A Practical Hand-on Guide for Deploying a Large-scale Wireless Multi-hop Network

Adel Aziz
EPFL
Lausanne, Switzerland
adel.aziz@epfl.ch

Julien Herzen
EPFL
Lausanne, Switzerland
julien.herzen@epfl.ch

Abstract

The knowledge of how to exactly deploy a wireless multi-hop network can be of interest for both researchers and engineers. Engineers may be interested in the possibility of extending the coverage area of their traditional access point and to provide ubiquitous Internet connectivity at a significantly reduced cost and some communities have already started to do so [1]–[3]. Researchers need large-scale testbeds offering a certain level of flexibility in order to study the behavior of existing protocols, validate their analytical model and prototype their novel algorithms. Indeed, experimental validation is a key phase in order to transfer the academical knowledge from a lab to the real world and this requires the deployment of a testbed. As researcher working in between theory and practice, we needed to deploy such a testbed. Since the beginning our deployment in 2007 we faced many challenges until reaching the current status of our testbed today in 2010 [7], mostly due to the fact that we could not find in the literature a complete hand-on guide of how to deploy such a network. In order to bridge this gap, we provide this technical report in order to share the know-how we acquire while building this 12 km^2 testbed on the EPFL campus.

I. Introduction

Wireless mesh networks have always been designed to become a commercial product that will deliver high-speed Internet access to regions that are still unserved nowadays. In their beginning, in 1994 under the name of *massive array cellular system*, wireless mesh network were thought to become a cost-effective alternative to replace the last-mile infrastructure in the large metropolitan cities [12]. With the explosion of wired broadband connectivity in developed countries, the business case has evolved to provide access to places where the infrastructure is inexistent or unavailable as it is the case in developing countries or emergency situations. Nevertheless, a need that remains unchanged in this field is the necessity to result in solution that can be demonstrated to work well in practice.

Research in mesh networks thus requires to complete a full cycle consisting of analysis, simulation and experimental evaluation in order to make true progress. As a result of this fact, we decided to build a large-scale wireless mesh network to support our research. Similar infrastructure did exist such as the MIT roofnet [2] or the Magnets [10] (currently renamed to BOWL[1]) testbeds. Nevertheless, as we could not have an easy day-to-day access to those testbed and their existed none in EPFL, we decided to build from scratch our own multi-hop testbed on campus. During this deployment, we faced multiple challenges and design questions and the goal of this chapter is to share some of the experiences and lessons we learned in this process.

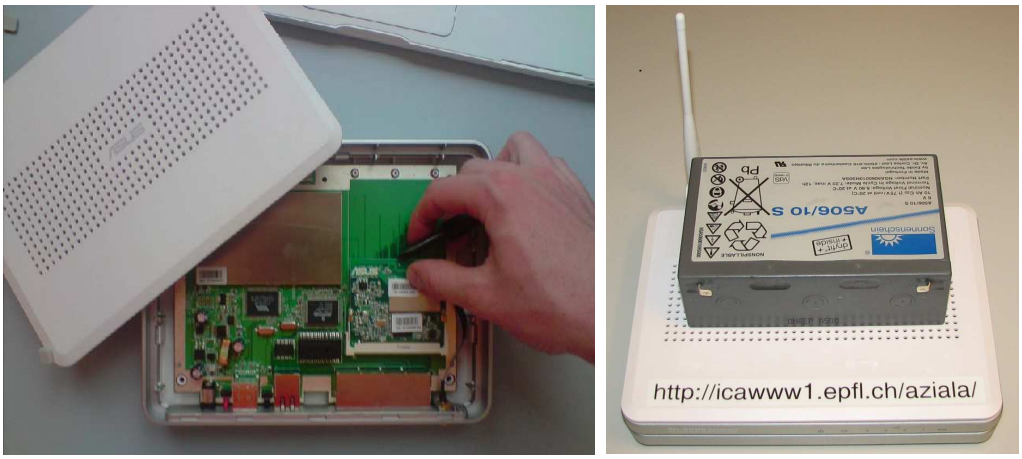
II. Selecting the Right Hardware and Software

A. Requirements and Challenges

The first step when deploying our indoor mesh network was to set the requirements for our testbed. In our case, we were interested in matching the following constraints.

- **Adaptability:** Even though our main interest lays in the scheduling problem of IEEE 802.11 mesh networks, we were interested in building a more general platform that would enable practical investigations of a wide range of research scenarios such as routing, user tracking, mobility, etc.
- **Programmability:** Due to the research-orientation of our deployment, we chose our hardware and software in order to provide adaptability even if it is at the cost of a small drop in relative performances. The first effect of this requirement appears in our choice of the firmware (openWRT[2]) and the driver (MadWifi[3]) that are both open-source and thus enable us to modify the source code in order to meet our research needs.
- **Large-scale:** Having already performed small-scale experiments (e.g., in the basement of our faculty building), we were interested in providing a platform that is closer to a real deployment. Toward this goal, we designed our testbed to cover six buildings of our campus with different node densities between the buildings.

[1]Berlin Open Wireless Lab (BOWL): http://bowl.net.t-labs.tu-berlin.de/
[2]OpenWRT firmware: http://openwrt.org/
[3]MadWifi driver: http://madwifi-project.org/



(a) (b)

Fig. 1. The Asus WL-500g Premium that we used in our testbed and where: (a) we changed the WiFi mini-PCI card to an Atheros-based one, and (ii) we designed a technique to power our router through a battery, which enabled us to support mobile scenarios.

- **Price constraint:** Finally, the last constraint we needed match was the budget constraint that forced us to evaluate different alternatives, in order to find the hardware that provides the best tradeoff between the different requirements. We also stress that due to budget reasons, we did not consider the use of Software Defined Radio (SDR) in our architecture, even though it was the option offering the highest level of programmability.

B. Hardware Description

In order to decide on the hardware forming our wireless testbed, we screened the different options and tested the four following routers as potential candidates (from most expensive to cheapest):

- **Mikrotik RouterBoard 532:** It was the first router we tested experimentally [8], which was available at a price around 250 \$. This router is based on a CPU of 400 MHz, with 64 MB of DDR RAM. It provides two mini-PCI connectors that allows the use of 2 independent wireless interfaces.
- **Asus WL-500g Premium:** It is a small and efficient residential router that cost around 90\$. It is based on a 266 MHz CPU (that can safely be overclocked to 300 MHz) with 32 MB of RAM and 8 MB of flash. It provides a mini-PCI slot that initially contains a Broadcom wireless card (which can easily be replaced by any other mini-PCI card) and two USB ports that are really useful to extend the storage capacity with USB memory sticks.
- **Linksys WRT54GL:** It is a popular home router that cost around 60 \$. It is based on a 200 MHz CPU with 16 MB of RAM and 4 MB of flash. It has a built-in wireless card that cannot be removed and thus does not offer the flexibility that mini-PCI gives.
- **La Fonera:** It is the smallest and cheapest router we tested with a price of around 20 \$. It is based on a 180 MHz CPU with 16 MB of RAM and 4 MB of flash.

After testing the routers under different settings, we found that the Asus WL-500g Premium offered the best price vs. performance tradeoff. The three major points that made us select the Asus instead of the Linksys or La Fonera are:

1) it provides the flexibility to change the wireless card (i.e. it uses a mini-PCI instead of a built-in wireless interface), which is useful to easily adapt to changes in the MAC technology;
2) it has two USB slots that can be used to extend the storage memory. This feature is particularly useful, because it allows to easily log arbitrarily large trace files on the router without any problem;
3) it does not have the 4 MB-flash limitation. Indeed, 4 MB is pretty limited in the case some useful software, such as Click [11], are included in the OpenWRT firmware image.

Moreover, when purchasing our hardware in 2007, we chose the Asus over the Routerboard for two main reasons. First, the USB capabilities were a useful feature provided only by the Asus. Second, the small CPU gain of the Routerboard was not justifying the price difference, which was an important factor as we were planning to buy around 60 routers (i.e., around $10,000$ \$ difference in the total price). We note that nowadays, in end 2010, an interesting option that we are considering for upgrading our testbed are the Alix boards from PC Engines, which have the advantage to run over an x86 architecture instead of mipsel.

The Asus routers were shipped with a power plug and a standard Broadcom mini-PCI WiFi card. These settings are good for a standard home user, but we needed to operate two modifications/enhancements in order to support the research scenario we had in mind (i.e., modifying/monitoring the MAC parameters and supporting mobile scenarios).

- **Allow modification/monitoring of MAC parameters:** In order to use the open-source MadWifi driver, we needed to change the mini-PCI card to an Atheros-based one as depicted in Figure 1.a. We used the NMP-8602 Atheros card that runs on IEEE 802.11a/b/g.
- **Enable mobility:** In order to eliminate the requirement to plug our router to the electrical grid (which prevents mobility), we equipped some of our devices with 6 V batteries as shown in Figure 1.b. To be able to meet the high amperage requirement to support the wireless interface of our router, we used A 506/10 batteries. Moreover, as the Asus routers works on an input voltage of 4.5 V, we added two inductances to the cabling connecting the battery to the router.

In addition to these modifications, we also equipped each router with a 2 GB memory stick that is automatically mounted during the boot-up of the router.

C. OpenWRT Firmware

OpenWRT is a free open-source and Linux-based firmware. We decided to flash our routers with this distribution, because of the facility with which we can (i) install existing program and (ii) deploy our own C code. We provide in Appendix B a detailed HowTo on the methodology that needs to be followed in order to (i) build an image that is ready to be flashed on the routers, (ii) cross-compile our own program to run on the mipsel architecture of the Asus, and (iii) flash the new image to the routers and make the first boot-up configuration procedure.

A particularly interesting software to install in the OpenWRT image is the Click modular router [11] that allows to completely control the processing/routing of a packet since it arrives to the router until it leaves it.

D. MadWiFi Driver

MadWifi is an interesting, but challenging wireless driver. On the one hand, its advantage comes from its open-source code that leaves the room for easy modification and enhancement of the protocol. On the other hand, it also suffers one of the typical drawback from open-source software as opposed to industrial products. For example, we noted that some of the supposedly working commands were only partially implemented (or even not implemented at all).

One of the key advantage that made us decide for MadWifi is that it provides an easy access to multiple MAC parameters of IEEE 802.11, such as the CW_{min}, CW_{max}, RTS mode on/off, etc.
Nevertheless, we needed to modify ourself the source code in order to achieve three objectives

1) We unlocked the modification, via the *iwpriv* command, of the MAC layer parameters for Best Effort traffic (BE) that is the standard type of traffic in the network.
2) We enabled the use of multiple MAC queues (up to 4 or 8) as it is proposed in the IEEE 802.11e standard to achieve Quality of Service. Moreover, the possibility of using different queues is relatively useful in congestion-control scheme as it allows to use different class of service depending on the nature of the next-hop.
3) We coded three new functions to the *wlanconfig* command in order to access the instantaneous buffer queue occupancy and to access and modify the maximal MAC buffer length (usually locked to 50 packets).

As the technical details behind each modification are rather complex, we defer their description to Section V. Moreover, we stress that nowadays, new open-source drivers such as ath5k and ath9k have been launched as follow-ups of MadWifi in order to supports the new generations of the IEEE 802.11 family, such as 802.11n.

III. Building, Installing and Configuring OpenWRT

OpenWRT [4] is a great open-source firmware that allows you to install a Linux-based system in your router and to easily cross-compile your own programs for its architecture. For a first quick installation of OpenWRT, one can directly download from their website the binary source that corresponds to the router architecture. Nevertheless, we recommend more advanced users to install the OpenWRT build-tree on their computer in order to build their own image for the router and add any package they are interested in. In the remainder of this section, we describe all the step-to-step procedure to follow in order to install the release 8.09 of OpenWRT on your router (note that additional information is also available in the wiki of the openWRT website).

A. Getting the Source and Compiling on Your Computer

The first step consists in downloading the openWRT buildtree and you do so by deciding in which directory you want to install the buildtree (we call it <*your-path-to-openwrt*>) and typing the following commands:

cd <***your-path-to-openwrt***>
svn co svn://svn.openwrt.org/openwrt/branches/8.09
svn co svn://svn.openwrt.org/openwrt/packages/

Following this procedure, you end up with two directories **8.09/** and **packages/**. Then to make all the packages of **packages/** accessible to openWRT, you need to type:

cd <*your-path-to-openwrt*>*/8.09/package*
for i in ../../packages/*/*; do ln -s $i; done

The next step is useful to directly include the desired configuration files directly into the built image. First we create the useful directories as follows:

cd <*your-path-to-openwrt*>*/8.09*
mkdir files
cd files
mkdir etc
cd etc
mkdir config
mkdir dropbear
mkdir init.d

Then we create the five following files inside those directories.

1) 1^{st} *File: "files/etc/config/system":*

```
#### Set the hostname
config system
option hostname TAP1
```

2) 2^{nd} *File: "files/etc/config/network":*

```
#### VLAN configuration
config switch eth0
option vlan0 "1 2 3 5*"
option vlan1 "0 5"
option vlan2 "4 5"

#### Loopback configuration
config interface loopback
option ifname "lo"
option proto static
option ipaddr 127.0.0.1
option netmask 255.0.0.0

#### LAN configuration
config interface lan
option ifname "eth0.0"
option proto static
option ipaddr 192.168.1.1
option netmask 255.255.255.0

#### DMZ configuration
config interface dmz
option ifname "eth0.2"
option proto static
option ipaddr 192.168.10.1
option netmask 255.255.255.0

#### WAN configuration
config interface wan
option ifname "eth0.1"
option proto dhcp

#### Wireless configuration
config interface wifi
option ifname "ath0"
option proto static
option ipaddr 10.10.10.1
option netmask 255.255.255.0
```

3) 3^{rd} File: "files/etc/config/wireless":

```
#### WiFi settings
config wifi-device wifi0
option type atheros
option channel 13
option agmode 11b

# REMOVE THIS LINE TO ENABLE WIFI:
# option disabled 1

config wifi-iface
option device wifi0
option network wifi
option mode adhoc
option ssid aziala
option bssid 02:ca:ff:ee:ba:be
option encryption none
```

4) 4^{th} File: "files/etc/dropbear/authorized_keys":

```
# File containing the SSH public keys of all the authorized machines:
ssh-rsa AAA...VhZw== adel@adel-desktop
ssh-rsa AAA...ISqw== julien@icsil1-pc12
...
...
```

5) 5^{th} File: "files/etc/init.d/done":

```
#!/bin/sh /etc/rc.common
# Copyright (C) 2006 OpenWrt.org

# REGULAR TASKS to be done when booting:
START=95
boot() {
[ -d /tmp/root ] && {
  lock /tmp/.switch2jffs
  firstboot switch2jffs
  lock -u /tmp/.switch2jffs
}
# process user commands
[ -f /etc/rc.local ] && {
  sh /etc/rc.local
}
# set leds to normal state
. /etc/diag.sh
set_state done

# Then add your OWN TASKS to be done when booting:
# TASK 1: Launch WPA
ifconfig eth0.1 promisc
wpa_supplicant -c /etc/config/wpa_supplicant/wpa_supplicant.conf -i eth0.1 -D roboswitch -B

# TASK 2: Set the channel rate to 1M
iwconfig ath0 rate 1M

#TASK 3: Mount USB drive
mkdir /root/mnt
mount /dev/sda1 /root/mnt
}
```

Once you are done editing the above configuration files, you are almost ready for cross-compiling in your computer an image that will be ready to be flashed on your wireless router. The only remaining step consists in selecting your router profile (to set for which architecture the compilation should be done) and the programs that you want to be included as packages in the image. To do so, you need to type:

make menuconfig

and this makes you enter a pretty intuitive graphical configuration menu with the following categories.

- **Target System**: This fields allows you to specify the platform of your router. Our Asus WL-500gP routers are based on the Broadcom BCM94704 platform, thus we select "Broadcom BCM947xx/953xx [2.6]". The number in bracket ([2.4] or [2.6]) is the version of the Linux kernel that is selected. In the earlier version of our testbed, we had our routers running the [2.4] version without any problem. Recently, we flashed all our router with a [2.6] version for a reason of compatibility with the Click package [11]. In case you are using other routers than the Asus WL-500gP and do not know the platform they are based on, you may find this information on [4].
- **Target Profile**: This field allows you to specify the model of your router.
- **Base system**: This category allows you to include additional library in the image of your router (this is useful if you start coding your own program that require a specific library).
- **Network**: This category provides you with a huge amount of package that you can include in your image. However, before adding too many packages, you must keep in mind that the more selected packages, the larger the size of your image after compilation and the allowed image size is restricted by the Flash memory of your router (4Mb for most routers and 8Mb for the Asus WL-500gP [1]). In our testbed, the package we were interested in adding to the standard configuration are: click, iperf, tcpdump and wpa-supplicant[5].
- **Kernel modules**: This category allows you to include useful modules and driver. In our case, we were particularly interested in (i) adding the open-source madWiFi driver to our image by selecting *kmod-madwifi* in the "Wireless Driver" directory; (ii) adding USB support to benefit from 2Gb extra memory through USB sticks. To do so, we had to select the modules *kmod-usb-core*, *kmod-usb-ohci*, *kmod-usb-storage*, *kmod-usb-uhci* and *kmod-usb2* in the "USB Support" directory. Additionally, we also had to include the modules *kmod-fs-ext2*, *kmod-fs-ext3*, *kmod-fs-hfs*, *kmod-fs-vfat*, *kmod-nls-cp437*, *kmod-nls-cp850*, *kmod-nls-iso8859-1* in the "Filesystems" directory.

The above list describes the package that we found, given our needs, the more useful to be included in our image. However, it is clearly not an exhaustive description of the multiple possibilities of available configurations, and we would recommend the interested readers to go quickly through the different categories of the configuration menu in order to have a clearer idea of what is available. Moreover, if you know the name of a program you would like to include without knowing its location on the configuration menu, a useful command is the search command that is obtained by typing "/" in the menu.

Now that you are done with the configuration, you are ready to launch your first compilation of OpenWRT by typing:

make V=99

where the "V=99" parameters provides you with all the debugging messages (just type **make** if you do not want to see them). During this first compilation, your computer needs to be connected to the Internet as openWRT will download the needed source code and cross-compile it. All this process takes a pretty long amount of time for the first compilation, so you may want to launch it over night. Note that OpenWRT only compiles everything at the first compilation (or if you execute a **make clean**). In the future compilations, only the modified files will be compiled and thus, the whole process will be significantly faster.

If you followed all the previous steps correctly and went through the compilation process without any errors, you should now obtain, in the *bin/* directory, different images *openwrt-*.trx* and *openwrt-*.bin* (the *.trx* is the one used for the Asus router). The next step is then to flash your router with the obtained image and we describe two ways of going it either through: (i) TFTP or (ii) SSH.

6) Flashing the image through TFTP: In order to follow this procedure, you need to start your router in the TFTP mode. For the Asus WL-500gP, this is done by keeping pushed down the RESTORE button (in the back of the router), while plugging in the power cord. If the Asus successfully started in TFTP mode, you notice it by seeing the READY light blinking. Once in this mode, the router has a TFTP server running that accepts a *.trx* image (i.e., the firmware) to be flashed. To do so, you need to plug your computer on the LAN port of the Asus router and type the following command

cd <*your-path-to-openwrt*>*/8.09/bin*
tftp 192.168.1.1
tftp> binary
tftp> rexmt 1

[4] `http://wiki.openwrt.org/oldwiki/tableofhardware`

[5] As explained further in Section IV-C, we are interested to have 802.1X (WPA) available on the routers. However, the default version available in openWRT does not work with our configuration. Section IV-C explains what to do in a scenario similar to ours.

tftp> put openwrt-brcm47xx-squashfs.trx

Once the TFTP upload is completed, the router initializes itself with the new firmware and this is a critical phase. Indeed, **do NOT turn off** your router during the 2 minutes following the TFTP transfer or you might break it without being able to recover. After, this 2-minute delay, you can unplug the power cord and plug it again and then you can start connecting to your router and configuring it. For details on how to do this, see the Section III-A.8

7) Flashing the image through SSH: This alternative way of flashing the router can be relatively useful if it happens that you can access the router in the normal mode, but not in the TFTP mode (we do not know the cause of this problem, but it sometimes happened to us). To perform your flashing through SSH, you need to plug your computer on the LAN port of the Asus router and type the following command

cd <*your-path-to-openwrt*>*/8.09/bin*
scp openwrt-brcm47xx-squashfs.trx root@192.168.1.1:/tmp[6]
ssh root@192.168.1.1

Then once logged in the router TAP1, you type

root@TAP1# **mtd write openwrt-brcm-2.4-squashfs.trx linux && reboot**

At the end of this process the router will automatically reboot by itself (you do not need to touch the power plug) and you will be able to connect to it and configure it by following the instruction of next section.

8) First connection to the router and its configuration: A router that has been just flashed cannot be directly accessed via *ssh*, because the connection is blocked. To activate this connection and allow the connection through the wireless and WAN port, you need to set a password and follow the following steps:

telnet 192.168.1.1
root@TAP1# **passwd**
root@TAP1# **/etc/init.d/firewall disable**
root@TAP1# **iptables -F**

After this you should be able to connect to your router in *ssh* and use your router normally. Moreover, if for security reasons you want to disable the connection via password, you can easily do that by editing the file **/etc/config/dropbear**.

IV. Extending OpenWRT with some useful packages

In the previous section, we explained how to build your own image of openWRT by including some of the existing packages directly into the image through the **menuconfig** window. However, one might be interested in adding later on some extra packages to a working router without wanting to go through the process of flashing it once again. In this section, we will show how this can easily be done and how you can easily cross-compile add your own packages/programs into the **menuconfig** window.

A. Basics for Adding a Package on your Router

After having compiled openWRT, one can see that many packages (the ***.ipk** files) are created in the folder **bin/packages**. Moreover, in case you want to create a new ***.ipk** file (for example you want to add the vpnc package to an already flashed router), you just need to select this package in the **menuconfig** window and to run **make** once again. Once the compilation process ends, you have a new package created and you install it following the procedure below (note that this example is for the Asus router that is built on a mipsel architecture, router based on another architecture should follow a similar methodology):

cd <*your-path-to-openwrt*>*/8.09/bin/packages/mipsel*
scp vpnc_0.5.3-1_mipsel.ipk root@192.168.1.1:/root
ssh root@192.168.1.1

Then once logged in the router TAP1, you simply type

root@TAP1# **opkg install vpnc_0.5.3-1_mipsel.ipk**

B. Adding the Click Modular Router

Before describing the procedure of how to cross-compile **click** [11] for openWRT, we note that this is a relatively big package. Therefore, it might be wise to directly incorporate it in the flashed image instead of installing it afterwards with the **opkg** command.

In our deployment, we chose to use the MultiFlowDispatcher library [13], which allows to dynamically span sub-element at run time (e.g., in per-flow queueing, we only create a queue once a new flow appears in the network). In order to compile

[6]Instead of /tmp, you might want to adapt the directory in which you copy the image in the router. Indeed, you need to copy it onto a volume with enough free space.

Click with new library included (in this case MultiFlowDispatcher), we needed (i) to add the file **multiflowdispatcher.cc** in <***your-path-to-openwrt*>*/8.09/build_dir/mipsel/click-1/lib*,
(ii) to add the file **multiflowdispatcher.hh** in <***your-path-to-openwrt*>*/8.09/build_dir/mipsel/click-1/include/click*, and
(iii) to modify the Makefile in:
<***your-path-to-openwrt*>*/8.09/build_dir/mipsel/click-1/userlevel/Makefile.in*
in order to add **multiflowdispatcher.o** in the **GENERIC_OBJS** variable.

Now that you know how to cross-compile **click** for your router, you might be interested in adding your own elements to Click. To do so, you might want to read a bit on the basics of how to add an element for click on a computer (i.e., without cross-compilation) on the FAQ section of the Click website [7]. Then once you understand this process, the only difference to add your element in the openWRT buildtree, is that you should add your elements in the folder:
<***your-path-to-openwrt*>*/8.09/build_dir/mipsel/click-1/elements/local/*

Note that during the debugging process of your elements you might need to re-install many times the *click*.ipk* package on your router, as any modification of one of your elements is equivalent to a new version of click that contains the new element and that needs to be installed. Personally, we find this procedure rather cumbersome and we even encounter problem re-installing click through the standard **opkg** command. Instead, we found that a quicker way to re-install click on your router (that always worked for us) is to simply copy the binary file on your router. This simple technique is performed by typing:
scp <***your-path-to-openwrt*>*/8.09/build_dir/mipsel/click-1/userlevel/click* **root@192.168.1.1:/usr/bin/**

C. Adding 802.1x Support for a Secure Wired Connection

Indeed, one of the main goal of wireless multi-hop networks is to avoid using wires. However, if the network is deployed for research purposes, one should consider connecting the nodes to a wired network in order to be able to control them, deploy programs and perform maintenance tasks. If the nodes are deployed within your research institution's building(s), you hopefully will be able to use an already existing wired infrastructure.

In Section VI we present an useful way of remotely controlling the nodes. We describe here the procedure that we followed to connect the nodes using the wired network that was already present in the buildings.

Connecting the wireless nodes to a regular wired LAN should not represent a big issue, provided that ethernet interfaces are present on your hardware. However, some institutions require the users of the infrastructure to use secure protocols such as VPN[8] or 802.1X[9] (also known as WPA).

We had successful experiences connecting our nodes through VPN with openWRT, using the `vpnc` [5] open-source VPN client. However, the VPN gateway is disconnecting the nodes once every couple of hours, changing their IP addresses. A much more convenient way of connecting the routers while using a legacy secure protocol available in our institution is to connect the nodes using 802.1X. This way, each node is part of the school secure network and has a public IP address that almost never changes.

Indeed, such a configuration requires that each node connected to the wired network can access an 802.1X plug. In our case, we had to ask the IT department of the university to install one such plug in every office where we were willing to install a node, as very few such plugs were active by default.

Once the plugs in place, we installed a WPA (802.1X) client on the nodes. We use the well-known `wpa-supplicant` program [6] for this purpose. However, the default `wpa-supplicant` package (`wpa-supplicant-0.6.10` at the time of writting) does not work correctly with our configuration.

Here are all the steps that we follow to enable 802.1X authentication with OpenWRT 8.09 :

[7] `http://read.cs.ucla.edu/click/learning`
[8] `http://en.wikipedia.org/wiki/Virtual_private_network`
[9] `http://en.wikipedia.org/wiki/IEEE_802.1X`

- Get the `wpa-supplicant-0.6.9-2` package provided by Jouke Witteveen at `http://www.liacs.nl/~jwitteve/openwrt/8.09/brcm-2.4/packages/`, or on the Aziala website [7].
- Install it on the nodes using the package manager : `opkg install wpa-supplicant_0.6.9-2_mipsel.ipk`
- In our case, the config file is `/etc/config/wpa_supplicant/wpa_supplicant.conf` and it looks like this:

```
ctrl_interface=/var/run/wpa_supplicant
ctrl_interface_group=root
ap_scan=0
network={
key_mgmt=IEEE8021X
eap=TTLS
identity="p-aziala"
password="xXxXxXxX"
phase2="auth=PAP"
ca_cert="/etc/config/wpa_supplicant/Thawte_Premium_Server_CA.pem"
priority=5
}
```

  Of course, this file is relevant to our own configuration. In particular, the `.pem` file is the certificate required by the authenticator. In order to avoid to copy it on every node, we include it in the OpenWRT build tree[a].
- The WAN interface is eth0.1 with our configuration. We need to set this interface in promiscuous mode : `ifconfig eth0.1 promisc`
- The command to launch wpa-supplicant is: `wpa_supplicant -c /etc/config/wpa_supplicant/wpa_supplicant.conf -i eth0.1 -D roboswitch -B`.
  The option `-B` is to run wpa-supplicant as a daemon in background.
  We added this line at the end of `/etc/init.d/done` so that the nodes automatically authenticate on the wired network at boot time.

[a] `files/etc/config/wpa_supplicant/Thawte_Premium_Server_CA.pem`

D. Adding your Own Program/Package

Now that you master the technique of including existing program, you might be interested in including your own program (for example *HelloWorld.c*) in the openWRT tree in order to cross-compile it and produce a nice *HelloWorld.ipk* file that you will be able to install on your router and share with other people. To achieve this goal, you have to follow the procedure below:

1) **Create the OpenWRT Makefile:** The role of the Makefile is to make your package appear correctly in the **menuconfig** and to tell the cross-compiler where it should look for the source code (most likely locally for your own code, but it can also be online if you provide an URL). The Makefile for your program (here HelloWorld) is easily created by typing:
   # **cd <*your-path-to-openwrt*>*/8.09/package***
   # **mkdir HelloWorld**
   # **cd HelloWorld**
   # **cat > Makefile**

```
include $(TOPDIR)/rules.mk

PKG_NAME:=HelloWorld
PKG_VERSION:=1
PKG_RELEASE:=1

PKG_SOURCE:=$(PKG_NAME)-$(PKG_VERSION).tar.gz

include $(INCLUDE_DIR)/package.mk

define Package/HelloWorld
SUBMENU:=C
SECTION:=lang
```

```
CATEGORY:=Languages
TITLE:=Simple ”Hello World” Program
# URL:=http://website/.../helloworld
# DEPENDS:=+libpcap @!mips
endef

define Package/HelloWorld/description
HelloWorld is a basic program that prints the message ”Hello World” on standard outputs and then exits.
endef

define Build/Compile
$(MAKE) -C $(PKG_BUILD_DIR) \
CPPFLAGS=”$(TARGET_CFLAGS) -I$(STAGING_DIR)/usr/include -I$(STAGING_DIR)/include” \
all
endef

define Package/HelloWorld/install
$(INSTALL_DIR) $(1)/usr/bin
$(INSTALL_BIN) $(PKG_BUILD_DIR)/HelloWorld $(1)/usr/bin/
$(INSTALL_DIR) $(1)/usr/share/HelloWorld
endef

$(eval $(call BuildPackage,HelloWorld))
```

2) **Put your source code in the buildtree:** To do so, you first need to create the **.tar.gz** file that contains your program with its Makefile (note that this is a second Makefile different from previous point). For our Asus based on the mipsel architecture, you create this **HelloWorld.tar.gz** file by doing:
   #> **mkdir HelloWorld-1**
   #> **mv HelloWorld.c HelloWorld-1**
   #> **cd HelloWorld-1**
   #> **cat > Makefile**

```
CC = mipsel-linux-uclibc-gcc

all:
$(CC) HelloWorld.c -o HelloWorld
```

   #> **cd ..**
   #> **tar czf HelloWorld-1.tar.gz HelloWorld-1/**
   #> **mv HelloWorld-1.tar.gz <*your-path-to-openwrt*>*/8.09/dl***

Once you follow this process, you should be able to see and select your package through the **menuconfig** window. After selecting it and launching a compilation with **make**, you should see your package HelloWorld.ipk created under the *bin/packages/mipsel* directory and you can then install this **.ipk* file using the standard procedure shown at the beginning of this section.

E. Cross-compiling your Programm without Making a Package

Creating a package is a cleaner way to transfer your program into OpenWRT. Nevertheless, you might be interested in the first debugging phase to quickly compile your program without going through the work of dealing with the Makefiles.

If that is the case, you can also easily compile your program by using the cross-compiler that are available int the *staging_dir* directory from your Openwrt buildtree. For example, in the case you want to compile your program HelloWorld.c (or HelloWorld.cpp) for the Asus WL-500gP router (mipsel architecture), you only need to type:

cd <*your-path-to-openwrt*>*/8.09/staging_dir/toolchain-mipsel_gcc4.1.2/bin/*
./mipsel-linux-uclibc-gcc HelloWorld.c -o HelloWorld
HelloWorld root@192.168.1.1:/root/mnt

V. Hacking the MadWiFi Driver to Study/Modify the IEEE 802.11 Protocol

A. Unlocking the modification of MAC parameters

The IEEE 802.11e uses four different classes of service: Best Effort (BE), Background (BK), Voice traffic (VO), and Video traffic (VI). Nevertheless, by default in IEEE 802.11, all the traffic is queued as BE traffic. Moreover the parameters for BE traffic are locked by default in the MadWifi driver and this prevent us from changing MAC parameters such as CW_{min}, CW_{max}, etc. In order to unlock these modifications, the MadWifi driver needs to be hacked by commenting the corresponding part of the code in the function **ieee80211_wme_updateparams_locked()** of **madwifi-trunk-r3314/net80211/ieee80211_proto.c**. Note that a patch containing all the needed modification is available at [10].

B. Enable MadWifi to announce the FULL_BUFFER status to upper layers

By default MadWifi does not inform the upper layer when its queue is full and it directly discards the packet instead. This is pretty annoying if one plans to use an additional queue at the IP layer (for example with Click). Indeed this queue will always be empty, because it will always gives any new packet to the MAC queue regardless of whether the MAC queue has space to accept new packets or not.

In order to overcome this problem, we patch the driver by modifying some files both in MadWifi and Linux.
First, in MadWifi the files that need to modified are **madwifi-trunk-r3314/net80211/ieee80211_output.c** and **madwifi-trunk-r3314/net80211/ieee80211_proto.h**. In **ieee80211_output.c** the modifications take place in the methods **ieee80211_hardstart()** and **ieee80211_parent_queue_xmit()** (that has its return value changed from void to int). These changes consist in returning a value (i.e., an int) in order to tell the upper layer if the MAC has space available in its queue to accept new packets or not. Finally, the change in **ieee80211_proto.h** consists simply in updating the type of the function **ieee80211_parent_queue_xmit()** from void to int.
Then in Linux, the modifications consist in using this new feedback delivered by the MAC in order to only transmit a packet to the MAC if it has enough room to accept it. The modifications of the code take place in the files **linux-2.6.26.8/net/core/dev.c** and **linux-2.6.26.8/net/packet/af_packet.c**.
For the readability of this report we do not discuss all the changes here, but we provide the links for the patch we created with details modification both for MadWifi[11] and for Linux[12].

C. Add new commands to access MAC parameters

In our research, we were particularly interesting in knowing the queue occupancy of a node, but unfortunately this information is not given by default by the MAC driver. In order to satisfy our needs, we hacked the MAC driver in order to add some new commands that allow to:

- Enable/disable the use of the 4 MAC queues existing in IEEE 802.11e (by default this is disable).
- Get/Set the maximal MAC buffer size (by default this is 50 packets).
- Access the current MAC queue occupancy.

In order to achieve this goal, we extended the *wlanconfig* command with our own commands, and we note that a similar methodology than the one we used can be use to access/modify some other parameters of interest. More specifically, the files that we needed to modify to reach our objectives were 3 files in the folder **madwifi-trunk-r3314/ath/** (see the details in[13])

- **if_ath.c**
- **if_athioctl.h**
- **if_athvar.h**

2 files in the folder **madwifi-trunk-r3314/net80211/** (see the details in[14])

- **ieee80211_output.c**
- **ieee80211_var.h**

and 1 file in the folder **madwifi-trunk-r3314/tools/** (see the details in[15])

- **wlanconfig.c**

[10] http://icawww1.epfl.ch/aziala/P1_unlock_BEparam_modif.patch
[11] http://icawww1.epfl.ch/aziala/P2_2_enable_fullBuffer_MACsignal_madwifi.patch
[12] http://icawww1.epfl.ch/aziala/P2_1_enable_fullBuffer_MACsignal_linux.patch
[13] http://icawww1.epfl.ch/aziala/P3_1_enable_additionalMAC_commands.patch
[14] http://icawww1.epfl.ch/aziala/P3_2_enable_additionalMAC_commands.patch
[15] http://icawww1.epfl.ch/aziala/P3_3_enable_additionalMAC_commands.patch

VI. Installing and Using Net-Controller as a Network Management and Vizualization Tool

A. Overview

Net-Controller [9] is a tool that allows to easily control a mesh network. In addition, it also allows to generate dynamic plots representing the evolution of some values that are measured by the nodes.

Net-Controller runs on a centralized host and relies on a wired control network, as shown in Figure 2. Such a control network is needed in order to guarantee fast and stable access to the nodes without disturbing the wireless traffic.

It consists in two programs:

- The program that runs on the centralized host. Its purpose is to display the GUI and allow the user to centrally control the network and generate the plots. This is a Python program, whose main class is `NetController.py`.
- The program that runs on the network nodes. Its purpose is to allow each node to communicate some data to the contralized host. This data is typically values that have been measured by the network node and the user wants to plot, or the list of flows that the network node is aware of. This is a C program called `cidaemon`.

`NetController` and `cidaemon` communicate through UDP. In addition `NetController` sends commands to the nodes using SSH.

B. Installation

You can download Net-Controller from Sourceforge[16]. The latest version of the code can be downloaded with SVN using the following command:

```
svn co https://netcontroller.svn.sourceforge.net/svnroot/netcontroller netcontroller
```

Since the user interface is written in Python, it does not require to be compiled. One simply needs the following installed on the machine:

- `Python (>= 2.5)`
- `PyQT`
- `Matplotlib`

Then, once in the source folder of the user interface (`/ui`), it suffices to launch:

```
python NetController.py
```

`cidaemon` needs to be compiled in the following way (using `gcc` in this example):

```
$YOUR_GCC -Wall -lm -pthread -o cidaemon cidaemon.c
```

Where `$YOUR_GCC` denotes your version of `gcc`.

C. Configuration

Net-Controller uses one main configuration file and needs at least one map file, that maps node indexes to IP addresses.

[16] `http://sourceforge.net/projects/netcontroller/`

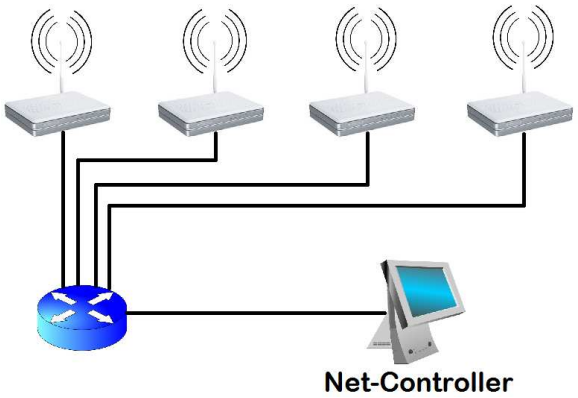


Fig. 2. Overview of the testbed

1) `netcontroller.cfg`: The file `netcontroller.cfg` describes the main options of the program. We explain them here.

- `interval` is the period (in seconds) between two updates of the plots. It can be useful to increase the plot update interval when the control network has large delays. However, setting intervals smaller than a typical RTT should not hurt much (although in this case the curves may be drawn in grey if the values are not received fast enough from the network).
- `flow_interval` is the period between two updates of the list of flows. This typically does not need to be small.
- `log_dir` is the directory in which the logs of the commands launched on the nodes will be saved.
- `trace_dir` is the directory in which the traces containing all the values that have been plotted.
- `map` is the name of the file that contains the mapping between the node indexes and their IP addresses on the control network. Specifying a valid file name is mandatory here, as pretty much every operations done by Net-Controller requires to communicate with the nodes through the control network.
- `trafficMap` is the name of the file that contains the mapping between the node indexes and their IP addresses on the actual wireless network. This is used by the traffic manager of Net-Controller in order to start/stop traffic.
- `max_commands` is the maximum number of simultaneous running commands. Each command is launched in a separate thread. If the limit is reached, new commands will wait until running commands are done. A value around 100 is probably good for most situations.
- `default_nbPoints` is the default number of points of the plots. For example, if the update period is one second, a `default_nbPoints` value of 60 means that the plots will represent the values received during the last minute, by default. Indeed, this value can be changed by the user once the GUI is launched.
- `direct_time` represents the direction of the time in the plots. If 'True' (direct time), the plot moves from right to left. If 'False' (reverse time), the plot moves from left to right.
- `ip` is the IP address of the central host that runs Net-Controller on the control network.
- `port` is the UDP port on which the user interface on the central host is listening. This is used to receive the values to plot and the lists of flows.
- `client_port` is the UDP port on which the nodes of the network are listening (on the control network). This is used to transmit the requests for the values to plot, or the requests for the lists of flows.

D. Using the graphical interface

In Net-Controller, the nodes are represented by integer indexes. After having specified a set of indexes, one can choose actions to perform related to these nodes. The main actions that are available at the time of writing are related to plots, commands or traffic. The graphical interface of Net-Controller during a typical execution is shown in Figure 3

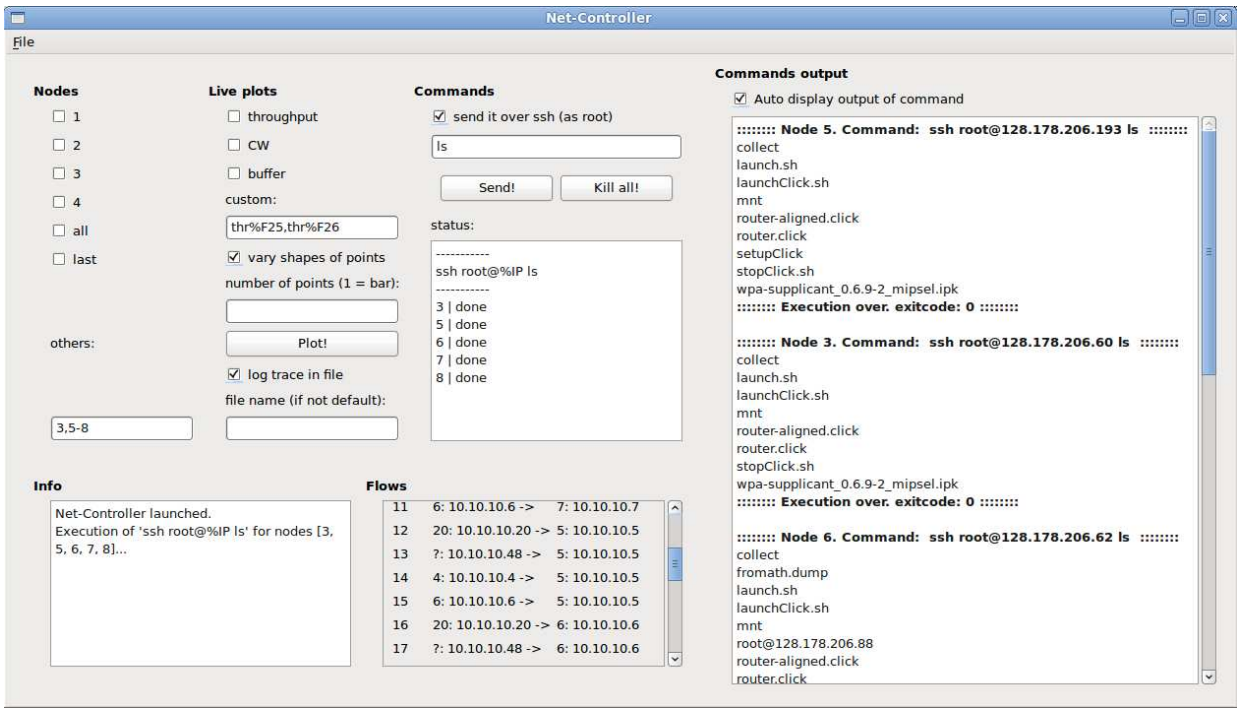


Fig. 3. Net-Controller GUI

The nodes one which perform actions can be selected on the left of the GUI ('Nodes' section). Shortcuts for selecting the nodes with indexes 1 to 4, all the nodes at once, or the node with the highest index, are provided in the form of checkboxes. To select any arbitrary set of nodes, one should use the 'other' textbox. In this textbox, it is possible to specify simple regular expressions. Set of contiguous indexes are noted using "-" between the smallest and the highest index, and several such expressions can be combined separated by ",". For example, if one wants to select the nodes $\{3, 5, 6, 7, 8\}$, the expression to use is "`3,5-8`".

1) Plotting stuff: The section 'Live plots' allows to select one or several parameters to plot. A parameter is just a keyword that uniquely identifies a quantity to plot. The 'Plot!' button opens a new window representing a plot of all the indicated parameters as measured in each of the selected nodes. The GUI offers shortcuts in the form of checkboxes for three parameters. 'CW' denotes the value of the IEEE 802.11 CW_{min} parameter, read directly from the Madwifi driver. For example, in our previous example, if one checks this box and clicks the 'Plot!' button, a window will appear showing the temporal evolution of CW_{min} for each one of the nodes $\{3, 5, 6, 7, 8\}$. The parameter 'buffer' denotes the occupancy of the MAC layer sending queue (here to, as read from Madwifi).

The parameter 'throughput' is a bit particular. Indeed, the throughputs that the nodes measure are related to the notion of flow. A flow denotes a tuple $< IP_{source}, IP_{destination} >$. All the measured throughputs are *per flow*, that means that the throughput represents what the selected nodes have received *for this flow*. That also means that the parameter has to denote the flow in which one is interested in. All the nodes of the network report to Net-Controller the flows they are aware of, and Net-Controller assigns unique IDs to the flows and displays them in the 'Flows' section of the GUI. Now, if one wants to plot the throughput flow with ID i, the name of the parameter is '`thr%F`i'. The checkbox 'throughput' is simply a shortcut for '`thr%F0`'.

It is possible to plot several parameters for the set of selected nodes. However, the parameters have to denote the same thing (i.e., use the same unit, since the same y-axis is used). It is currently only possible to plot several throughputs values (i.e., for several different flows) on one plot window.

An other particularity of Net-Controller is that it allows to plot parameters that are read through a Click socket [11]. In this case, a parameter name of the form '`click->element.handler`' will plot the temporal evolution of the handler `handler` of the element `element` (see Click documentation for more details about elements and handlers [11]).

If nothing is specified in the 'number of points' textbox, the default value for the number of points will be used. If the number of points is 1, the plot will display bars instead of curves.

By default, all the values that are plotted are locally stored in trace files, in the directory specified by the `trace_dir` option. If nothing is specified in the 'file name' textbox, a default name (that takes into account the exact creation time and selected nodes/parameters) will be used.

2) Sending commands: Once a set of nodes is selected in the 'Nodes' section, one can send some commands to all of them. By default, the checkbox 'send it over ssh (as root)' is checked. That means that all the commands that are typed in the textbox will be sent with '`ssh root@%IP `' prepended to them, where '`%IP`' denotes the IP address of the node to which the command is sent. For this to work properly, your public key should be present in the '`authorized_keys`' file related to the SSH configuration present on the nodes (for more details on how to setup SSH, see Section III). If you don't want to prepend '`ssh root@%IP `' (for example if the user that you want to use on the nodes is not root), you can uncheck the box. In this case, you can launch commands over the whole set of selected nodes using the following pre-defined variables: '`%IP`' (the IP address of the node) and '`%INDEX`' (its index). For example, if you want to create a local directory for each selected node, you can enter something like '`mkdir dir_%INDEX`. If you want to send a set of SSH commands with a custom user, say `joe`, you can enter something like '`ssh joe@%IP your-command`'. If you want to systematically be able to send SSH commands as `joe`, you should modify the prefix string in the `sendCommand()` method of Net-Controller (in the file `NetController.py`).

In addition, Net-Controller allows to send a special kind of commands to interact with Click handlers [11]. A commands such as '`click<-element.handler=value`' will set the handler `handler` of the element `element` to the value `value` (see Click documentation for more details about elements and handlers [11]).

The status of the command for each of the selected nodes is displayed in the 'status' textbox. The status can be among the three following states:

- `running...`: the command did not return yet.

- `done`: the command returned with an exitcode equals to zero.
- `failed`: the command returned with a non-zero exitcode.

For each command that is launched, its `stdout` and `stderr` outputs are appended to a file that contains such outputs for all the commands launched on the same node. These files are located in the directory indicated by the `log_dir` option. If the checkbox 'Auto display output of command' is checked (which is the case by default), the part of this file that concerns the last command is read and displayed on the right textbox once the command returns. That allows the user to quickly see the outputs of the commands.

3) The traffic manager: The traffic manager can be send through the menu `File -> Launch traffic manager` or with the keyboard shortcut `Ctrl+T`. The traffic manager is shown in Figure 4.

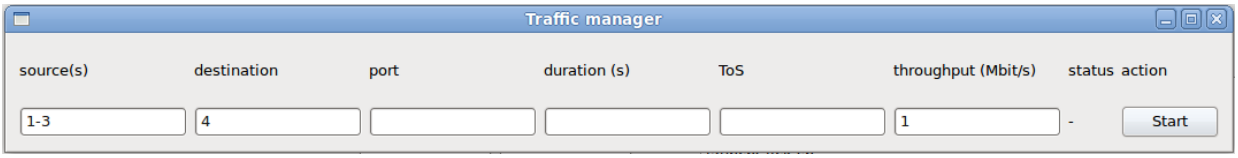


Fig. 4. The traffic manager

The goal of the traffic manager is to easily start/stop traffic between the nodes of the wireless network. The `iperf` program[17] is used to generate the traffic. Therefore, it is needed at the nodes for the traffic manager to work. The way the traffic manager works is rather simple. One simply has to specify a few options related to the traffic that is about to be generated, click the 'Start' button. Clicking 'Start' displays a new line after the current one, that allows to generate a new flow. Once a flow is properly running, its 'status' turns to 'On' and the button on the right becomes 'Stop'. Clicking 'Stop' stops the flows. It actually kills each Iperf and SSH processes individually. Therefore, it is highly recommended to wait for the status on this line to be 'Off' before trying to start a new flow.

The different option for the flow generation are:

- 'source(s)': That represents a set of nodes, that can be written using the same regular expression than for inputing the set of nodes on the main GUI (see Section VI-D). An `iperf` client process will be launched in each of these nodes.
- 'destination': The node that acts as the traffic sink. An `iperf` server process will be launched on this node.
- 'port': The port to use. For the moment, only UDP traffic is implemented. The input for setting TCP instead should be added anytime soon. Entering nothing here will select ports automatically (from 6000 and incrementing for each new flow).
- 'duration (s)': The duration of the traffic in seconds. Nothing defaults to 100,000 seconds.
- 'ToS': The type of service. Can be one of the four values recognized by the MAC layer (BK, BE, VI, VO). Default is BE (the usual Best Effort traffic class).
- 'throughput (Mbit/s)': The desired throughput can be set using a float or integer value.

VII. Acknowledgment

We are grateful to Alaeddine El Fawal for his participation in building this testbed and to Harald Schiberg, Thomas Huehn (TU Berlin) and Jae-Yong Yoo (GIST) for their valuable advices.

References

[1] *Freifunk*. http://freifunk.net/.
[2] *MIT Roofnet*. http://pdos.csail.mit.edu/roofnet/.
[3] *Net Equality*. http://www.netequality.org/.
[4] *OpenWRT firmware*. http://openwrt.org/.
[5] *VPNC client*. http://www.unix-ag.uni-kl.de/ massar/vpnc/.
[6] *WPA-Supplicant*. http://hostap.epitest.fi/wpa_supplicant/.
[7] A. Aziz, A. E. Fawal, J.-Y. L. Boudec, and P. Thiran. Aziala-net: Deploying a scalable multi-hop wireless testbed platform for research purposes. In *Mobihoc S^ 3 2009*, http://icawww1.epfl.ch/aziala/.
[8] A. Aziz, T. Huehn, R. Karrer, and P. Thiran. Model validation through experimental testbed: the fluid flow example. In *Proc. of TridentCom'08*, Mar. 2008.
[9] J. Herzen, A. Aziz, and P. Thiran. Net-controller: a network visualization and management tool. In *Demo at IEEE INFOCOM 2010*, http://icawww1.epfl.ch/NetController/.
[10] R. P. Karrer, P. Zerfos, and N. M. Piratla. Magnets - a next generation access network. In *Poster at IEEE INFOCOM'06*, Barcelona, Spain, Apr. 2006.
[11] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, Aug 2000.
[12] V. Pierobon. Patent wo/1996/008884: Massive array cellular system, 1994.
[13] H. Schiöberg and D. Levin. Multiflowdispatcher and TCPSpeaker. SyClick!, Nov. 2009.

[17] `https://sourceforge.net/projects/iperf/`