netutils.capture.utils.CaptureFileIPv4ToIPv6* class at the library. We skip the capture file opening for read and for write and go directly to the following code:

Listing 3.1: "Dispatching"

```
switch (proto)
{
    case IPPacketType.TCP:
    {
        TCPPacketIpv4 tcppkt = new TCPPacketIpv4(buff);
        TCPPacket pkt = convertTCPIPv4toIPv6(tcppkt);

        wr.addPacket(pkt.getRawBytes(), rd.getTimeStamp());
        counter.myTotalTcp++;
        break;
    }
    case IPPacketType.UDP:
    {
        UDPPacketIpv4 udppkt = new UDPPacketIpv4(buff);
        UDPPacket pkt = convertUDPIPv4toIPv6(udppkt);

        wr.addPacket(pkt.getRawBytes(), rd.getTimeStamp());
```

```
            counter.myTotalUdp++;
            break;
        }
```

Nothing special in the code, we are switching by the protocol type TCP or UDP, then creating the matching object to convert, call convert which returns TCPPacket (which is in fact TCPPacketIPv6) and write the result byte array to the new capture file. In the conversion function we need to build a new TCP packet which uses IPv6 as the underline protocol but preserves the TCP and most of the Ethernet layers. As shown in the following code snippet we use three types of builders EthernetFrameBuilder, IPv6PacketBuilder and TCPPacketBuilder. For each builder we copy the relevant parts form the converted packet. For Example, we copy the converted packet mac addressed to the frame builder. We copy all the tcp flags, sequence number, acknowledge number...etc. As IPv6 address is different from IPv4 address we use a convert function which map uniquely each IPv4 address into IPv6 address.

Listing 3.2: "TCP conversion"

```
public static TCPPacket convertTCPIPv4toIPv6(TCPPacketIpv4 theTcppkt,
    IPv4AddressToIPv6AddressConvert convert)
    {
        EthernetFrameBuilder frame = new EthernetFrameBuilder();
        frame.setDstMac(new MACAddress(theTcppkt.getDstMacByteArray()));
        frame.setSrcMac(new MACAddress(theTcppkt.getSrcMacByteArray()));

        IPv6PacketBuilder ipv6Builder = new IPv6PacketBuilder();
        ipv6Builder.setDstIP(convert.convert((IPv4Address) theTcppkt.
            getUnderlyingIPPacketBase().getDestinationIP()));
        ipv6Builder.setSrcIp(convert.convert((IPv4Address) theTcppkt.
            getUnderlyingIPPacketBase().getSourceIP()));

        TCPPacketBuilder tcpBuilder = new TCPPacketBuilder();
        tcpBuilder.setFlags(theTcppkt.getAllFlags());
        tcpBuilder.setDstPort(theTcppkt.getDestinationPort());
        tcpBuilder.setSrcPort(theTcppkt.getSourcePort());
        tcpBuilder.setSeqNum(theTcppkt.getSequenceNumber());
        tcpBuilder.setAckNum(theTcppkt.getAcknowledgmentNumber());
        tcpBuilder.setWindowSize(theTcppkt.getWindowSize());
        tcpBuilder.setPayload(theTcppkt.getTCPData());

        ipv6Builder.addL4Buider(tcpBuilder);
        frame.addL3Buider(ipv6Builder);

        return tcpBuilder.createTCPPacket();
    }
```

After building the three layers we connect them by calling:
ipv6Builder.addL4Buider(tcpBuilder);
frame.addL3Buider(ipv6Builder);

The last phase is calling tcpBuilder.createTCPPacket(); which returns an instance of TCPPacket (from parse package). Using the returned instance we can access any field. For instance, we can access fields that are automatically calculated, such as checksum. We can print the packet into a

readable text or just get its raw byte array representation. Note that most of the fields of IPv4 header are not copied to the new IPv6 packet. This is because the headers do not have much in common, most of the fields hace completely changed, and the ones that did not changed we can use the default (TTL for example only changed its name to *Hop Limit*).

Chapter 4

Capture file utilities

Capture file utilities is consisted of a collection of classes that support common capture file analyze and editing functions. The classes are built on top of the basic parsing, capture file readers and capture file writers packages.

4.1 CaptureFileFlowAnalyzer

CaptureFileFlowAnalyzer provides a flow abstraction to the capture file. On creation, the entire file is read into memory, a fact which limits it use to a reasonable size capture files (maximum of few hundreds of Mega). While reading the capture file into the memory, the class splits the packets into flows, where only TCP,UDP and other IP flows are considered. None IP packets and IP fragments are considered as single flow respectably. Flows are uniquely defined by their five tuple as explained at 2.2.3. The class provides iterator for accessing the flows and each flow type, which is also represented by class, provides iterator to its packets. The flow class also support some printing functions (to buffer or stdout). The following snippet shows an example of printed flow. By default, only the details which are relevant to the flow level are printed. MAC addresses, headers fields...etc are not printed. An exception are TCP flags which are relevant to the flow establishment. A more detailed printing functionality is available using the parsing package as explained on chapter 2.

```
Packet #1 (1).      Payload length=0.      Protocol=TCP    --->>
-------------------------------------------------------------------------------
--->>64.103.125.235 : 3976 , 72.14.205.189 : 80
-------------------------------------------------------------------------------
[Syn]

Packet #2 (2).      Payload length=0.      Protocol=TCP    <<---
-------------------------------------------------------------------------------
<<---72.14.205.189 : 80 , 64.103.125.235 : 3976
-------------------------------------------------------------------------------
[Syn] [Ack]

Packet #3 (3).      Payload length=0.      Protocol=TCP    --->>
```

```
-------------------------------------------------------------------------------
--->>64.103.125.235 : 3976 , 72.14.205.189 : 80
-------------------------------------------------------------------------------
[Ack]

Packet #4 (4).      Payload length=1260.      Protocol=TCP    --->>
-------------------------------------------------------------------------------
--->>64.103.125.235 : 3976 , 72.14.205.189 : 80
-------------------------------------------------------------------------------
47 45 54 20   - 2F 6D 61 69   - 6C 2F 63 68   - 61 6E 6E 65      GET /mail/channe  -
6C 2F 74 65   - 73 74 3F 61   - 74 3D 78 6E   - 33 6A 33 36      l/test?at=xn3j36  -
37 63 69 71   - 31 35 71 31   - 79 70 37 6D   - 7A 69 32 32      7ciq15q1yp7mzi22  -
7A 64 79 67   - 37 32 78 79   - 26 56 45 52   - 3D 36 26 69      zdyg72xy&VER=6&i  -
74 3D 30 26   - 54 59 50 45   - 3D 68 74 6D   - 6C 26 7A 78      t=0&TYPE=html&zx  -
3D 38 38 36   - 32 32 6F 2D   - 32 77 7A 6A   - 31 38 26 44      =88622o-2wzj18&D  -
4F 4D 41 49   - 4E 3D 6D 61   - 69 6C 2E 67   - 6F 6F 67 6C      OMAIN=mail.googl  -
65 2E 63 6F   - 6D 26 74 3D   - 31 20 48 54   - 54 50 2F 31      e.com&t=1 HTTP/1  -
2E 31 0D 0A   - 41 63 63 65   - 70 74 3A 20   - 2A 2F 2A 0D      .1..Accept: */*.  -
0A 52 65 66   - 65 72 65 72   - 3A 20 68 74   - 74 70 3A 2F      .Referer: http:/  -
2F 6D 61 69   - 6C 2E 67 6F   - 6F 67 6C 65   - 2E 63 6F 6D      /mail.google.com  -
2F 6D 61 69   - 6C 2F 3F 75   - 69 3D 32 26   - 76 69 65 77      /mail/?ui=2&view
```

The CaptureFileFlowAnalyzer uses Wireshark and other common capture files tool convention for counting packets starting with one. The first packet in the capture file is numbered as one (seems logical for gui application, but less straightforward when writing in code). Following this convention the first flow in the capture file is numbered as one as well. Each flow defines packet directions. Each packet is generated by the flow originator or is generated by the flow accepter. This is important when we want to learn something about the patterns of the traffic being exchanged or simply to find if the flow was established correctly (3-way-handshake in TCP). Example for interesting pattern, is the ratio of the total downloaded bytes and the total uploaded bytes of the flow initiator. Was it client-server flow? server-client flow ? or was it a p2p flow?

The CaptureFileFlowAnalyzer supports both IPv6 and IPv4 based flows provided that they are not fragmented (although adding support for fragments is rather simple, I did not needed it). We illustrate the basic use of the class by writing a simple utility which counts the number of UDP and TCP flows in a capture file. The utility also counts the number of TCP flows which has a complete 3-way-handshake in the record (the 3-way-handshake indicates that the flow was recorded from the start).

Listing 4.1: "Count TCP and UDP Flows"

```
public static void main(String[] args) throws IOException, NetUtilsException
{
    CaptureFileFlowAnalyzer ca = new CaptureFileFlowAnalyzer(args[0]);

    int tcp = 0;
    int udp = 0;
    int es_tcp = 0;
```

```
    for (Flow f : ca)
    {
        switch(f.getFlowType())
        {
        case TCP:
            tcp++;
            FlowTCP ftcp = (FlowTCP) f;
            if (ftcp.isEstablished())
                es_tcp++;
            break;
        case UDP:
            udp++;
            break;
        default:
            break;
        }
    }

    System.out.println("Total TCP Flows                 : "+tcp);
    System.out.println("Total Established TCP Flows : "+es_tcp);
    System.out.println("Total UDP Flows                 : "+udp);
}
```

We first create instance of CaptureFileFlowAnalyzer with the name of the capture file to analyze. The constructor throws an exception in case of an error, such as file do not exists. We then, iterate over the flows (using java syntax sugar for iterators) and switch by their type to increment the matching counter. For TCP flow we cast the flow instance to its real type, which is FlowTCP. FlowTCP class has TCP related functionality such as finding if the flow includes the 3-way-handshake (was established). We use this functionally to increment the establishment counter. The information gathered is printed to the stdout and the program terminates. A similar utility, not including the establishment counter, was presented on 2.4. Note how much more work is needed in order to add the establishment counter using only packet abstraction.

In the next utility we find the flow numbers of flows which are of type TCP, they have full 3-way-handshake, and their first payload match a certain regular expression which characterizes HTTP flows . In fact, this is a simple DPI (deep packet inspection) utility. We later can extract those flows using 4.2(flow extraction) and analyze them.

Listing 4.2: "HTTP Flow Finder"

```
// create analyzer
CaptureFileFlowAnalyzer ca = new CaptureFileFlowAnalyzer(args[0]);

// regular expression pattern for HTTP.
Pattern p = Pattern.compile("(GET|POST|HEAD|POST|DELETE|TRACE|CONNECT).*HTTP");

int match = 0;

// iterate flows
for (Flow f : ca)
{
    if (f.getFlowType() == PacketType.TCP)
    {
        FlowTCP ftcp = (FlowTCP) f;
```

```
        // first two conditions
        if (ftcp.isEstablished() && ftcp.isPayload())
        {
            int n = ftcp.getFirstPayloadPacketNum();
            byte data[] = ftcp.getTCPPkt(n).getTCPData();

            // try to match
            Matcher m = p.matcher(new String(data));
            if (m.find())
            {
                System.out.println("Flow #"+f.getFlowNum());
                match++;
            }
        }
    }
}
System.out.println("Total Flows   :"+ca.getNumberOfFlows());
System.out.println("Total Matches :"+match);
```

We use the java regular expression package to find matches on the payload. The signature is just an example, but will probably match most of the HTTP based flows. The utility algorithm is simple, we first create CaptureFileFlowAnalyzer instance, then we iterate over the capture file flows looking for TCP flows with full TCP 3-way-handshake and with at least one payload packet. We then run the regular expression matcher on the payload to see if it is an HTTP flow. If we have a match we print the flow number using the CaptureFileFlowAnalyzer api. In order to get the first payload we first get the first payload packet number in the flow (flow packets are numbered from one as mentioned earlier). Using the packet number we get the actual first payload TCP packet. We get the TCP packet payload by calling the matching TCPPacket access function. We turn the payload byte array into string and feed the matcher. At the end we print the number of total flows in the capture file and the total number of flows which were identified as HTTP. In the next section we show how we can separate those flows from the capture packet.

Example of the utility output. We can see the number of the flows and that we have 57 HTTP flows out of 122 flows (only part of the output is presented)

```
Flow #78
Flow #95
Flow #97
Flow #98
Flow #99
Flow #103
Flow #104
Flow #105
Flow #106
Flow #118
Flow #119
Flow #120
Flow #122
Total Flows   :122
Total Matches :57
```

4.2 Flow Extraction

FlowExtract class gives a functionality for extracting flows from capture files and writing them to a new capture file or to a string buffer as readable text. The flows can be extracted by their number in the capture file or by packet number. Flows are order by their first appearance in the capture file. When using a packet number, we extract the flow which the input packet is part of.

In the following example we use the FlowExtract class to print the first HTTP flow found in the capture using the utility from previous section. We add the following code.

```
if(match == 1)
{
    FlowExtract fe = new FlowExtract();
    fe.loadCaptureFileFlowAnalyzer(ca);
    fe.extractFlowToScreen(f.getFlowNum());
}
```

We load the FlowExtract with the CaptureFileFlowAnalyzer instance to save performance. The FlowExtract can get the capture file name as its input and create its own analyzer which is a total waste in our example, since we already have such instance. If this is the first matching flow, we print its content to the stdout. The class also provides a functionally for writing the flow to a capture file. Part of the flow print out is displayed in the following figure. We can see that all the conditions are met, that is the flow has 3-way-handshake and has a payload packet. The fourth packet which is the first payload contains our regular expression and it is clear that this flow is an HTTP flow (the number inside the parentheses is the global number of the packet in the capture file, In the example the first matching flow packet is the packet number 248 in the capture file).

```
Flow #20
Total packets: 10
Full TCP handshake: true
Payload: true

Packet #1 (248).      Payload length=0.      Protocol=TCP    --->>
-------------------------------------------------------------------------------
--->>72.163.164.210 : 4352 , 64.12.128.81 : 80
-------------------------------------------------------------------------------
[Syn]

Packet #2 (249).      Payload length=0.      Protocol=TCP    <<---
-------------------------------------------------------------------------------
<<---64.12.128.81 : 80 , 72.163.164.210 : 4352
-------------------------------------------------------------------------------
[Syn] [Ack]

Packet #3 (250).      Payload length=0.      Protocol=TCP    --->>
-------------------------------------------------------------------------------
--->>72.163.164.210 : 4352 , 64.12.128.81 : 80
-------------------------------------------------------------------------------
[Ack]
```

```
Packet #4 (251).      Payload length=952.      Protocol=TCP    --->>
-------------------------------------------------------------------------------
--->>72.163.164.210 : 4352 , 64.12.128.81 : 80
-------------------------------------------------------------------------------
47 45 54 20   - 2F 62 65 74   - 61 2E 61 64   - 70 3F 74 79      GET /beta.adp?ty  -
70 65 3D 61   - 69 6D 20 48   - 54 54 50 2F   - 31 2E 31 0D      pe=aim HTTP/1.1.  -
0A 41 63 63   - 65 70 74 3A   - 20 2A 2F 2A   - 0D 0A 52 65      .Accept: */*..Re  -
66 65 72 65   - 72 3A 20 68   - 74 74 70 3A   - 2F 2F 77 65      ferer: http://we  -
62 6D 61 69   - 6C 2E 61 6F   - 6C 2E 63 6F   - 6D 2F 33 34      bmail.aol.com/34  -
30 33 32 2F   - 61 69 6D 2F   - 65 6E 2D 75   - 73 2F 53 75      032/aim/en-us/Su  -
69 74 65 2E   - 61 73 70 78   - 0D 0A 41 63   - 63 65 70 74      ite.aspx..Accept  -
2D 4C 61 6E   - 67 75 61 67   - 65 3A 20 65   - 6E 2D 75 73      -Language: en-us  -
0D 0A 41 63   - 63 65 70 74   - 2D 45 6E 63   - 6F 64 69 6E      ..Accept-Encodin  -
67 3A 20 67   - 7A 69 70 2C   - 20 64 65 66   - 6C 61 74 65      g: gzip, deflate  -
0D 0A 55 73   - 65 72 2D 41   - 67 65 6E 74   - 3A 20 4D 6F      ..User-Agent: Mo  -
7A 69 6C 6C   - 61 2F 34 2E   - 30 20 28 63   - 6F 6D 70 61      zilla/4.0 (compa  -
74 69 62 6C   - 65 3B 20 4D   - 53 49 45 20   - 36 2E 30 3B      tible; MSIE 6.0;  -
20 57 69 6E   - 64 6F 77 73   - 20 4E 54 20   - 35 2E 31 3B      Windows NT 5.1;  -
20 53 56 31   - 3B 20 2E 4E   - 45 54 20 43   - 4C 52 20 31      SV1; .NET CLR 1  -
2E 31 2E 34   - 33 32 32 29   - 0D 0A 48 6F   - 73 74 3A 20      .1.4322)..Host:  -
77 65 6C 63   - 6F 6D 65 2E   - 6D 61 69 6C   - 2E 61 6F 6C      welcome.mail.aol  -
2E 63 6F 6D   - 0D 0A 43 6F   - 6E 6E 65 63   - 74 69 6F 6E      .com..Connection  -
3A 20 4B 65   - 65 70 2D 41   - 6C 69 76 65   - 0D 0A 43 6F      : Keep-Alive..Co  -
6F 6B 69 65   - 3A 20 52 53   - 50 5F 43 4F   - 4F 4B 49 45      okie: RSP_COOKIE  -
3D 74 79 70   - 65 3D 33 30   - 26 6E 61 6D   - 65 3D 54 6D      =type=30&name=Tm  -
6C 7A 61 44   - 55 7A 4E 67   - 25 33 44 25   - 33 44 26 73      lzaDUzNg%3D%3D&s  -
74 79 70 65   - 3D 30 3B 20   - 73 5F 6C 61   - 73 74 76 69      type=0; s_lastvi  -
73 69 74 3D   - 31 32 30 30   - 35 36 33 32   - 32 31 34 35      sit=120056322145  -
```

In the left side we see the packet content as hex bytes displayed in four bytes chunks. On the right we see the same sixteen bytes (single line) as asci string (useful for textual protocols).

4.3 Misc

The library contains CaptureFileConcat which concatenates capture files into one capture file while maintaining the relative packet time stamps. CaptureFileSqueezeTime squeezes the packets time stamps using user configuration or default of one msec. The squeezing is required when we use injection tool which is sensitive to the packet recording time stamps and when the gap between packets is relatively long. We can squeeze capture file which is spread over two minutes to less then one second if it fits our test purposes.

Chapter 5

Injecting and Recording

The injection and recording package adds raw packet accessability. In other words, using the package we can define our own packet and inject it on any of the PC interfaces, we can also listen for specific types of packets on an interface and process any packet which match our criteria. Java only provides the basic sockets for communication, that limits the user control over a specific protocol to payload only, namely the user cannot changes headers. For example, there is no way for sending ARP packets, ICMP packets...etc. On the sniffing side, Java has no API for listing to network traffic which is not intended to the Java application (Socket was opened by the application). Using netutils we can do both. A common functionality for the injection is transmitting a recoded capture file into some network device such as router. A common functionality for recording is filtering relevant network traffic before recoding it in order to save space. There are many tools (WireShark, tcpdump...etc) for both functionalities but they never cover all options as code can. For example, recording only flows that match a certain signature. In the following sections we show few examples on how to inject and record network traffic. We end the chapter by showing how to write ping utility.

5.1 Installation

The package includes 2 *C* libraries libnet.o/libent.lib (Linux or windows format) and netutils.o/netutils.dll. In addition the package contains the java netutils.jar file which contains the java source and classes (compiled for 6.0).

5.1.1 Dynamic Libraries

As mentioned, the java library calls *C* code using java JNI (Java Native Interface). The *C* code is compiled into two shared libraries (Linux) or dll's (Windows). The dynamic libraries should be placed in the java java.library.path (path where the JVM searches for dynamic libraries).

There are two options for placing the libraries:

1. Putting the libraries in one of the default path's, you can use the jave program shown on 5.1 to see the default path.

2. Put the library anywhere and add the path to the java.library.path as shown:

   java -Djava.library.path=place.....

Libraries for Windows XP are supplied as part of the package zip files. For full compilation instructions see section **??**.

Listing 5.1: "Printing Java default java.library.path"

```
public class ShowJavaLibraryPath
{
    public static void main(String[] args)
    {
        System.out.println(System.getProperty("java.library.path"));
    }
}
```

5.1.2 Dynamic Libraries

The netutils.jar should be added to the classpath.

java -Djava.library.path=place -cp lib
netutils.jar;....

5.2 JPcap

The edu.huji.cs.netutils.capture.* includes classes for capturing network traffic on a network interface.

5.2.1 Interfaces

The device network interface must be identified by its name or by its IP. IP identification will work only in cases where the interface has one IP configured. In case that multiple IP's are configured we must use the interface name. The interface name is an internal naming used by the device library and can be found by JPcap as shown 5.2.

Listing 5.2: "Print all interfaces names to standard output"

```
public static void main(String[] args) throws NetUtilsException
{
    JPCapInterface intArr[] = JPCap.getAllIntefacesNames();
    for(JPCapInterface next : intArr)
    {
        System.out.println(next.toString());
    }
}
```

The program output is shown in listing 5.3. In the example we can see that the device has two different network interfaces, each has one IP configured. In this case we can identify each one of the interfaces by its IP or by its unique name.

Listing 5.3: Program output example

```
Name: \Device\NPF_{7300A526-F488-4007-BF77-C0164530524C}
Address: 10.10.20.23

Name: \Device\NPF_{D862664D-B7E1-4381-803D-D101772AC818}
Address: 64.103.125.199
```

5.2.2 JPCap basics

Generally speaking, sniffing is usually done as follows: opening JPCap instance, setting filter (optional), adding listeners and calling sniffing start.

edu.huji.cs.netutils.capture.JPCap class is the basic class for sniffing packets. Each instance must be bounded to a specific network interface as discussed on section 5.2.1. The interface name or IP is provided to the constructor, in case that no specific interface is provided, the JPCap instance will try to bind to the first interface it can find. Once the instance was created successfully, that is without an exception being thrown we can start the sniffing.

The next phase before staring the sniffing is setting the filer which is optional and adding listeners (both can be done after sniffing has started). As said, we can add a filter to the JPCap instance. The filter will cause each packet that does not match it to be dumped by JPCap. Note that JPCap only duplicates the packets and will not affect the traffic in anyway (the operating system will still deliver the packet to its destination regardless of the JPCAP filter). The filter is a syntax filter using tcpdump filter format, for example, putting "tcp port 80" as a filter will raise only TCP packets on port 80 to the handler. All the rest of the packets are filtered. For more information about tcpdump filtering syntax see http://linux.die.net/man/8/tcpdump .

In order to receive packets that have passed the filter, we need to add a listener object to the JPCap instance. The listener object must implement the interface *edu.huji.cs.netutils.capture.JPCapListener* which basically has a single method *public void processPacket(byte[] thePacket);*. This is the basic listener that when used, all frames (Ethernet frame) are received as a raw byte array and the user should parse it (using its own code or *edu.huji.cs.netutils.build.** classes). There are some more high level abstraction classes listeners for TCP/UDP...etc which already transform the packet into the correct class, For example, *edu.huji.cs.netutils.capture.JPCapTCPPktListener*.

We illustrate the usage of JPCap by simple example. In the following example shown on 5.4 we print all TCP packets on port 80 to the standard output.

Listing 5.4: Printing TCP packets on port 80

```
1
2 public class PrintTCPPort80
3 {
4     public static void main(String[] args) throws NetUtilsException, IOException
5     {
```

```
6           String filter = "tcp port 80";
7           String subnet = "255.255.255.0";
8
9           // bind to first interface found
10          JPCap jpcap = new JPCap();
11          jpcap.setFilter(filter, subnet);
12
13          // add tcp listener that only prints the packet content as Hex.
14          jpcap.addListener(new JPCapTCPPktListener()
15          {
16              public void processPacket(TCPPacket thePkt)
17              {
18
19                  System.out.println(thePkt.toHex());
20              }
21          });
22
23          jpcap.startJPcap();
24          System.in.read();
25          jpcap.stopJPcap();
26      }
27
28  }
```

The code is pretty much self explanatory. We open an instance in line 10 without specifying any specific interface, i.e. JPCap instance will be bounded to the first interface found. We set a filer and add a listener using Java anonymous class notation. The listener which extends the JPCapTCPPktListener class implements the process packet method in line 19. In this simple case we just print the packet to the stdout using the TCPPacket class toHex() method. We could have skipped the set filter stage and filter the packet ourself when processing the packet by checking the TCP packet ports using the class access methods. However, using filter will produce better performance as we save the copy of the packet byte buffer to the Java memory and the filtering is done in *C* which is faster (no matter what java groupies say).

```
TCP : src [94.127.72.60:80] , ip [10.56.237.100:2305]
00 50 09 01   - B9 50 63 2D   - 66 DF 78 BB   - 50 10 14 D6   -
84 FF 00 00   - 0C FD 2C 51   - FA 9B 43 F5   - 14 99 41 0D   -
F8 8A A9 C7   - 13 30 9F 49   - 72 43 8D 71   - CD 4A 9C 6C   -
C0 0C 24 33   - C4 88 95 FC   - 4C CF B0 E5   - 19 33 F6 C3   -
49 83 B4 BE   - CC 17 B8 B2   - 83 3C D6 B0   - 56 D6 11 1C   -
AA EC E0 E8   - F9 91 58 0E   - B0 04 2A E6   - 42 35 26 3E   -
```

5.2.3 JInject basics

JInject classes located at *edu.huji.cs.netutils.inject.** package are used for injection of network traffic. As opposed to JPCap where we have a single basic class and we only change the listeners and filters in order to achieve different layer of abstraction, JInject has several classes for injection, each class is used for different layer. When we use a lower layer injector, we have a control on more parameters. Actually, when we use Ethernet injector, we have a control on the entire parameters in the injected frame. Of course, when we use the lower level, for example Ethernet, we need to take care of the

details, such as ARP and putting the appropriate MAC address...etc. There are also mixed layer classes, for example, JInjectUDPLayer2Injector. This is not a network tutorial ,but sometimes we need to set MAC address in UDP packets, for example when sending discover on DHCP protocol.

We have the following injectors:

- EthernetInjector - Provides control over the entire frame including Ethernet MAC address and the carried protocol. Useful when we want to inject none IP protocol, such as ARP, or when we need to have control over the MAC addresses (sending broadcasts for example).
- JInjectIPLayer2Injector - The class provide control over all the IP fields and the underlying Ethernet header.
- IPInjector - used when we want to have a control over the entire IP packet fields, but we want the underlying Ethernet layer to work automatically (setting MAC's, sending ARP's...etc).
- JInjectUDPLayer2Injector - provides a control over UDP fields and Ethernet layer.
- TCPInjector - should be used when we want to inject TCP packets easily. The injector provides control over all the TCP fields.
- UDPInjector - should be used when we want to inject UDP packets easily. The injector provides control over all the UDP fields.

5.2.4 LibPCap file

LibPCap is a format for saving network traffic into capture file used by the common pcap library and the Wireshark free traffic analyzer. All the LibPCap files code is located in package edu.huji.cs.netutils.files.*, where we have two main files PCapFileWriter and PCapFileReader (the package includes readers for additional capture file formats and a factory for creating the correct readers).

We first show an example of writing a LibPCap file. We use the same example from 5.2.2, this time instead of filtering the TCP traffic on port 80 using a filter we do it in the handler. We assume that java args[0] contains a valid file name. No input validation and proper exception handling is done to keep the example concise.

```
1  ....
2  PCapFileWriter fwr = new PCapFileWriter(args[0]);
3
4  // bind to first interface found
5  JPCap jpcap = new JPCap();
6
7  // add tcp listener that only prints the packet content as Hex.
8  jpcap.addListener(new JPCapTCPPktListener()
9  {
10     public void processPacket(TCPPacket thePkt)
11     {
12         if(thePkt.getDestinationPort() == 80 || thePkt.getDestinationPort() == 80)
13         {
14             try
```

```
15              {
16                  fwr.addPacket(thePkt.getRawBytes());
17              }
18              catch (IOException e)
19              {
20                      e.printStackTrace();
21              }
22              catch (NetUtilsException e)
23              {
24                      e.printStackTrace();
25              }
26          }
27      }
28  });
29
30  jpcap.startJPcap();
31  System.in.read();
32  fwr.close();
33  jpcap.stopJPcap();
```

Nothing much to say about the example. The PCapFileWriter also includes an option for appending LibPCap file and limiting its maximum size (set to 1Giga on default). Important note: LibPCap file writer do not flushes the data to the disk on every packet, this is done this way to save performance. Please make sure you close the file writer properly or some of the packets may be lost.

In the next example, shown on 5.5, we demonstrate how to read a capture file written in pcap format. In the example we read a pcap file and print only TCP packets content to the screen. pcap file is built of a file header and then a list of pcap blocks, where each block contains one packet and its capturing information. The capturing information contains the packet length, time stamp ...etc. On line 10 we try to open the file. The constructor will throw IOException in case file wasn't found or any other problem. Once the file was opened we can read it block by block using the method readNextBlock(). The method returns the next block or null if end of file has reached. From the block we can read the frame as a raw byte array and parse it. The code on line 15 demonstrates how to validate that the frame is a TCP/IP packet. Once validated we can construct a new TCPPacket from the raw byte array. Calling the constructor with non valid TCP byte array will cause an exception to be thrown. In chapter 2 we show a more elegant way of iterating a capture file.

Important Note: Reading a file is a very slow IO operation. When trying to inject the packets according to their time stamps it would be much better to first read file into the memory and only then run the injection. Of course, reading the entire captured file into memory is not always possible, but in this case we have to keep in mind the the file reading is our bottleneck.

Listing 5.5: Example of reading PCap file

```
1
2  public static void main(String[] args) throws IOException
3  {
4      if(args.length<1)
5      {
6          System.out.println("missing file name parameter");
7          System.exit(-1);
8      }
9
```

```
10      PCapFileReader frd = new PCapFileReader(args[0]);
11      PCapBlock nextblock = null;
12      while( (nextblock = frd.readNextBlock()) != null )
13      {
14          byte data [] = nextblock.getMyData();
15          if(EthrenetPacket.statIsIpPacket(data) && IPPacket.getIpProtocolType(data)
                == IPPacketType.TCP)
16          {
17              TCPPacket tcp = new TCPPacket(data);
18              System.out.println(tcp.toString());
19          }
20      }
```

5.2.5 Building and parsing frames

The package *edu.huji.cs.netutils.build.** contains infrastructure code for parsing and building Ethernet, IP, TCP, UDP and ICMP frames/packets/segments. This initial set of protocols covers the most common protocols in the Internet (based on TCP/UDP) and could be easily extended in the future by writing new parse/build java classes for new protocols. When using *edu.huji.cs.netutils.build.** classes, we have to separate between sniffed packets and built packets.

Parsing sniffed packets

Sniffed packets, namely packets that were sniffed by JPcap instance and were passed to listener, or packets that we read form a capture file, always contain a complete frame (frame is the Ethernet layer bytes). In other words, we have a byte array which includes all the network layers. For example, the fourteen first bytes are the Ethernet header (assuming no Vlan is present) and the rest belongs to the above protocol (ARP, IP...etc). In case the next layer is IPv4, we have twenty bytes that consist the IP header (in fact, it may be longer, depends on the IP options, which are not present often) and the rest of the bytes are the above protocol (UDP, TCP ...etc). If we use the simple listener, we will get a byte array that contains a complete packet. The following code shows and example of parsing. The class presented in 5.6 implements a listener that parse Ethernet frames and check whether the network layer is TCP. In case of a TCP segment the listener prints the segment data length (payload) to the stdout.

Listing 5.6: Print TCP segments length

```
1
2   public class PrintTCP implements JPCapListener
3   {
4
5    @Override
6    public void processPacket(byte[] thePacket)
7    {
8       // first we check if this frame carries IP packet
9       if(EthrenetPacket.statIsIpPacket(thePacket))
10      {
11          // we check the protocol type is TCP
12          if(IPPacket.getIpProtocolType(thePacket) == IPPacketType.TCP)
13          {
```

```
14              TCPPacket tcpPkt = new TCPPacket(thePacket);
15              System.out.println("TCP segment payload length:"+tcpPkt.
                    getPayloadDataLength());
16          }
17      }
18
19  }
20
21  }
```

The processPacket method is called on each frame that has been passed the JPCap filter (if filter was configured) see 5.2.2. When processing a packet we first want to determine its frame protocol type, validating that this is an IP protocol packet. If the packet is indeed an IP packet then we can continue and find its IP protocol type as done on line twelve. If the packet is TCP segment then we create a new TCPPacket instance with the packet buffer as a parameter and use the TCPPacket accessor methods for getting the TCP payload length.

Important Note: When creating a new TCPPacket, UDPPacket...etc using byte[] buffer the byte buffer is not duplicated, and therefore any changes that are made in the object would be reflect at the underlying byte buffer as well. In other words, if we create two TCPPacket objects using the same input buffer, then any change that is made in one of the TCPPacket objects will change the other TCPPacket object as they share the same buffer. In that case, a duplication of the buffer should be handed in the constructor. The buffers are not duplicated because of performance consideration. When we want to modify packet it is better to use packet builder as described on chapter 3

Building sniffed packets

Usually we will build packets for injection purposes. When building a packet, we first chose the level of abstraction. If, for example, we want to build a TCP packet then we will create new TCPBuilder object as described at 3. When using an upper level abstraction, we have a more simple code as some of the details will be taken care by the object. For example, we can leave the mac addresses, checksums...etc empty and they will be filled by the injector. Of course, if we need to control any field, we still gain the simplicity as the TCPBuilder class already has access methods for setting them.

We show an example of how to build a TCP SYN packet.

Listing 5.7: Send TCP SYN

```
1
2  public static void main(String[] args) throws InterruptedException
3  {
4      try
5      {
6
7          // init injector using first interface name
8          TCPInjector inj = new TCPInjector(JPCap.getAllIntefacesNames()[0].getName())
               ;
9
10         // build TCP part
11         TCPPacketBuilder tcppkt = new TCPPacketBuilder();
```

```
12          tcppkt.setSrcPort(4000);
13          tcppkt.setDstPort(4001);
14          tcppkt.setSYNFlag(true);
15          tcppkt.setSeqNum(0);
16
17          // IP part
18          IPv4PacketBuilder ipv4 = new IPv4PacketBuilder();
19          ipv4.setSrcAddr(new IPv4Address(inj.getIp()));
20          ipv4.setDstAddr(new IPv4Address("64.103.125.91"));
21
22          // connet IP and TCP (Ethetnet is auot completed)
23          ipv4.addL4Buider(tcppkt);
24
25          inj.injectTCP((TCPPacketIpv4) tcppkt.createTCPPacket());
26
27          inj.releaseResource();
28          //pcap.stopJPcap();
29      }
30      catch (NetUtilsException e)
31      {
32          e.printStackTrace();
33      }
34  }
```

As explained in chapter‘3 we build TCP segment by building the transport layer and the network layer separately and then connect the layer. We skip here the FrameBuilder (data layer) and let the code auto complete it. In fact, in this scenario we use TCP injector which means that the Ethernet layer would be taken care by the injector regardless the information we provide.

In the following figure we see the output of WireShark application recoding of the traffic. The seven packet it our injected packet. We can see that it was responded with RST.

No.	Time	Source	Destination	Protocol	Info
1	0.000000	171.71.179.143	10.56.237.101	TCP	https > 4785 [ACK] Seq=1 Ack=1 Win=23 Len=0
2	0.000066	10.56.237.101	171.71.179.143	TCP	[TCP ACKed lost segment] 4785 > https [ACK]
3	8.704231	74.125.230.152	10.56.237.101	TLSv1	Application Data, Application Data
4	8.858008	10.56.237.101	74.125.230.152	TCP	ds-mail > https [ACK] Seq=1 Ack=80 Win=3222
5	14.467204	Cisco_3c:78:00	Broadcast	ARP	Who has 64.103.125.91? Tell 10.56.237.101
6	14.467271	00:26:f2:50:d4:92	Cisco_3c:78:00	ARP	64.103.125.91 is at 00:26:f2:50:d4:92
7	14.467346	10.56.237.101	64.103.125.91	TCP	terabase > newoak [SYN] Seq=0 Win=65535 [TC
8	14.494759	64.103.125.91	10.56.237.101	TCP	newoak > terabase [RST, ACK] Seq=1 Ack=1 Wi

Chapter 6

Examples

In this chapter we show few more complex examples of using netutils. All the examples code are included as part of the package. All the code was checked on Windows XP using the provided dll's and should work on Linux as well.

6.1 Ping

Ping is a well known utility for checking whether a host in the network is reachable. The ping is based on the ICMP (Internet Control Message Protocol) EHCO message type. ICMP is a very simple protocol, its header is built of only three fields Type,Code and Checksum.

For ping message, we set the message type to ECHO REQUEST(8), Code is not relevant here (we set it to zero) and checksum is done as usual by a 16-bit one's complement of the one's complement sum of the ICMP message bytes, starting with the Type field. The checksum field should be cleared to zero before generating the checksum (see ICMP class implementation). The data of the ECHO message is just a time-stamp, sequence and id generated on the originating host. On receiving ECHO REQUEST ICMP message, the host should return ECHO REPLY(0) ICMP message with the received payload (the payload contains the times tamp). On reply, the sending host extracts the time-stamp from the returned message and calculate the total time it took.

In the following example we will implement a very simple ping utility. Again, we omit the "proper programming" details, such as input validation, and concentrate in the library usage. The ping utility works as follows:

1. Create JPCap instance with ICMP listener for getting the reply's.
2. Create IPInjector for sending ICMP messages.
3. Build and send ICMP ECHO REQUEST.
4. In the listener we check if the ICMP message type is ECHO REPLY and if it is then we extract the time-stamp and print to total time.

Since the code it a little bit longer then the usual examples we split it to two. First we show the listener code.

```
class PingListener extends JPCapICMPListener
{
  private int myResponses = 0;

  private long myIP = 0;

  /**
  *
  * @param theIp
  */
  public PingListener(String theIp)
  {
    myIP = IP.getIPAsLong(theIp);
  }

  public void processPacket(ICMPPacket thePkt)
  {
    try
    {
      // check that the packet is a replay
      if (thePkt.getICMPType() == ICMPPacketType.ECHO_REPLY_TYPE)
      {
        // make sure this is a response from the correct host
        if(thePkt.getSourceIP() == myIP)
        {
          // parse the data
          PingData data = new PingData(thePkt.getICMPData());

          // the time took will be the current time - the time in the data arrived.
          System.out.println("got replay from " + IP.getIPAsString(thePkt.
              getSourceIP()) + ",time = " + (System.currentTimeMillis() - data.time)
               + " msec , ttl = " + thePkt.getTTL());
          myResponses++;
        }
      }
    }
    catch (RuntimeException e)
    {
      e.printStackTrace();
    }

  }

    /**
     *
     * @return number of responses.
     */
    public int getTotalResponses()
    {
        return myResponses;
    }
```

```
}
```

The PingListener class extends JPCapICMPListener abstract class, which filter ICMP packets. We could use the basic JPCAPListener, but then, we should check the packets types and make the conversion into ICMPPacket ourselves.

Once an ICMP packet has arrived to the handler, we first check its type to see if this is an ECHO REPLY. Note that any ICMP packet that is sent or received by the interface, including the ICMP ECHO REQUEST packets that we generate in the ping utility (our sent packets), are forwarded to the listener (generally, any listener will get both outgoing packets and ingoing packets).

Once we have verified that the packet is an ECHO REPLY message, we check that the message source IP indeed belongs to the correct host (maybe this is a response for other interface), namely this is the host that we are trying to 'ping'. If the host IP matches then we print the message details to the stdout. The printed details include the host source IP,the time it took and the TTL value. The PingData class is just a utility class for converting or extracting the different fields (time-stamp,id ...etc) into or form a continuous byte array.

Section of the main code is shown on 6.1. We first open JPcap instance, add the PingListener, and call start. On line twenty six we open an IPInjector which is bounded to same network interface as the JPCap. We use the ICMPPacketBuilder class and IPv4PacketBuilder to build ICMP ECHO REQUEST packet. The interface IP is set as the source IP, as we want the packet to return to this interface. The destination host is set to the pinged host (received as a parameter). We send the ICMP messages in a loop, where the sequence and id are increased on each packet. In this simple 'ping' utility we don't really check the sequence and id as we should have done. When time out elapsed we gracefully close all instances.

Listing 6.1: Ping main

```
1  if(args.length>1)
2  {
3      System.out.println("Missing parameter: destination IP");
4      System.exit(-1);
5  }
6  String targetIp = args[0];
7
8  // number if icmp echo to send
9  int count = DEFAILT_COUNT;
10
11 if(args.length == 2)
12 {
13     count = Integer.parseInt(args[1]);
14 }
15
16 // initialize the jpcap on first interface found
17 JPCap sniffer = new JPCap(JPCap.getAllIntefacesNames()[0].getName());
18
19 // add ping listener
20 sniffer.addListener(new PingListener(targetIp));
21
22 // start sniffer
23 sniffer.startJPcap();
24
25 // init Ip injector on the same interface as the sniffer.
```

```
26 IPInjector inj = new IPInjector(sniffer.getMyInterfaceName());
27
28 // building the ip layer
29 IPv4PacketBuilder ipv4 = new IPv4PacketBuilder();
30 ipv4.setDstAddr(new IPv4Address(targetIp));
31 ipv4.setSrcAddr(new IPv4Address(sniffer.getmyInterfaceIp()));
32
33 // build the data. the data is any byte array.
34 // in the implementation it contains pkt id,sequence and time stamp
35 PingData pingdata = new PingData();
36 pingdata.id = 1;
37 pingdata.sequence = 200;
38 pingdata.time = System.currentTimeMillis();
39
40 // build the icmp packet
41 ICMPPacketBuilder icmp = new ICMPPacketBuilder();
42 // the type is echo request
43 icmp.setType(ICMPPacketType.ECHO_REQUEST_TYPE);
44 icmp.setCode(0);
45 // set the data
46 icmp.setPayload(pingdata.getAsByteArray());
47 ipv4.addL4Buider(icmp);
48
49 // inject according to the wanted count number
50 for (int i = 0; i < count; i++)
51 {
52     inj.injcet( icmp.createICMPPacket().getIpv4Packet());
53     sleep(1000);
54
55     // change the sequence and the time stamp for next ping
56     pingdata.sequence++;
57     pingdata.time = System.currentTimeMillis();
58     icmp.setPayload(pingdata.getAsByteArray());
59 }
60
61 // stop injector
62 inj.releaseResource();
63
64 // give the sniffer 5 more sec (for last replay to arrive )and close it.
65 sleep(5000);
66 sniffer.stopJPcap();
```

An example of the utility outpt:

```
got replay from 10.56.217.163,time = 421 msec , ttl = 252
got replay from 10.56.217.163,time = 344 msec , ttl = 252
got replay from 10.56.217.163,time = 343 msec , ttl = 252
got replay from 10.56.217.163,time = 312 msec , ttl = 252
```

6.2 Port Scan

Port scan is a simple utility for finding all the open ports on a host. Open port means that the operating system is excepting to open TCP connections on this port. The port scanner utility works as follows: it sends a syn packet on each port and waits for syn-ack (see TCP RFC, 3-way-handshake). If syn-ack arrives then the port is marked as open.

The relevant parts of the code are shown at 6.2, the complete example is included in the netutils package. The utility receives the source IP and destination IP as parameters (not shown here). On line seven we see an example of filter use. Only packets from the destined host would be dispatched to the listener (see section 5.2.2. In fact, we could also specify in the filter that only TCP packets with syn flag should be dispatched, but we want to show how to do it in code.) We use the anonymous class notation when adding a JPCapTCPPktListener. The code in the listener is self explanatory, we check if the TCP packet has an ACK flag and a SYN flag and print the port on match (we know this the correct host because we used host filter).

Lines 34-45 shows how to build a SYN TCP packet. We configure only the mandatory fields (destination and source IP, destination and source ports) and the TCP SYN flag. The rest of the TCP packet is auto completed. We inject the SYN packets in a loop, while increasing the destination port on each iteration. Note that the packet is not duplicated on sent and therefore we have a single packet which we change. Since the sent method is blocking, that is calling the method will block until injection completion, this is not a problem. A simple sleep is added between iteration to slow the scan making it more reliable. When scan ends, we gracefully close all resources.

Listing 6.2: Port Scan

```
1  try
2  {
3      // open sniffer on the first interface found
4      JPCap sniffer = new JPCap();
5
6      // set filter on the dst ip
7      sniffer.setFilter("host "+dstip,IP.getIPAsLong(mask));
8
9      // create tcp listener
10     sniffer.addListener(new JPCapTCPPktListener(){
11
12             /**
13              *
14              */
15             public void processPacket(TCPPacketIpv4 thePkt)
16             {
17                 // if the packet ack falg is on
18                 if (thePkt.isAck())
19                 {
20                     // if the packet syn ack is on
21                     if (thePkt.isSyn())
22                     {
23                         System.out.println("Source IP: "+thePkt.
                               getUnderlyingIPPacketBase().getSourceIPAsString());
24                         System.out.println("Got syn,ack on port "+thePkt.
                               getSourcePort()+", "+thePkt.getDestinationPort());
25                     }
```

```
                }
            }});

    // start the sniffer
    sniffer.startJPcap();

    // init the injector on the same interface as the sniffer
    TCPInjector inj = new TCPInjector(sniffer.getMyInterfaceName());

    // build the tcp packet
    IPv4PacketBuilder ipv4 = new IPv4PacketBuilder();
    ipv4.setDstAddr(new IPv4Address(dstip));
    ipv4.setSrcAddr(new IPv4Address(srcip));

    TCPPacketBuilder tcp = new TCPPacketBuilder();
    tcp.setSrcPort(4000);
    tcp.setSYNFlag(true);
    tcp.setSeqNum(0);

    ipv4.addL4Buider(tcp);

    // run on all port wanted range
    for(int i=0,port=1 ; i<MAX_PORT ; i++,port++)
    {
        tcp.setDstPort(port);
        //pkt.setDestinationPort(port);
        inj.injectTCP((TCPPacketIpv4) tcp.createTCPPacket());

        // every 100 syn sleep for a while
        // make sure not overloading the target pc
        if (i%100==0 && i>0)
        {
            Thread.sleep(100);
        }
    }

    // clean the injector and the sniffer
    inj.releaseResource();
    sniffer.stopJPcap();
}
catch (Exception e)
{
    e.printStackTrace();
}
```

For bugs or questions email roni.bar.yanai@gmail.com