Advanced Networking Lab

Thomas Hendriks Jakob Struye

March 2, 2017

Lab 3

Performance Measurements

In this lab, the performance and throughput in wireless networks will be investigated. Using tools like `iperf`, we will record the maximum throughput which can be achieved in wireless networks and have a look at the parameters influencing this throughput.

3.1 Bit Rates

Exercise 1: Basic throughput in IEEE 802.11a

This first exercise will give you an insight into the difference in usable throughput and the available bit rate. To determine the throughput, we will be using a tool called `iperf`. This is a client-server based tool which sends **tcp!** (**tcp!**) or **udp!** (**udp!**) traffic and reports the measured throughput. Consult the man pages for the details about this tool. A second tool to be used is `gnuplot`, in order to plot your results in a graph. More info about `gnuplot` can be found in [**?**] and a nice tutorial in [**?**]. A basic gnuplot script to generate your first plots is provided on the course website. The `iperf` tool is preinstalled on the wireless nodes, while `gnuplot` is available on the lab PCs.

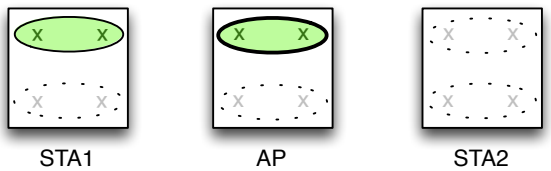


Figure 3.1: Basic throughput setup.

1. Start by configuring the setup as shown in figure **??**. Use `fc00:gid!::3/64` for the **ap!** (**ap!**) and `fc00:gid!::1/64` for **sta!** (**sta!**)1.

2. IEEE 802.11 supports various bit rates. Using `iw list` you can query the supported rates:
   Which rates are supported in IEEE 802.11a?
   L3-1-1

```
5.0GHz range:
    Bitrates (non-HT):
      * 6.0 Mbps
      * 9.0 Mbps
      * 12.0 Mbps
      * 18.0 Mbps
      * 24.0 Mbps
      * 36.0 Mbps
      * 48.0 Mbps
      * 54.0 Mbps
```

   Which rates are supported in IEEE 802.11b/g?
   L3-1-2

```
2.4GHz range:
    Bitrates (non-HT):
      * 1.0 Mbps
      * 2.0 Mbps (short preamble supported)
      * 5.5 Mbps (short preamble supported)
      * 11.0 Mbps (short preamble supported)
      * 6.0 Mbps
      * 9.0 Mbps
      * 12.0 Mbps
      * 18.0 Mbps
      * 24.0 Mbps
      * 36.0 Mbps
      * 48.0 Mbps
      * 54.0 Mbps
```

3. `iperf` can be used to measure the maximum throughput between two stations. Therefore, a server is started on one end and a client connects on the other end to

this server. A connection is set up and as **tcp!** tries to maximize the throughput on a connection, an estimate of the maximum throughput on a link can be calculated. When using `iperf` with the basic parameters, it will perform a 10 seconds test and report the achieved throughput at the client side.

4. Start a basic `iperf` session between the **sta!** and **ap!**, with the `iperf` server on the **ap!**:
   `AP:~# iperf -V -s`
   `STA1:~# iperf -V -c fc00:gid!::3`
   Copy the output of the `iperf` client:
   L3-1-3

```
STA1:~# iperf -V -c fc00:1::3
------------------------------------------------------------
Client connecting to fc00:1::3, TCP port 5001
TCP window size: 20.8 KByte (default)
------------------------------------------------------------
[  3] local fc00:1::1 port 47969 connected with fc00:1::3 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec  22.8 MBytes  19.0 Mbits/sec
```

5. Now, using `iperf`, collect the throughput for each available rate and write the results in a file `/mnt/L3-1-4.tcp.txt` On each line, first put the rate followed by a space and then the result in Mbit/s obtained from `iperf`, e.g. `54 30` denotes that a 30Mbit/s was measured when using a rate of 54 Mbps. You can change the rate used at the **sta!** using `iw`, e.g. to 54Mbps, as follows:
   `STA1:~# iw dev wlan0 set bitrates legacy-5 54`
   You can check the actual used bit rate from the output of `iwconfig`:
   `STA1:~# iwconfig wlan0`

   As we will be repeating the collection of these results in the following exercises, it will be easier to use some bash scripting to speed up this process. A helper script can be found in the file `iperf-tcp.sh`. Change this script so it loops over the correct bitrates, and use it with the command `./iperf-tcp.sh fc00:grID::3`. This script generates output that can be directly copied to `/mnt/L3-1-4.tcp.txt`

6. `iperf` will by default use **tcp!** to check the connection, but it is also possible to use a unidirectional **udp!** stream. Therefore, one can try to feed more data to the network than the network can support and as such measure how much can be actually delivered. Thus, repeat the previous scenario and collect the maximum achievable throughput using `iperf` in **udp!** mode for each available bit rate. Save these

results in the same format as in the previous item in a file `/mnt/L3-1-4.udp.txt` Again, a script called `udp.sh` is provided that automates this process. The script will try to send a UDP stream that fills the wireless link.

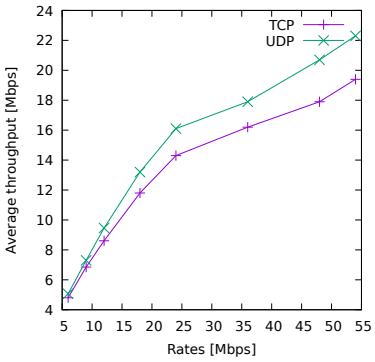
```
AP:~# iperf -V -s -u -l 1452
STA1:~# ./iperf-udp fc00:grID::3
```

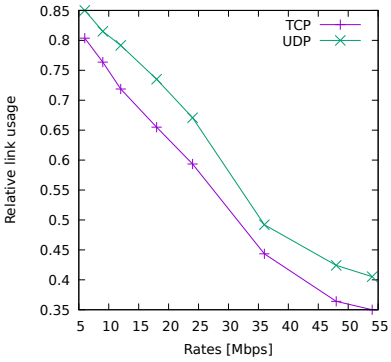
`-l` is the letter l, not the number one!

7. Now plot these results using `gnuplot`. On the course website, a `gnuplot` script, `tput1.gnuplot`, is provided to plot the throughput and the relative link usage over the various available rates. The used commands are straightforward and the script is inline commented so should be self-explanatory. Make sure you understand the various commands in the file. The script produces two PDF files which can be viewed with any regular PDF viewer. Place the `.txt` files you created in the previous steps in the same directory as the `gnuplot` script. Save the generated files in your lab report as `L3-1-4-tput.pdf` and `L3-1-4-usage.pdf`. Add the plots here and shortly discuss what can be observed. Also shortly discuss the difference between **udp!** and **tcp!**. To generate the PDF files, use:

```
gnuplot tput.gnuplot
```

L3-1-4





UDP's throughput is consistenly higher than TCP's. TCP has larger headers so more overhead. More importantly, TCP's congestion control mechanisms lead to reduced throughput. TCP will occasionally not send as fast as the network could handle (mainly during slow start), while UDP is constantly transmitting as fast as the wnic will allow it to. Throughput grows together with bitrate: the network supports sending more bits per second with higher bitrates.

The throughput does not grow as quickly as the bitrate: with higher bitrates the bits are sent faster and bits are more likely to get lost. For TCP this triggers retransmissions and congestion control. For UDP the lost data is simply not counted towards the throughput. Both have a negative effect on throughput. This means doubling the bitrate does not double the observed throughput. This translates to a

decreasing usage with increasing bitrate.

Exercise 2: Client to client throughput

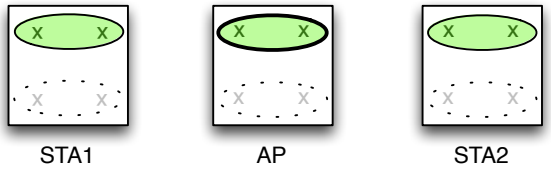
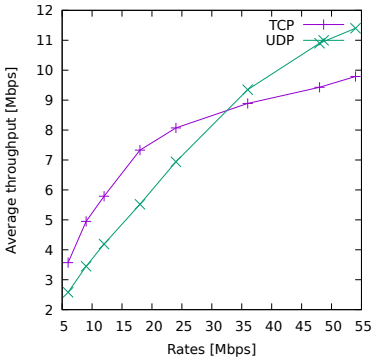


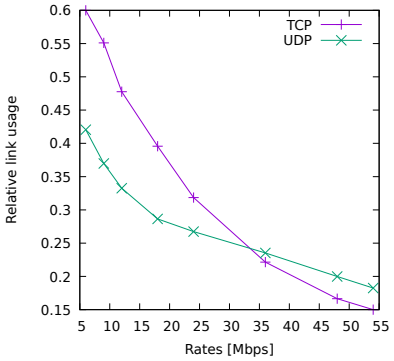
Figure 3.2: Client to client throughput setup.

In the previous exercise, the throughput was measured between a **sta!** and the **ap!** it was connected to. In this exercise, we are interested in the throughput between two **sta!**s associated with the same **ap!**.

1. Create the test setup as shown in figure **??**. Configure **sta!**2 with IP address `fc00:grID::2/64`.
2. Collect the **tcp!** and **udp!** throughput measures as in the previous exercise and save them to `/mnt/L3-2-1.tcp.txt` and `/mnt/L3-2-1.udp.txt`
3. Adapt the provided `gnuplot` script and create graphs for the new measurements. Include them in your report.
4. Comment on the obtained results and compare the results from this exercise with those from the previous exercise. Explain the difference.

L3-2-1





For low bitrates TCP performs better than UDP. Due to TCP's congestion control

there are fewer collisions compared to UDP. With lower bitrates the attainable maximum throughput is lower. This means TCP's slow start will reach it faster. The shorter the slow start phase, the smaller its negative effect on throughput can be. For lower bitrates this makes TCP perform better than UDP. For higher bitrates the influence of TCP's additional overhead and negative effect of congestion control become larger and UDP is more performant than TCP again.

For both TCP and UDP, throughput in this case is roughly half of the previous test. Each packet is transmitted twice (to the AP and from the AP), halving the total throughput between source and destination.

The effect of increasing bitrate on the usage is similar to the previous test.

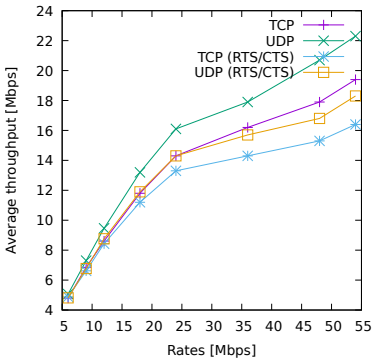
3.2 Network Settings

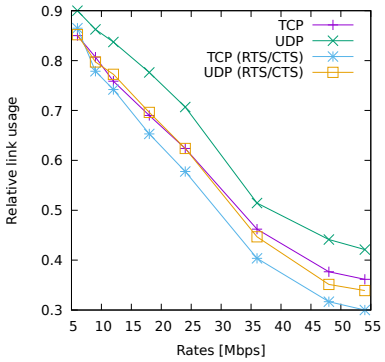
Exercise 3: **rts!**/**cts!**

In lab 2, we enabled the **rts!** (**rts!**)/**cts!** (**cts!**) mechanism to show that in IEEE 802.11, stations can reserve the medium in order to avoid collisions. In this exercise, we will study the effect of **rts!**/**cts!** on throughput.

1. Enable **rts!**/**cts!** on all devices.
   ```
   :~# iw phy phy0 set rts 1000
   ```

2. Repeat the iperf tests from **sta!**1 to **ap!**. Save your logs to `/mnt/L3-3-1.tcp.txt` and `/mnt/L3-3-1.udp.txt` Again modify the gnuplot script to create new graphs and include them in your report. To make the comparison more easy, modify the gnuplot script so that you plot both the results from exercise **??** and this exercise on the same graph. What do you observe? Why?
   L3-3-1





For both TCP and UDP throughput is consistently lower with RTS/CTS enabled. Looking at the full iperf output, we noticed UDP packet loss was, surprisingly, consistently higher with RTS/CTS enabled. Furthermore the RTS/CTS mechanism adds overhead, leading to lower throughput.

Some additional research indicated that packets dropped before they are ever sent, are also counted towards packet loss. By analyzing the number of actually sent packets for UDP with tcpdump, we noticed all 'packet loss' was due to packets being dropped at the sender: every packet that was actually sent also arrived. Any time spent sending RTS/CTS packets could be considered 'wasted' in this case: there is no packet loss for it to prevent, and the time could have been spent sending additional data. Assuming there was no packet loss after transmission for TCP either, this explanation holds for TCP as well.

Even with RTS/CTS enabled UDP still outperforms TCP: TCP's lower performance is caused by its additional overhead and congestion control, which is not changed by having RTS/CTS enabled.

Exercise 4: Frame length

The standard **mtu!** (**mtu!**) for ethernet frames is 1500 bytes. IEEE 802.11 however allows an **mtu!** up to 2274 bytes. In this exercise, you will measure the effect of frame size on throughput.

1. Continue from the previous setup, but make sure **rts!**/**cts!** is turned off on all nodes:
   `iw phy phy0 set rts off`

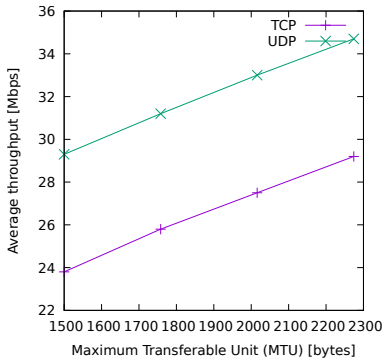
2. On **sta!**1, reset rate control to its default (automatic) value:
   `sta!1:~# iw dev wlan0 set bitrates`

3. Perform the following tests with 4 different maximum frame sizes: 1500, 1758, 2016 and 2274. The frame size can be controlled by changing the **mtu!**. *If you do this, do this on all nodes!*
   `ifconfig wlan0 mtu 1758`

4. For each of the **mtu!**s mentioned, start with a **tcp!** `iperf` test between **sta!** and **ap!** as in the previous exercises and make plots for each frame size. Generate a results file called `/mnt/L3-4-1.tcp.txt` containing **mtu!** and bit rate achieved. The file should have the same format as the ones produced this far, except that the first column now contains your **mtu!** setting rather than the link speed.
   `sta!1:~# iperf -V -c fc00:grID::3`

5. Repeat the tests, now using **udp!**. For `iperf`, use the `-l` parameter, followed by the current MTU setting minus 48 (e.g. 1452 if the **mtu!** is set to 1500) to generate packets that are large enough to fill the MTU. ***This must be done on both client and server!*** Save your results to `/mnt/L3-4-1.udp.txt`.
   `sta!1:~# iperf -V -u -l 1452 -c fc00:grID::3 -b 54M`

6. Modify the `gnuplot` script to generate the same graph as before. Plotting only the throughput versus **mtu!** suffices. A relative link usage graph is not required. What do you observe? Why? Be as precise as possible.
   L3-4-1



Increasing the MTU and thus the size of the packets leads to increasing throughput for both UDP and TCP. Every bump in MTU leads to a throughput increase of 1.7 to 2 Mbps. This can partially be explained by the fact that Iperf reports the throughput looking only at the application data. Header sizes are 32bytes for 802.11, 40bytes for IPv6, 8bytes for UDP and 20bytes for TCP, leading to a per-packet overhead of 80 and 92 bytes per packet for UDP and TCP respectively. The larger the packet size, the lower the relative overhead of the headers. In a

sense, iperf subtracts the header overhead from the actual packet throughput by only reporting the application data throughput.

This alone is not enough to explain the increase in throughput though. Let's consider UDP for the first two MTUs. At 1452 bytes per packet, the headers are 5.51% of the application data in size. At 1710 bytes per packet this drops to 4.68%. If we consider the throughput regarding the entire packet, the throughputs rise to 30.9 and 32.7 Mbps respectively. The relative gap between them is only slightly smaller than before, and thus still largely unexplained. Another interesting observation is that for UDP each increase in MTU decreases the reported packet loss by roughly 4%. By looking at a trace of the tests, it is clear that (almost) all packet loss occurs before the packets are actually sent, as opposed to in the air or at the receiver. This seems to indicate the packets are dropped deliberately at send queues. A detailed explanation of why the percentage of dropped packets is so significantly different depending on the MTU would require thorough knowledge of how the queues work (both at TCP/UDP and wnic level). Some quick Googling indicated more people have noticed this effect, but nobody was able to offer a detailed explanation for why it occurs.

(While all other data was gathered during the same session, this exercise was performed at another time. This explains why throughputs in this graph are higher than in the others).

3.3 Packet Size

Exercise 5: IP Fragmenting

Changing frame lengths has an effect on the throughput, as you have shown in the previous exercise. However, in that case, traffic was flowing between segments with the same fragment size. When we consider the default setup of an **ap!** which is connected to the Internet, the **wan!** (**wan!**) link is most likely limited to 1500 bytes per frame and thus IP fragmenting comes into play. In IPv6, fragmenting in intermediate hops is not allowed. The end hosts may, however, fragment the IP packets end-to-end to make sure they fit on the link with the smallest **mtu!**. Depending on the original payload length, fragments of various lengths will be created. In this exercise we will have a look at the impact of fragmenting on the overall throughput.

To do so, we will run the `iperf` server on the third node, which is only connected by wire to the **ap!**. Using additional routes, we will create a setup where **sta!**1 and **sta!**2 communicate via the **ap!**. Traffic between **sta!**1 and the **ap!** will be transmitted on the

wireless link, while traffic between the **ap!** and **sta!**2 will be transmitted on the wired link.

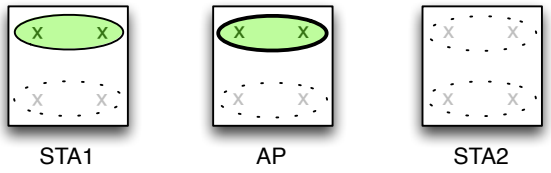


Figure 3.3: Fragmenting setup.

1. Start by configuring your setup as shown in figure **??**. **sta!**2 will only be used as `iperf` server and does not need an active **wnic!** (**wnic!**).
2. Set `fc00:gid!::3/64` as IP for the **ap!** and `fc00:gid!::1/64` for **sta!**1.
3. Traffic from **sta!**1 to **sta!**2 will by default use the wired network. To force a wireless hop, we will add some new routes:
   ```
   sta!2:~# ip route add fc00:gid!::/64 via <wired IP address of ap!>
   sta!1:~# ip route add <wired IP address of sta!2>/128 via fc00:gid!::3
   ```
4. IPv6 forwarding is by default not enabled on the lab nodes. We must enable it on the **ap!** for the setup to work. However, due to a bug, the **ap!** will clear its default route when we do this, making it instantly inaccessible over IPv6. Perform the following as a workaround:
   ```
   ap!:~# ip -6 route show | grep default
   ```
   You should get output like this:
   ```
   default via fe80::225:90ff:fe61:efac dev eth0 proto kernel metric 1024
   expires 1670sec
   ```
5. Enable IPv6 routing functionality on the **ap!**:
   ```
   ap!:~# sysctl -w net.ipv6.conf.all.forwarding=1 && ip route add default
   via fe80::225:90ff:fe61:efac dev eth0
   ```

   The IP address in that command should be the same as that in the output of the previous step.
6. Perform a `traceroute6` from **sta!**1 to **sta!**2 to check if the routes are set up correctly. Include your traceroute6 output below:
   ```
   sta!1:~# traceroute6 <wired IP address of sta!2>
   ```

L3-5-1

```
STA1:~# traceroute6 2001:6a8:500:e081:20d:b9ff:fe18:235c
traceroute to 2001:6a8:500:e081:20d:b9ff:fe18:235c (2001:6a8:500:e081
   :20d:b9ff:fe18:235c) from fc00:1::1, 30 hops max, 16 byte packets
 1  fc00:1::3 (fc00:1::3)  1.965 ms  0.984 ms  0.909 ms
 2  wmn3.mosaic.uantwerpen.be (2001:6a8:500:e081:20d:b9ff:fe18:235c)
    1.455 ms  1.027 ms  0.986 ms
```

The MAC addresses show the wireless interface's MAC address of AP and the wired interface's MAC address of STA2.

7. Now, for each **mtu!** (1500, 1758, 2016 and 2274), perform the following:

   - Set the **mtu!** on the wireless link.
   - Perform a TCP `iperf` from **sta!**1 to **sta!**2.
   - Perform a UDP `iperf` from **sta!**1 to **sta!**2. In each trace, set your frame size to the **mtu!** of the wireless link minus 48. Send at a bit rate of 54 Mbps.

8. The results of your tests should once again be saved to text files like in the previous exercise. Save them to `/mnt/L3-5-2.tcp.txt` and `/mnt/L3-5-2.udp.txt`. Create a throughput graph as in the previous exercise.

9. You should also make `.pcap` trace files from these experiments. However, as the link gets fully saturated, the trace file would become very large. Therefore, we will repeat the experiments sending far less traffic. This of course means that you will not see a difference in `iperf` results. The traces will help you, however, to understand what has happened in the previous exercise.

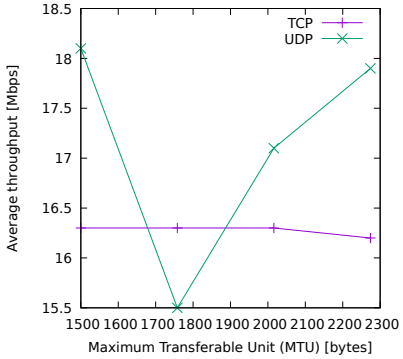
10. In order to limit the amount of traffic, we will do the following on **sta!**1:

    - Set the rate of `wlan0` to 6 Mbps.
    - for both TCP and UDP `iperf`, add the `-t 2` parameter. This limits the `iperf` trace to two seconds.
    - for UDP `iperf`, omit the `-b 54M` parameter. This way, `iperf` will only transmit at 1 Mbps.

11. Now, do the same `iperf` tests again, using the above parameters for `iperf`. For each run, make a `tcpdump` trace on interface `wlan0` of the **ap!**. Save the traces to `/mnt/L3-5-2.tcp.<MTU size>.pcap` and `/mnt/L3-5-2.udp.<MTU size>.pcap`, respectively.

12. What do you observe? Distinguish between the behaviour for TCP and UDP. Explain your findings by including your graphs and refering to the trace files you made. Be as precise as possible.

L3-5-2



UDP shows a very clear drop in throughput when increasing the frame length over 1452 bytes. While the first trace (/traces/L3-5-2.udp.1500.pcap) just shows regular UDP datagrams, the following three dumps show IPv6 fragmentation. For example in traces/L3.5.2.udp.1758.pcap, the iperf client first tries to send a UDP packet of length 1772 (packet 1). An ICMPv6 message indicating this packet exceeded the MTU is sent back (packet 2). Each following frame containing 1710 bytes of application data is now divided across two packets (e.g. packet 3+4) each respecting the Path MTU. First an IPv6 fragment is filled with as much data as the path MTU will allow. The remaining data, along with the UDP header is sent in a second packet. This means each 1710 byte frame now requires two sets of layer 2 and 3 headers, significantly increasing the overhead (relative to the amount of application data) and explaining the drop in throughput. We can calculate the relative overhead (assuming Ethernet at layer 2). The 1458 byte frames had 14 (ethernet) + 40 (IPv6) + 8 (UDP) = 62 bytes of headers, meaning 4.1% of transmitted data was overhead. For the 1710 byte frames this becomes 2 * 14 (ethernet) + 2 * 48 (IPv6 with fragmentation header) + 8 (UDP) = 132 bytes of headers, leading to 7.2% overhead.

For further increases in frame length, the throughput starts to rise again. Each 1710 byte frame was divded into parts of 1448 and 262 bytes and each sent in a separate packet. Each packet could contain up to 1452 bytes with this path MTU. For the 1968 and 2226 frame lengths, each frame is still fragmented into only two parts, but the second fragment is bigger (respectively 520 and 778 bytes of application data). By sending fuller packets, the relative overhead decreases and the throughput increases.

For TCP the throughput remains stable while increasing the wireless link's MTU. In

UDP's case the fragmentation was triggered by also increasing the frame length. For TCP no such option exists. Even if we set the Maximum Segment Size (MSS) to something exceeding the Path MTU, the kernel will first discover the Path MTU and lower the segment size to not exceed the Path MTU. In each of the traces the data is sent to the iperf server in packets of 1514 bytes. Having an MTU higher than the actual packet size somewhere in the data path makes no difference, there is no real difference between the four test cases with TCP.

Acronyms

AP access point

CTS Clear to Send

grID group ID

MTU Maximum Transferable Unit

RTS Request to Send

STA station

TCP Transmission Control Protocol

UDP User Datagram Protocol

WAN wide area network

wnic wireless **nic!**