WiFi project report

T. Moerland

September 18, 2003

1 Project Overview

The goal of this project is to design and implement a WiFi, or 802.11, Management Information Base (MIB). The implementation should run on the nodes of the Wireless Leiden network. It should be available as stand-alone SNMP agent, and also be integrated with SPLASH.

In this project the following items have been addressed.

- Testbed setup
- Design of 802.11 MIB
- Implementation of MIB in SNMP agent
- SPLASH
- Example SNAP packets

These issues are considered in the next sections.

2 Testbed setup

For the project development a suitable test environment is necessary. Since the software should run on nodes of the Wireless Leiden network, the test nodes should be similar. This means the machines should run on the FreeBSD 5 platform and have multiple interfaces that can be either Access Points (AP) or clients. Ad-hoc mode is not used in WL.

Different testbed configurations have been used at different development stages.

1. **Single machine, single NIC**. An Avaya wireless NIC (the standard LIACS NIC) is used to experiment with communication between userland and the driver. This includes getting and setting of variables like channel, SSID, WEP keys.

2. **Two machines, one client and one AP**. This setup is used for experimenting with (storing statistics of) the communication between Access Points and clients. Since only Prism chipsets are capable of operating in AP mode, a card with this chipset is required (Avaya has a Wavelan chipset).

3. **One machine, two NICs**. Used to test the modified network driver and the implemented 802.11 MIB when multiple NICs are present.

After these stages, the software is ready to be tested on the Wireless Leiden network.

3 Design

3.1 MIB

The goal of manageable wireless network interfaces via SNMP can be achieved in two ways. The most straighforward way is perhaps to implement an existing MIB. Specifically, the IEEE 802.11 standard comes with the IEEE802dot11-MIB ([1]). This MIB certainly has variables useful for our purpose, however a large number is not only hard to implement but also not very relevant in our case. We could make a compliance statement, indicating which specific parts of the MIB has been implemented.

Unfortunately, the MIB has no support at all for the HostAP mode which is vital to the Wireless Leiden network. Moreover, the variables that do seem useful, are sometimes outdated (e.g. a WEP key size of 5 bytes, thus only supporting 64 bit encryption) or defined a bit different than we need them to be. Also, there are no variables for the statistical information we would like to have. There exist lots of other wireless network related MIBs, see for example [2]. These have similar problems as the IEEE802dot11-MIB.

The second option is to design our own MIB. When placed on the enterprise subtree, we can choose any variables and definitions we need, thus allowing for Wireless Leiden specific design. The implementation of such a MIB is somewhat more straightforward, since all the variables share the same OID prefix.

Taking the second approach, the MIB is designed to provide information about wireless network interface cards. This information includes most variables that can be configured by using the `ifconfig` and `wicontrol` commands, e.g. SSID, channel, port type (client, ad-hoc, HostAP). Also WEP key values can be managed. The other part of the available information is related to client-AP links (both interlinks and normal user client links). This includes bandwidth per station, signal quality per station, association and disassociation status and timestamps. Refer to the light80211MIB for all the details. The MIB is succesfully parsed with maximum warning level by `smilint` [5], and has been carefully reviewed.

3.2 SNMP agent extension

The implementation is based on `net-snmp-5.0.6`, as this is the latest release of `net-snmp` and SPLASH is based on it. This ensures easier integration with SPLASH.

There are several ways to implement an extension to a `net-snmp` agent, most notably writing a subagent with the `AgentX` mechanism and statically extending the agent. Since `AgentX` is still beta version, static extension was considered more reliable.

4 Implementation

The source files for this project have been documented quite extensively. Comments have a Doxygen [3] readable format, view the result here [4]. In this section some of the concepts behind this code and issues involved are discussed. Especially the extensions to the wireless driver are discussed.

Note that at some places descriptions might get a bit detailed. Those parts can be considered a tutorial for developers planning to extend or modify the source.

4.1 Wireless Driver

The driver is extended to provide additional, mostly statistical, information about other stations (i.e. remote interfaces). In the next subsections various issues are discussed.

4.1.1 Memory structures

The original wi driver has the option (by compiling with -DWICACHE) to keep a little cache of the last packets received from MAC addresses. The structure of this cache, as defined in `if_wavelan_ieee.h`, contains an entry for the MAC address, the IP address and entries for the signal, noise and quality of the last packet received from that address. For our purpose, we needed to extend this cache with other information like averages of and deviations in signal, noise and quality, (dis-) association time and the like.

For compatibility with programs depending on the original cache, it was decided to create a new extended cache structure, defined as follows:

```
struct wi_sigextcache {
        char    macsrc[6];
        u_long  rcvpkts;         /* total number of packets received from this MAC */
        u_long  rcvbytes;        /* total number of bytes received from this MAC */
        u_long  sndpkts; /* total number of packets sent to this MAC */
        u_long  sndbytes; /* total number of bytes sent to this MAC */
        float   EWMAvg[3];       /* Signal, Noise, Quality Averages
                                        these are Exponential Weighted Moving Avg */
        float   deviation[3];    /* corresponding deviations */
```

```
        struct timeval assoctime;  /* timestamp of last association */
        struct timeval disassoctime;  /* timestamp of last disassocation */
        float   EWMAOutBandwidth; /* EWMA of the bandwidth (direction out) */
        float   devOutBandwidth; /* deviation of the bandwidth (direction out) */
        float   EWMAInBandwidth; /* EWMA of the bandwidth (direction in) */
        float   devInBandwidth; /* deviation of the bandwidth (direction in) */
        u_long  tempsndbytes;    /* used for bandwidth calculation*/
        u_long  temprcvbytes;    /* contains number of bytes in the last sec*/
        u_int16_t remove_ctr;    /* used to remove this entry from the cache*/
        u_int16_t nr_assocs;     /* the number of associations of the station */
};
```

Note that all code regarding this extended cache can be included at compile time by defining WICACHE (option -DWICACHE), and can thus also be easily removed.

Similar to the original cache, the extended cache is kept in the structure `wi_softc` associated with each wireless interface. It is stored as a fixed-size array. The maximum size of these arrays was limited by the maximum number of bytes that can be transferred in a single ioctl call. Currently, this is 1024 (512 words). Of course this is just another defined value, but it is unwise to simply increase it. However, by using multiple ioctl calls to get the whole array, the cache array can be of arbitrary size. Equal indexes of the original and the extended cache correspond to equal MAC addresses. The size of the original cache is limited to about 20 entries because only a single ioctl call can be used for compatability. Entries above this limit are ignored.

4.1.2 Adding and removing entries

To understand which stations can be found in the cache, we will discuss adding and removing of entries. A difference must be made between interfaces in HostAP and BSS mode.

HostAP mode. Entries are added to the cache when client stations associate with the interface. An association request is sent by the client, and processed by the driver (in `wi_hostap.c`). If all goes well (correct authentication mode, WEP key, etc), the client is associated and is added to the cache. When a station disassociates (sends a disassociation request) or is disassociated by the AP (e.g. after inactivity timeout), the `remove_ctr` in the cache entry is set to a specific value (`timeout_time`: interface dependent, located in `wi_softc`; SNMP settable). Every second this value is decreased and when one is reached, the entry is removed. Of course, if the client re-associates before this occurs, the `remove_ctr` is set to zero and there is no danger of deletion.

Note that the HostAP interface has an inactivity timeout, which is activated after association or last packet received. It is set to 120 seconds, after which the client is disassociated. Another 120 seconds later, it is also deauthenticated and deleted from memory. If the client is active, but was just not sending any packets, it will reassociate immediately.

BSS mode. Entries are added to the cache when packets are received from stations (for example as repeated by the AP the interface is associated with). Also, an entry for the AP is added on association. Since there is no event in the wi driver on assocation with AP when in BSS mode, we check every second whether the current BSSID has changed. The current BSSID is the MAC address of the AP that the client is associated with. Special BSSID 44:44:44:44:44:44 is used to indicate that there is no association, so if this changes to a valid MAC address we know we have associated. Non-AP entries are removed in `wi_softc->timeout_time` (the same SNMP settable value again) seconds after the last packet has been received. The AP entry is only removed the same number of seconds after disassociation. Again, we don't get an event when this occurs but check every second what happened to the current BSSID. If it changed to the unassociated value (all 0x44's) we have disassociation. Note that if the current BSSID changed to a different value, the client 'hopped' to a different AP.

There are some known problems with this mechanism in client-AP interlink setup, refer to section 4.1.4.

Note that in both modes 'gone' stations are removed eventually from the cache.

4.1.3 Variables

- **Signal information**. For every packet received, the interface delivers information about signal strength, noise level and link quality. The link quality is in de wi driver simply calculated as the signal strength minus noise level. Of these three values averages and deviations are kept. After every received packet these averages and deviations are updated according to the Exponential Weighted Moving Average method. The following two formulas are used (read as assignment statements):

  $$\begin{aligned} avg &= (1-c) * avg + c * val \\ dev &= (1-c) * dev + c * |val - dev| \end{aligned}$$

  where val is the value for signal strength, noise level or link quality and c is a constant (settable by SNMP). Since c is settable, a management application can set how fast previous values are 'forgotten' (exponentially die away).

  Note that all incoming packets are considered, no differentiation occurs between packets from different layers. Specifically, no difference is made between IP packets and 802.11 management packets. Also, not all management packets are visible to the driver, e.g. beacon frames.

- **Bandwidth information**. Bandwidth is calculated in both directions, here 'Out' and 'Sent' means from interface to station in cache, and 'In' and 'Received' means vice versa. We calculate Exponential Weighted Moving

Average and deviations (as explained under Signal information). We use our function-called-once-every-second again. During the interval between different calls to this function, the number of bytes sent and received are summed up. When the function is called, the values are used to calculate the average bandwidth.

The number of bytes received per packet is easily read in the `rxeof` function, which is called after a packet has been received. The number of bytes sent however, must be calculated in the `write_data` function. Since this function is called twice for every packet sent, we need to be careful. Also, 802.11 header length is variable (depending for example on WEP usage and type of packet). See implementation for details.

- **(Dis-) association information**. Section 4.1.2 deals with when and how association is detected and registered. Whenever such an event is detected, the current time is stored (this can later be converted to a time interval, indiciting the amount of time that has passed since the event). In the case of an association, the counters for packets and bytes in and out, which are made readable by the SNMP deamon are reset to zero. Also the number of associations is incremented. All other variables remain unchanged, this is also true for the averages.

  On a disassociation, only the current time is saved to that variable. No variables are reset, and the calculation of averages continues. This means the signal/noise/quality averages will remain unchanged, and the bandwidth averages will drop to zero, since no new packets arrive. As explained before, the cache remove counter is activated to ensure eventual deletion of the entry from cache.

  The accuracy of the BSS mode disassocation detection is one second, while the AP has an accuracy of hundredths of seconds. This leads to differences in corresponding AP and client timestamps of at most one second.

4.1.4 Interlinks

The adding and removing of entries, the setting of variables, they have been specifically designed with the Wireless Leiden interlinks in mind. This is why previous sections are rather detailed.

In these point-to-point antenna setup, one interface is in HostAP mode and the other in BSS mode. It is evident that certain information from about the link must be equal on both sides. A lot of effort is put into making sure this holds. Unfortunately, in some special circumstances this might not be the case.

First of all, as discussed before, the accuracy of the BSS mode (dis-) association timestamps is one second, while the AP has a much higher accuracy. This automatically leads to differences between corresponding values of at most one second.

Furthermore, if communication between client and AP is physically blocked, the client notices the beacon loss. It gets disassociated and registers this as described above. The AP itself on the other hand, does not notice this disassociation because it doesn't receive a disassociation request. It waits for it's inactivity timeout to kick in, after which it notices there's no response from the client (a reassociation request). Thus, the disassociation timestamp for the HostAP will be 120 seconds (the inactivity timeout) late!

Now one might expect problems when the client associates again with the AP, before those 120 seconds have passed, since the AP will still think the client is actually associated. It might not register this reassociation and thus not reset the counters or set the new association timestamp, while the client does all of this. Fortunately, when the client sends an association request, the AP can notice that this client was already associated but at the same time do all the sets/resets of a normal association. This way, AP and client will keep the same timestamps and counters. Or, in other words, any discrepancies between those interfaces will be due to problems that a management application should solve!

The other way around, if an inactivity disassociation occurs and the client reassociates immediately (which is done by the firmware, not the driver) this might not be noticed by the BSSID check done every second. In this case, the AP noticed the disassociation and resets the counters and so on, but the client doesn't.

Here is an example of a client not noticing the disassociation. Interface number 2 is the AP, number 4 is the client. The AP has his last disassocation 16 seconds ago, and the corresponding reassociation 15 seconds ago. The client however, thinks he has been associated for the past 10 minutes. The 352 days values are zero timestamps, those events have not occured.

```
l80211StaLastAssoc.2.1 = Timeticks: (3048918077) 352 days, 21:13:00.77
l80211StaLastAssoc.2.2 = Timeticks: (1507) 0:00:15.07
l80211StaLastAssoc.4.1 = Timeticks: (62002) 0:10:20.02
l80211StaLastDisassoc.2.1 = Timeticks: (3048918079) 352 days, 21:13:00.79
l80211StaLastDisassoc.2.2 = Timeticks: (1623) 0:00:16.23
l80211StaLastDisassoc.4.1 = Timeticks: (3048918080) 352 days, 21:13:00.80
```

In the case of the WL interlinks, an inactivity disassociation of the client station by the HostAP will probably not occur, because of deamons sending packets once in a while ($\leq$ 2 minutes). Also, it is unlikely that interlinks will be blocked by physical objects. These observations suggest that the possible discrepancies described above are unlikely to occur. Also, it is highely likely that the interlink stations remain in the driver cache.

4.1.5 Performance

The additional information calculated and stored by the driver should have no effect on interface performance. Specifically, the packet processing time must not increase. The calculation of signal averages for every received packet is not considered computationally expensive. The interruption every second for bandwidth average calculations, (dis-) association detections and such might take some time depending on number of entries in the driver cache. Note that every interface has his own call, so calculation time increases with the number of interfaces. Specifically the BSSID record read from the interface takes some time. However, since we only do this once a second this is considered negligable.

4.2 SNMP agent extension

As discussed in section 3.2, code is added to the existing SNMP agent, `snmpd`.

The `net-snmp` package comes with a MIB compiler, `mib2c`. This perl script takes a MIB file as input and produces skeleton `C` code for inclusion in `snmpd`. There are several 'modes' in which this compiler can operate. The most simple one is the *scalar* mode, in which a MIB containing just scalar variables is used. The skeleton code for the implementation of such a MIB is rather simple, with an initialization function and handlers (read-only or read-write) for all the variables. The developer only needs to fill in the corresponding values. Of course, one needs to make sure the correct types are returned, the correct variable memory size is passed and that the variable remains in memory after handler termination (so no local non-static variables).

It soon became clear that not only scalar variables are needed. The WEP key values should be kept in a (variable sized) table, as well as the multiple SSIDs an AP can have. And when we start storing information on nearby stations we have a very dynamic table.

The `mib2c` compiler has several modes for creating tables. For dynamic tables, with lots of external interaction (with the kernel) the *iterator* table is most appropriate. This table 'iterates' over existing rows in the table, until the requested row is found. Then the handler is called, passing information about this row and the requested column and mode (GET/SET). The handler can then take appropriate action and respond correctly. The developer should implement the start of the iteration and the function for going through the next iteration. These functions return index values, which are used to determine whether the requested row has been reached. See also source code and doxygen pages for more information.

Implementing a table might be tricky, trying to implement a table as a row of a table (so called 'table-within-table') turns out to be impossible. According to the SMI definition, tables shall only have scalars as its rows. But we really do need this, in the case of multiple interfaces on a system. We need to provide a table of interfaces, each of them containing several tables themselves as mentioned above.

The solution is to create augmenting tables, which are really stand alone tables but are connected to the interfaces table by a second index. Thus, those augmented tables become double indexed. Fortunately, this is a relatively minor extension to the iterator table code.

Once more the reader is referred to the doxygen pages for information on adding variables to the tables.

The `snmpd` extensions discussed above are seperated from the code for communication with the wireless interface (driver). These are placed within a single source file. This file and it's functions have been designed to be very easy to use from other sources. This might not always lead to the most efficient solution, but it does ensure robustness and simple usage. Thus, `snmpd` source code remains legible and understandable. Also, when porting or upgrading to different drivers, network cards or operating systems, the `snmpd` code can remain untouched.

One final note about the station table, which returns information available from the drivers' station cache. This cache is actually cached too, for a predefined number of seconds (currently 5). This is done for `snmpwalk` requests, which read all the variables in the table. When we have to do a call to the kernel for every variable requested, this will slow down the agent significantly. Also, there is a hazard in changes made by the kernel to the cache while the agent is walking through it's table. The 5 second boundary is chosen such that it is fast enough to have new information at every request (unless two requests follow within this interval) and such that one request to the driver for cache information is sufficient for the entire `snmpwalk`.

5 SPLASH

The final goal of this project is to be able to create SNAP packets that can have access to 802.11 specific management information. SPLASH has been designed to let SNAP packets perform SNMP requests in general, so extending the `snmpd` with 802.11 information cleares the way for those 802.11 aware SNAP packets.

Unfortunately, SPLASH has been developed on Linux. Now source code developed for Linux *should* compile clean on FreeBSD systems, unless Linux specific code is used.

5.1 Porting

5.1.1 SNAP

Aside from minor changes like including extra header files and some `structs`, effort was put in rewriting the code that was using the Linux `proc` filesystem, specifically regarding routing and interface information. By reading files located in this system, the user (-code) has access to detailed system information. The `proc` filesystem however is not installed on FreeBSD by default. Adding `procfs` support includes only minor system and interface information. There exists no file for the routing table, which SNAP needs for implementation of the `forw` and `send` commands. The new FreeBSD implementation uses `sysctl` to get the routing table from the kernel. The information is stored locally, and a nexthop function is implemented.

The `snap_exec` utility has been extended to receive multiple messages by calling with the `-l` parameter.

5.1.2 SNAP services

The following observations hold for the SNAP services.

- **proc**. Obviously, the SNAP service 'proc' is not available on FreeBSD. Even if `procfs` support is installed, this is only minimal.
- **if**. The interface service uses the route file for the same purpose as SNAP does, i.e. calculating next hops. The code that is now used in SNAP can be trivially applied here, by linking the appropriate SNAP source file (`router.c`). In this project that has not been done, mainly because the interface services are not used by the example SNAP packets (section 5.2) and could thus not be tested. Moreover, a subset of the services provided here is also provided by the snmp service.
- **route**. The route service uses `ioctl` calls to add or delete routes from the routing table. Similar calls exist in FreeBSD, but different structures are used in the calling process. Porting is possible, but not done in this project for similar reason as under the above point.
- **memmap**. No porting necessary.
- **test**. No porting necessary. Some string manipulation functions have been added however, specifically string concatenation, IP address to string conversion and concatenation of integer to string. These functions are useful for creating more extended `demux` output (at the source). For example, combinations of IP address, status message and request specific information.
- **snmp**. No porting necessary, main addition is the SNMP getnext request. This is also the service to put additional 802.11 specific services when those turn out to be required, for example a service returning the indexes of wireless interface cards.

5.2 Example packets

For testing and demonstration purposes, some SNAP packets have been designed. The packets have been documented, so for details refer to the packets themselves. Output printed below is created on the Wireless Leiden network, see their nodemap [6].

Because of security issues, no read-write community or users have been configured. Thus, all variables are read-only. This forces the packets aim at information gathering, rather than problem solving.

To check the status of SNAP deamons, `dping` (which directly 'pings' the destination) and `ping` (which is processed by the SNAP deamons on intermediate hosts) packets have been designed.

Also, traceroutes have been created that can start from an arbitrary host. Those packets move immediately to the specified starting point, where they start the normal traceroute. On every hop, a SNAP packet is sent back to the source, containing a message. Another possibility is to accumulate all the data in the SNAP packet, and display all this after returning to source. This method fails however, when one of the links is down. With the chosen implementation, it is immediately clear where a link failure occurs.

```
[172.17.8.1 arrived at start]
[172.20.0.1 next hop]
[172.18.8.1 arrived at dest]
```

In addition, those traceroute packets can be programmed to return SNMP information. In the example below, the `system.sysName.0` is returned, the system's name.

```
[172.17.8.1 arrived at start  CNodeCope.wLeiden.NET]
[172.20.0.1 next hop  CNodeJorg.wLeiden.NET]
[172.18.8.1 arrived at dest  CNodeJasper.wLeiden.NET]
```

A more advanced packet is capable of calculating the incoming and outgoing interfaces. When such an interface is in BSS mode, the MAC address of the HostAP can be read from the `l80211IfaceCurrentBSSID` variable. Combining this information, an SNMP request can be performed to read statistical information about the link client-AP. Traceroute packets have been created to read link quality information and return this to the source. Note that when the interface is in HostAP mode, the packet cannot immediately know what the MAC address of the client side of the interlink is. Of course solutions exist, for example remembering the client MAC address while going back and forth over the interlink.

Finally, a tracelist packet has been created to walk a predefined path to it's destination, and then back to the source again. By looking up link quality information of each outgoing link, we get information on all the links.

```
[172.19.128.1      2cope.UniGor2.wLeiden.NET      link quality =  40]
[172.17.8.1        2jorg.Cope0.wLeiden.NET        link quality =  12]
[172.20.0.1        centrum.Jorg0.wLeiden.NET      link quality =  nonBSS]
[172.18.8.1        2jorg.Jasper1.wLeiden.NET      link quality =  14]
[172.20.0.1        centrum.Jorg0.wLeiden.NET      link quality =  nonBSS]
[172.17.8.1        omni.Cope2.wLeiden.NET         link quality =  nonBSS]
[ok]
```

The second column is the stationname of the outgoing link, the third column contains the link quality to the HostAP. When the outgoing link was HostAP itself, nonBSS is returned. Obviously, this is the case for half of the interfaces. The packet gives us an overview of the interfaces used, and the corresponding link quality. In the above example, we see that the link between node Unigor and node Cope is pretty good. A more intuitive representation could be created using a script.

6 Conclusion

The Wireless Leiden network is the only Dutch WiFi network of considerable size (spanning 20 kilometers, with more than 25 nodes). In this project support for SNMP 802.11 specific requests and SNAP packets has been added to the network nodes. This provides possibilities for new interesting research. The example packets provide a demonstration of the possibilities on such a network. Many other packets can now be written, based on 802.11 specific request alone, or in combination with standard `mib-2` information.

During 802.11 MIB design, the comments and extensive reviews received from Harrie Hazewinkel and Dirk-Willem van Gulik have been very helpful. Regarding installation and configuration of the software on the Wireless Leiden nodes, assistance received from the *techniek* mailing list, specifically from Hugo Meiland and Dirk-Willem van Gulik, has made fast progress possible and is much appreciated.

References

[1] `IEEE8021dot11 MIB` http://standards.ieee.org/getieee802/download/MIB-D6.2.txt

[2] `Cisco MIBs` ftp://ftp.cisco.com/pub/mibs/v2/

[3] `Doxygen` http://www.stack.nl/~dimitri/doxygen/

[4] `Source documentation` http://www.liacs.nl/home/tmoerlan/WL/doxy/html/

[5] `Smilint` http://www.ibr.cs.tu-bs.de/projects/libsmi/

[6] `Wireless Leiden nodemap` http://wiki.wirelessleiden.nl/wcl/cgi-bin/moin.cgi/NodeMap