MAC layer functions for SLEF

Lorenzo Keller <lorenzo.keller@epfl.ch>

October 20, 2006

Contents

1 Requirements

1.1 Pseudo broadcast

Pseudo broadcast is a technique used to improve throughput of broadcast transmissions in case of congested networks. The mechanism consists in sending a

packet in unicast to a station using RTS/CTS. Other stations will receive the packet by capturing all the frames that are transmitted on the network, even if they are not directed to them.

1.2 Controlled injection rate in the MAC layer

The injection rate of packets in the MAC layer has to be controlled. The application must not be allowed to deliver to the MAC layer more packets than the number that can be sent by the network adapter. It is also necessary to know the nominal rate of the network.

1.3 Monitoring of other stations in range

An indication of activity on the network has to be provided. This function has to detect the activity of other SLEF stations in the neighborhood. The address of the last transmitting station and the time of transmission have to be provided.

2 Implementation

2.1 Structure

The required function are implemented using jpcap, a library that allows to send and receive raw Ethernet frames form Java. Jpcap relies on pcap[1] , a C library that provides an high level interface to packet capture systems. The structure of the implementation can be seen in figure 1. Pcap is OS specific but it has already been ported on many OSs[2]. This library is included in standard Linux distributions and an interactive setup for Windows is available. Since this library gives access to very low level and security sensitive functions administrative rights are required to use it. Performance tests shows that it is possible, using this mechanism, to achieve the same throughput of standard UDP[3].

It was necessary to use this library to be able to implement channel activity monitoring and reception of pseudo broadcast. With standard Java Sockets it is not possible to receive packets that are sent between other stations. TCP/IP stack never receives unicast frames that are addressed to other stations because they are discarded by the network card[4].

Since the application is not using IP sockets it is not necessary to send real IP frames. Simple Ethernet frames are used in the implementation, . The type field of the Ethernet header is different from the one used by IP therefore these frames do not interfere with other applications that could be running on the same station or on other stations of the same network. This approach has the interesting side

[1] See `pcap(3)` on systems where it is installed and http://www.tcpdump.org/
[2] among them Linux and Windows
[3] See the corresponding section in this report
[4] The topic is explained in details in the corresponding sections of this report

effect that IP networking has not to be initialized on the station, no IP address is necessary.

2.2 Pseudo broadcast

2.2.1 RTS/CTS

In order to use the RTS/CTS mechanism the station has to be configured accordingly. The 802.11 standard specifies a variable in the MIB (Management Information Base) called `dot11RTSThreshold`[5]: packets with a size smaller than this value are sent without RTS/CTS. By default the threshold is set to a value that completely disables the mechanism. The parameter is accessible by OS specific interfaces.

Under Linux 2.6 the parameter can be set with a command line utility[6] . The OS provides a programmatic access to this parameter via two `ioctls`[7].

Windows NDIS (Network Driver Interface Specification) 5.1[8] provides an API to access some parts of the 802.11 MIB, parameters can be managed via OIDs (object identifiers). They can be queried from user space through the WMI API (Windows Management Instrumentation) but it is not possible to change them from user space. This requires the creation of a new device driver or the use of a non public interface[9]. This API is not very useful because the new WiFi configuration client used in Windows XP conflicts with this kind of kernel drivers and because updated drivers for the wireless cards are necessary[10]. A more successful approach is to use driver specific settings to control some parts of the MIB[11]. These are accessible from the device properties window and form the registry, allowing a programmatic access.

In the implementation it was decided to create an additional library that allows to change the RTS threshold under Linux. An even simpler solutions however is not to use this library and simply run the configuration command in the application startup script. An example of such script can be viewed in listing 1

For Windows the user of the application has to change the parameters of its card by hand since it is not possible to have a generic method to do it.

2.2.2 Reception of unicast packets for other stations

Reception of unicast packets that are sent to other stations is not supported by the normal network stack. NICs normally discard packets that are not directed to their address nor to broadcast address. This behavior avoids useless interrupts for

[5] See page 483 802.11-1999

[6] `iwconfig [interface name] rts [value]` (for more informations see `iwconfig(8)`)

[7] See `wireless.h` (`SIOCSIWRTS`, `SIOCGIWRTS`)

[8] Available in Windows XP SP1 and Windows 2003

[9] NDISUIO, this interface is not stable and has changed between service packs

[10] Toshiba wireless cards for instance don't have such driver

[11] Toshiba wireless cards based on Orinoco don't provide this kind of option, but for instance U.S. Robotics 22Mbps cards provide them

packets that are not for the station. In order to capture packets that are sent to other stations it is necessary to set the NIC to use a special promiscuous mode. This feature is supported by most NICs[12]. In promiscuous mode only data frames can be captured. Management frames like RTS and CTS are not captured. Frames captured are normally translated to normal Ethernet frames: it is not possible to capture 802.11 specific headers[13].

Capturing packets in promiscuous mode is not supported by Java. The APIs are OS specific. On Linux and on most Unix systems the kernel provides a special socket type to do raw operations[14], this socket has an option that can used to enter promiscuous mode[15]. Windows don't provides a socket type to do frame sniffing. An NDIS intermediate device has to be implemented an inserted in the kernel.

Packet capture is a very common task and for this reason the pcap library has been developed. Pcap interfaces with the different OS specific mechanisms and provides an high level API. This library is available for all the major OSs. The standard bindings are in C but two Java binding have been developed, one from Keita Fujii of University of California Irvine[16] and the other by Patrick Charles[17]. The UCI implementation of the bindings seems more actively maintained and therefore was selected for the project.

2.3 Controlled injection rate in the MAC layer

2.3.1 Channel maximal rate

When a station wants to start a new IBSS it uses `MLME-START.request`[18]; it has to specify two set of rates, one, called `BSSBasicRateSet`, has to be supported by all the stations that want to join and the other, called `OperationalRateSet`, is a superset of the first and contains all the supported rates of the IBSS.

When a station wants to join a BSS it uses the `MLME-JOIN.request`[19] primitive. With this primitive a set of operational rates supported by the stations is specified.

All management, broadcast and multicast frames have to be transmitted at a rate contained in `BSSBasicRateSet`. All CTS and ACK have to be transmitted at the rate of the immediately previous frame when this rate belongs to PHY mandatory rates, or otherwise at the highest possible rate in the `BSSBasicRateSet`.

An internal algorithm has to select a rate for the transmission of the other frames. This rate is stored in an internal variable called `MACCurrentRate`. This

[12]Some Cisco 802.11 cards with standard drivers cannot be put in this mode

[13]Some cards can be forced to capture management frames and 802.11 headers but this function is not widely supported

[14]The name of the socket type depends on OS and version, under Linux 2.6 it is `PF_PACKET` (see `packet(7)`)

[15]In Linux 2.6 its name is `PACKET_MR_PROMISC` (see `packet(7)`)

[16]http://netresearch.ics.uci.edu/kfujii/jpcap/doc/index.html

[17]http://jpcap.sourceforge.net/

[18]See 802.11-1999 § 10.3.10.1.2

[19]See 802.11-1999 § 10.3.3.1.2

variable is not readable from the MAC upper edge interface nor is the rate at which frames are received.

In order to be sure of the nominal PHY rate the two sets of supported rates have to contain only one value. This can be achieved by setting the NIC MIB accordingly. The same API discussed for the RTS threshold parameter applies in this case. Under Linux the rate can be set and queried via a command line utility[20] or through ioctls[21]. Under Windows NDIS OIDs are provided and some drivers provides proprietary setting to change their rates, but it is not in general possible to set a fixed rate.

The implementation contains a library that can change and retrieve the rate of a wireless card, however this runs only on Linux. Like with the RTS threshold parameter, the use of the library is not advised. A simpler solution is to use a script like the one that is shown in listing 1 at the application startup. Under Windows the user has to change manually, if available, the parameter.

2.3.2 Rate control

If the send method blocks until the frame is sent, it is guaranteed that the injection rate in the MAC layer is not higher than the maximal channel rate. This mechanism to control the rate was used under Windows because the send method of pcap on this platform follows this behavior [22]. Unfortunately any error code is returned in case of transmission failure.

Under Linux the networking code in the kernel is different. Each socket has a send buffer which will contain packets waiting to be sent. The default size of this buffer is specified by the kernel parameter `net.core.wmem_default`. The size is measured in bytes and can be set per socket with the socket option `SO_SNDBUF`[23], it is possible to changed it at any time. Moreover in the kernel a queue discipline (`qdisc`), a module that manages the scheduling of packets[24] is associated to each network device. By default the `qdisc` is `pfifo_fast`: this discipline is composed by three queues which contains packets with different priorities. Packets from lower priority queues are sent only when higher priority queues are empty. The size of the bands can be controlled by a command line utility[25] or via ioctls[26] and it is measured in packets. Other types of disciplines can be installed in the kernel with the same utility.

The send function under Linux blocks until there is sufficient space in the socket buffer to store the new packets. After a packet has been stored in the buffer the send function tries to enqueue its transmission; the `qdisc` can either

[20] `iwconfig [device name] rate [rate] fixed`

[21] `SIOCSIWRATE, SIOCGIWRATE`

[22] Third party intermediate devices (like packet schedulers, firewalls or VPN clients) could change this behaviour

[23] it is not possible to change this options directly from pcap, OS specific code has to be written

[24] Some software network devices can be without `qdisc`

[25] `ifconfig [device name] txqueuelen [length]`

[26] `SIOCGIFTXQLEN, SIOCSIFTXQLEN`

accept or refuse the packet. The send function then finishes with a return value corresponding to the `qdisc` decision. In the case the packet is refused by the `qdisc` it is immediately removed from the socket buffer. The time at which the transmission will happen is not known by the caller of send function neither is the acknowledgement. A packet in the queue will never be discarded[27]. On the other side of the `qdisc` the network device driver dequeues packets and sends them to the card firmware for transmission. When transmission is completed the network card generates an interrupt that triggers the deallocation of the packet from the socket buffer.

The default value of the queue lengths is 1000 packets and the default buffer size is 107520 bytes[28]. This means that the buffer can contain less than 1000 packets bigger than ˜110 bytes and therefore if only one application is running on the station and is using that packet size the send call will always block. In other conditions the two parameters have to be tuned.

The jpcap library throws the same exception for any kind of transmission error[29], for this reason it is not possible to treat in a special way queue drops. The code that was implemented immediately aborts on errors when doing unicast. For broadcast, in case of error, the implementation retries a user configurable number of times the transmission before throwing an exception. Unicast transmission is tried only if a fresh MAC address is available, this is necessary because failed transmissions, very probable in case of MAC addresses sensed long time, before are not detectable. The complete algorithm can be seen in figure 2. The dashed line represent a path that not is taken in case when the destination station is not available.

2.3.3 Retries

At MAC layer unicast packets are retransmitted for a specific number of times in case of error. This parameters is part of the MIB but there are no widespread methods to set it[30]. Pseudo broadcasts sent to non reachable stations are therefore retired many times. After a packet has been delivered to the MAC layer it is impossible to change it, for this reason it is impossible to change its destination address between retries. When RTS is not used packets that are retransmitted are received multiples times by the stations sniffing the network [31]. With RTS enabled a frame is sent only after a CTS is received therefore packets sent to non reachable stations are never seen by other stations. In this case since Linux don't provide any acknowledgment of transmission to the applications it is not possible to detect that an unicast packet has not really been transmitted.

In the implementation no attempt to change the number of retries is done.

[27] but its transmission could be delayed indefinitely

[28] Fedora Core 5 kernel 2.6.17-1.2187_FC5

[29] packet is bigger than the MTU, packet is not acknowledged, packet is dropped by the `qdisc`,...

[30] NDIS 5.1 don't provide an OID to set this property; on Linux this option is configurable with `iwconfig [device] retries [number]` but changing this property is not supported by all cards

[31] Tested with a Windows machine transmitting and a Linux machine receiving

2.4 Monitoring of other stations

Monitoring network activity is implemented with the same mechanism used to receive pseudo broadcast packets. In order to detect only stations that are using the SLEF protocol a filter is used: only frames with the correct Ethernet type id are captured.

This method can't detect management frames. Monitoring this kind of traffic requires a lower level access to network devices that is not widely supported [32]. Sniffing management frames is anyway not generally advisable because this traffic could be generated by other applications and not indicate real SLEF activity on other nodes.

In the implementation the wireless interface is put in promiscuous mode and all data frames of the Ethernet type corresponding to SLEF are captured. For every packet received an internal variable holding the source MAC address and another holding a timestamp are updated.

3 Performance comparison of UDP and jpcap

Since SLEF needs an efficient use of the channel jpcap was tested in order to verify that it will not be significantly slower than the standard IP stack. The objective of the comparison was to check if one of the two stacks can achieve a bigger maximal throughput than the other.

The following factors were considered significant for the comparison: packet size and operating system.

Other factors influence the absolute value of the test but should not change significantly the relative performance of the different methods:

1. MAC Protocol settings: RTS, fragmentation, short preamble, WEP, Mode
2. Signal conditions (noise level, signal level)
3. Load of the receiving / transmitting system
4. Other traffic on the network
5. Transmission rate
6. System performance
7. Wireless card model

In the test a station sent packets as fast as possible to a receiver. The performance metric used in the test was the number of bytes of application data received per second.

A fact needs to be remarked jpcap can send raw Ethernet frames, so it has a smaller overhead than UDP. The length of a Jpcap header is 34 bytes, the one of each UDP packet is 62 bytes.

[32] Under Linux and Windows monitor mode is available only on specific drivers

3.1 Test setup

For the test the following computer were used:

1. Computer A: Pentium IV, 512 MB Ram, Dell TrueMobile 1150 (orinoco), Fedora Core 5 (kernel 2.6.17-1.2187)

2. Computer B: Pentium IV, 512 MB Ram, U.S. Robotics 2216 (acx100), Windows XP Home

The two systems had a minimal number of process running and were not generating any significant amount of network traffic.

The MAC and PHY layers were configured as follows:

1. RTS threshold: 10 byte

2. Fragmentation threshold: never

3. Preferred TX rate: 11Mbps

4. Short preamble: off

5. WEP: off

6. Mode: ad-hoc

A scan of the neighborhood showed that channel 11 and 6 were quite busy. On channel 1 only one network was present and didn't show much traffic. This channel was used to carry out the test. The two computers were the only nodes in the IBSS.

3.2 Acquisition of performance data

The test was done with two Java programs, one that sent data and one that received it. The test was done once with computer A transmitting and computer B receiving and then in the other direction to test the impact of different operating systems. This is not a complete analysis of the operating system factor but already gives a good picture of the situation.

The test was carried out with packet sizes varying from 100 to 1300 bytes of application data.

The algorithm of the test was the following:

1. The sender first sends a control packet to the receiver to inform of the packet size that will be used for the test.

2. The sender waits 10 seconds.

3. It sends data for 30 seconds.

4. Finally it waits other 10 seconds.

Sender and receiver polled internal counters every second and wrote them to a log file. Computer A logged channel condition during the test; this was done only on computer A because the method used to read data is available on Linux only.

3.3 Analysis of the results

3.3.1 Channel conditions

Uniform channel conditions have to be guaranteed across tests in order to have a fair comparison. During the four test the conditions were the following:

The signal level is almost always the same, considering that the values are recorded with integer precision. The noise level is more variable but the upper level is always similar. There are some interferences that are not constant in time but considering that the signal level is always higher than the maximal noise level it possible to say that interferences were never a problem in any of the tests. From inspection of the error counters on computer A it is possible to see that these consideration are correct, indeed no significant transmission problems were detected; more than 4 millions packets were sent, the number of detected errors was of order of 10. From these considerations it is possible to say that the conditions during the test on the channels were uniform.

3.3.2 Performance results

The rate is computed over a period of 20 seconds in the middle of the test in order to avoid transients conditions. The graphs in figure 6 and figure 5 show the data. UDP and jpcap approaches have similar performances. The difference in throughput when computer B sends UDP packets of length 1300 bytes is due to the fact that Windows starts fragmenting UDP packets at that size.

The results clearly shows that there is no significant performance loss while using jpcap.

4 Sample code using the library

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStreamReader;

import ch.epfl.lca.slef.mac.SlefMAC;
import jpcap.JpcapCaptor;
import jpcap.NetworkInterface;
import jpcap.NetworkInterfaceAddress;
```

```
import jpcap.PacketReceiver;
import jpcap.packet.EthernetPacket;
import jpcap.packet.Packet;

/*
 * Sample application
 */

public class Sample implements PacketReceiver {

    private SlefMAC mac;

    public static void main(String[] args) {
        Sample s = new Sample();
        s.start();
        s.send(new byte[100]);
    }

    /* send some data */
    private void send(byte[] bs) {
        try {
            mac.sendPacket(bs);
        } catch( IOException ex) {
            System.out.println("The␣OS␣refused␣the␣packet,␣another␣
               application␣on␣this␣computer␣is␣probably␣using␣too␣
               much␣aggresively␣the␣network");
        }
    }

    /* start receiving packets */
    private void start() {
        mac.start();
    }

    public Sample() {
        try {
            mac = new SlefMAC(this, 1000, 1000, 10);
        } catch (Exception ex) {
            try {
                /* autodetection has failed try to manually select
                   an interface */
                NetworkInterface[] devices = JpcapCaptor.
                   getDeviceList();
                System.out.println("Select␣one␣of␣the␣following␣
                   interfaces");
```

```
                for (int i = 0; i < devices.length; i++) {
                    System.out.println(i+" :"+devices[i].name + "("
                        + devices[i].description+")");
                }
                System.out.println("Select the card (insert -1 for
                   automatic detection):");
                BufferedReader br = new BufferedReader(new
                   InputStreamReader(System.in));
                int id = Integer.parseInt(br.readLine());
                mac = new SlefMAC(id,this, 1000, 1000, 10);
            } catch (IOException e) {
                /* an error happened opening the device, give up */
                System.out.println("Error opening the device");
                System.exit(-1);
            }
        }
    }

    /* Implements the PacketReceiver interface, called everytime a
       packet is recevied */
    public void receivePacket(Packet packet) {
        /* If the packet is an ethernet packet it is possible to
           look at the source address */
        if (packet.datalink instanceof EthernetPacket) {
            EthernetPacket ethernetPacket = (EthernetPacket) packet
               .datalink;
            System.out.println("Received packet from :" +
               ethernetPacket.src_mac);
        }
        System.out.println("Packet contains following data: " +
           packet.data);
    }

}
```

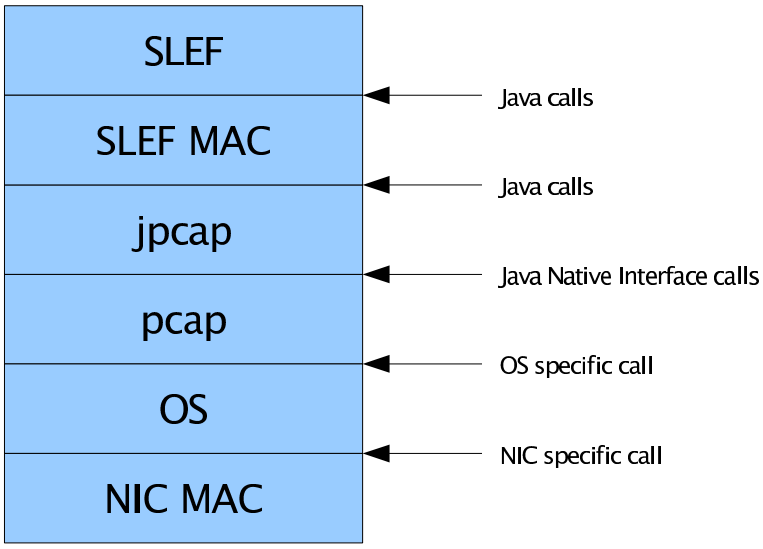

Figure 1: Stack used by the application

```
#!/bin/bash

for dev in ‘grep : /proc/net/wireless | sed ’s/:.*//’‘
do
    # set the RTS threshold
    iwconfig $dev rts 0
    # set the rate to the fixed rate of 11 Mbps
    iwconfig $dev rate 11M fixed
done
```

Listing 1: Initialization script for linux

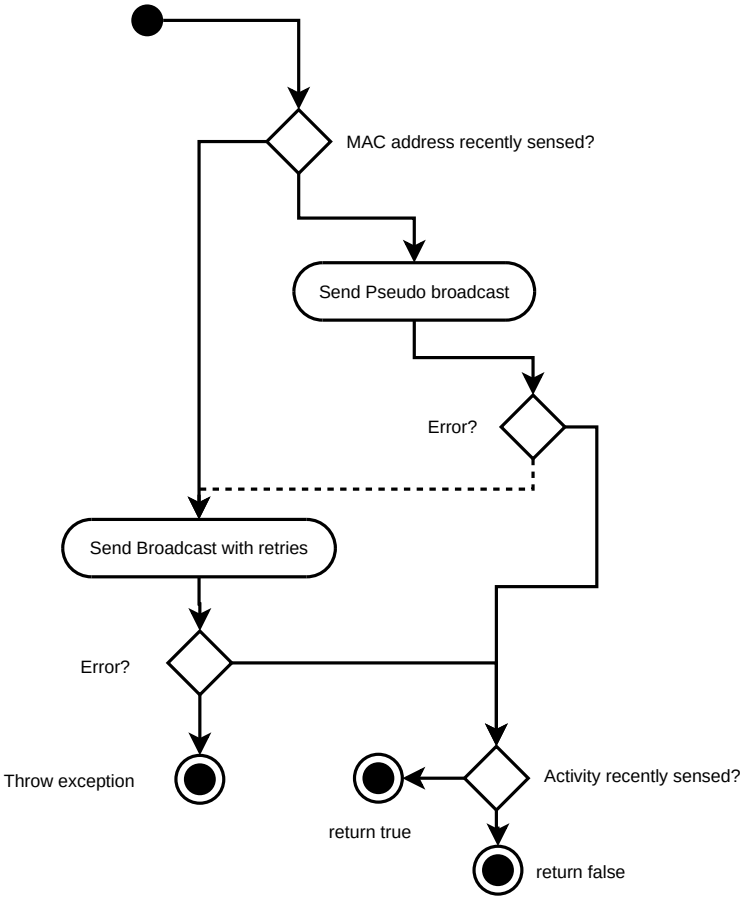

Figure 2: Algorithm of the send function

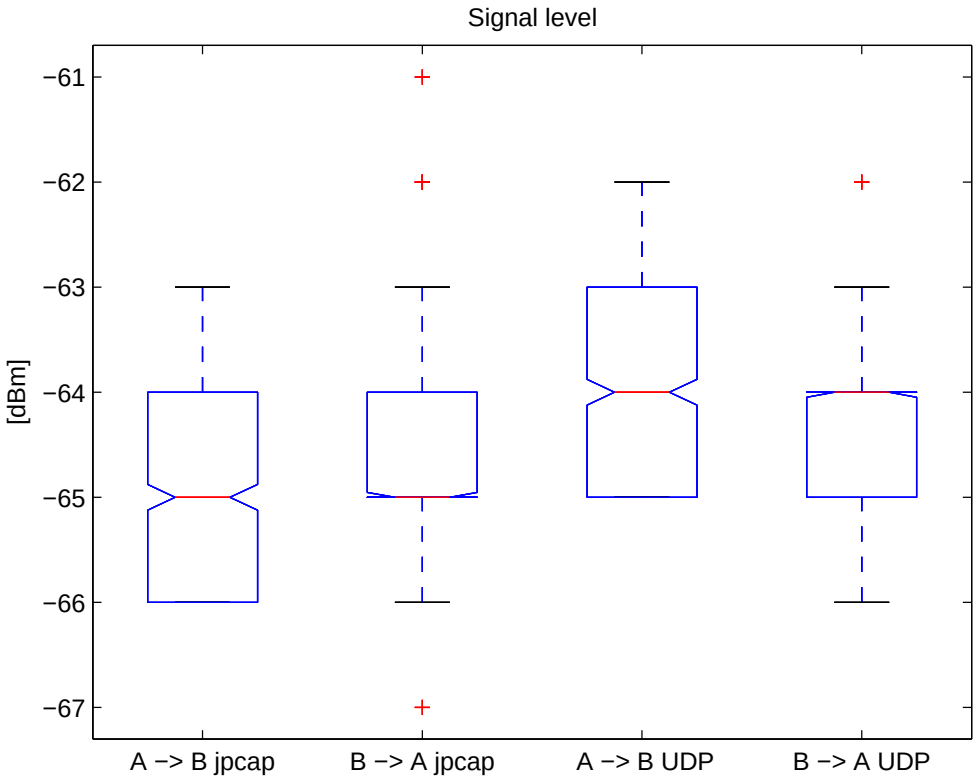

Figure 3: Signal level during the tests

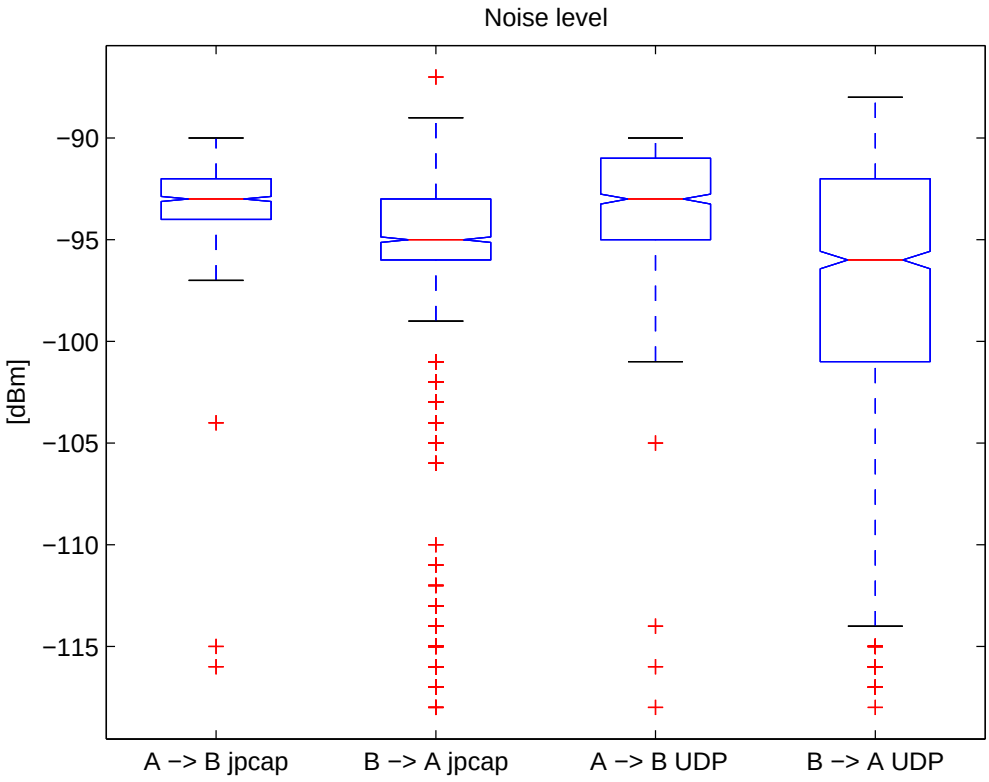

Figure 4: Noise level during tests

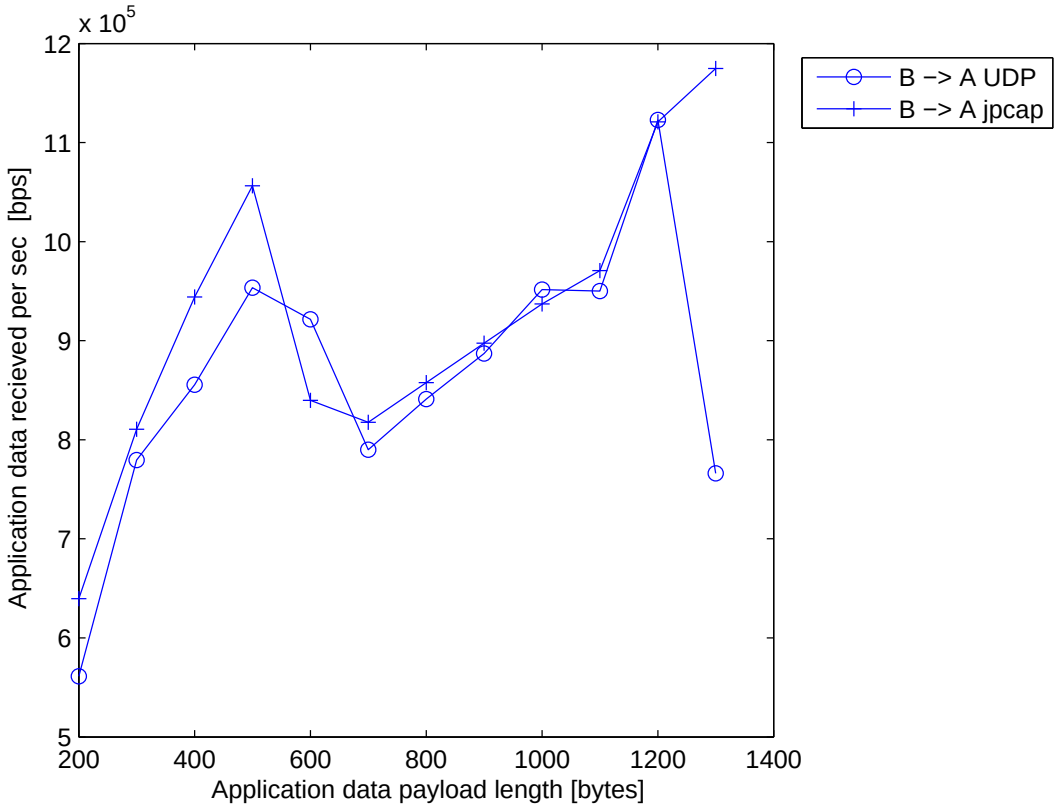

Figure 5: Performance results from Windows to Linux

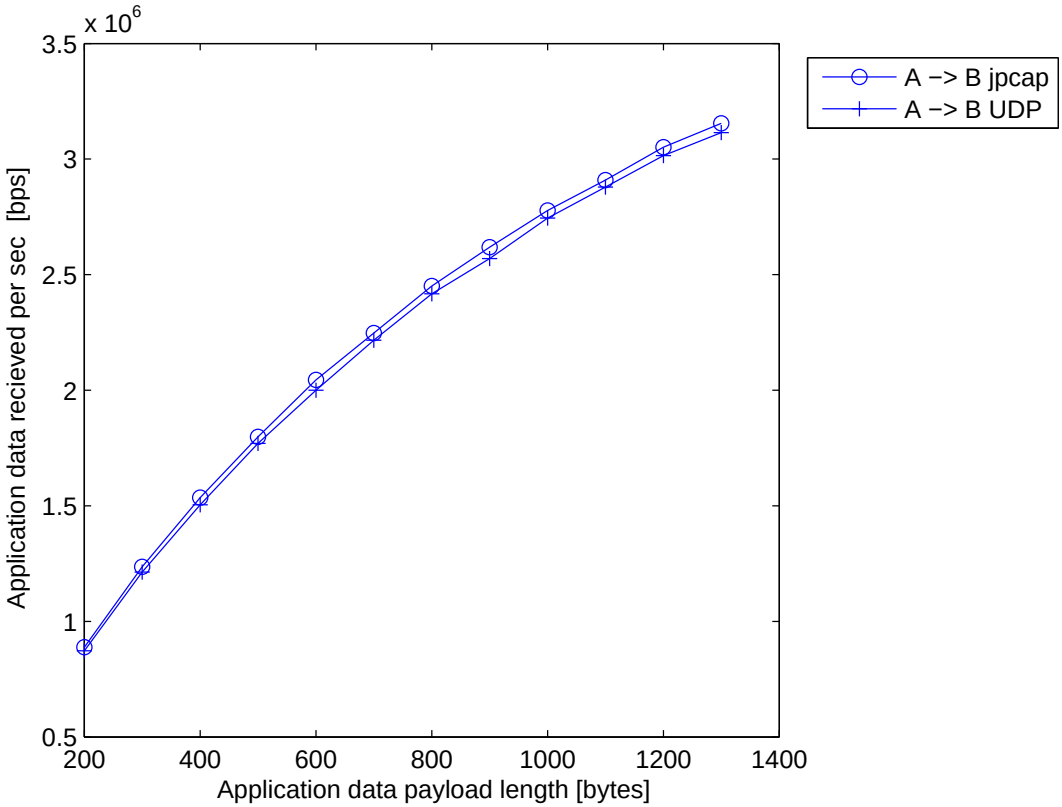

Figure 6: Performance results from Linux to Windows