Faculty of Economic Sciences, Communication and IT
Department of Computer Science

Jonathan Vestin

CloudMAC: Access Point Virtualization for Processing of WLAN Packets in the Cloud

Scalability to large VAP Deployments

Degree Project of 15 credit points
IT-Design: Programvarudesign

Date/Term:	12-06-05
Supervisor:	Andreas Kassler
Examiner:	Donald F. Ross
Serial Number:	C2012:12

Karlstads universitet 651 88 Karlstad
Tfn 054-700 10 00 Fax 054-700 14 60
Information@kau.se www.kau.se

CloudMAC: Access Point Virtualization for Processing of WLAN Packets in the Cloud

Jonathan Vestin

This thesis is submitted in partial fulfillment of the requirements for the Bachelors degree in Computer Science. All material in this thesis which is not our own work has been identified and no material is included for which a degree has previously been conferred.

Jonathan Vestin

Advisor: Andreas Kassler

Examiner: Donald F. Ross

Abstract

In large organisations, wireless networks can consist of a hundreds or even thousands of wireless access points. The workload to configure these systems can be large due to the variety of configuration interfaces. The type of interface used to configure the access point (AP) varies greatly between manufacturers and models. This causes management of a large network to be complex.

In this thesis, a way to solve this problem was investigated and developed. This was done through an architecture we call CloudMAC. CloudMAC splits the single physical access point into two separate physical machines. One machine runs a Virtual AP who is transparently connected to the another machine, running the Physical AP. The connection is provided by a tunnel over the Ethernet wire. This extension to our current wireless networks allows for easier configuration and management.

In order to test the performance of this new architecture, a prototype was constructed and a series of performance tests were run. The results of the performance tests shows that this new, more flexible architecture gives a minimal loss in performance compared to a standard wireless network.

The applications of this architecture include, but are not limited to: simplifying the administration by centralizing the processing; eases deployment of new applications; making wireless access points shut down or start up depending on network load, thus saving energy.

Acknowledgments

I would like to thank **Peter Dely** for teaching me all I needed to know about linux networking, for helping me with the problems that I had throughout the project and for giving advice on how to write parts of the dissertation.

I would also like to thank **Andreas Kassler** for being my supervisor for this project, helping me correct the final version of the dissertation and being the person who originally inspired me to aim for a career in computer science research.

Contents

List of Figures

List of Tables

List of Code Excerpts

1 Introduction

Wireless networking is a relatively young technology, within computer science, but has seen gradually increasing developments over time [10]. Wireless networks are now used by large organizations and in large areas. Because of this the wireless networks will be expansive, consisting of many access points. The administration and management of these access points could become difficult, due to the sheer size of the area and amount of devices.

There is also the issue of energy saving in wireless networks. In large networks, there are time periods where certain access points are unused. Allowing to dynamically power these devices down and power them up in relation to the usage could provide energy savings. Furthermore, it would be preferable if when a wireless access point is powered off, it will perform a seamless handover, without dropping the connection.

One already available solution to this problem is running one or more virtual machines on each of the access points (APs). These virtual machines perform the processing logic for the access point. All configuration is thus in the virtual machine, not in the physical access point. Copying the virtual machine essentially replicates the functionality of the access point. This method allows easy management and configuration of access points. The drawback of this method is that the resources available at the access point are often scarce. Virtualizing an operating system, which often requires a large amount of resources, can result in performance loss.

The basic idea behind the CloudMAC system is to split the access point into two separate machines, a Physical AP and a Virtual AP. The Physical AP will be a simply device which only forwards packets it receives from the Virtual AP onto the air interface, and also sends any information it receives to the Virtual AP. The Virtual AP will perform all the processing logic and contains the association state of the stations connected to the network.

This allows to create new, very flexible, network applications. This could prove advantageous in various areas, such as power conservation by turning off lightly used / unused access points

dynamically, saving energy. Another area of usage is that multiple entities could potentially deploy their own processing logic, using only a single device.

One possible problem with this approach may be the loss in performance. This is due to that one device is now spread over the network. However, our evaluation shows that there is only a very small performance loss in this new, and very flexible, architecture.

The rest of the thesis is organized as follows: Chapter 2 contains the background of the project and related work. Chapter 3 is a description of the CloudMAC architecture. Chapter 4 is the design and implementation of the system. Chapter 5 is the results of the tests we ran. Chapter 6 presents our final conclusions.

2 Background and Related Work

2.1 Introduction

In this thesis we investigate a way to move the processing logic of the physical AP to a separate machine. This separate machine will run a virtual AP which is transparently connected to the physical AP. This new architecture has many areas of application, such as easy replication of access points, easily changing the processing logic, dynamically expanding and contracting the network etc. With such a system, the possible future developments could for example be:

- Turning access points, with few users, off to conserve energy. This could for example be done through the usage of a OpenFlow Switch, which can monitor the activity and dynamically power off and power on APs as required.
- Allow for complex processing logic with relatively light access points.
- Use a single router for multiple networks with different processing logic. This could help multiple researchers perform experiments on a single device.

2.2 Terminology

Some of the most important terminology to be used in the report is briefly presented here:

SoftMAC: A SoftMAC device is a Wireless Networking Device, where the MLME (MAC Sublayer Management Entity) is managed in software. An example of a driver api using SoftMAC would be mac802_11. [15].

mac80211: A framework which developers can use to develop drivers for SoftMAC devices [15].

VAP: Virtual Access Point. A virtual access point a virtual interface to a physical device. For example in the newer Linux kernels one can have multiple virtual wireless devices all representing the same physical wireless card. This could be used to allow a physical cards to be used in monitoring mode and managed mode simultaneously.

Manager Node: The machine which runs the virtual AP. This machine could manage one more more physical APs.

AP Node: A very simple physical AP that only forwards data from and to the Manager Node.

Station Node: A station connected to the Manager Node through an AP Node.

2.3 Wireless Networking

Wireless networking is a relatively new technology in computer science and has seen many recent developments. Wireless networking enables users to avoid the restriction of cables and use networks in new locations and ways. For example one could browse the Internet while on international flight or one could establish network connectivity in a building where installing a cable infrastructure would be impossible (such as a historical building). [10]

Wireless communication works by sending signals over air instead of cable. This works by using radio waves, which essentially is electromagnetic waves broadcasted by a wireless node. This introduces several problems such as increasing error probability and a burst-like error distribution [12]. There has been a lot of research recently to develop new and better standards, hardware and software to cope with these difficulties [10].

In a wireless network, we have multiple stations that are associated to a single access points. This allows for wireless communication between the stations. In wireless terms, this is called a Basic Service Set (BSS). Wireless Networking have several standards, but the most used by consumers today are wireless `a`, `b`, `g` and `n`. Wireless `b` was the first standard to be

developed and has a maximum data rate of 11 Mbps using the 2.4 GHz band. Later the wireless `a` standard was developed, which featured a increased data rate at 54 Mbps, using OFDM technology at the 5.0 GHz band. [1]

Thereafter, the ODFM technology of wireless `a` and the band of wireless `b` were combined into wireless `g` giving us 54 Mbps at the 2.4 GHz band [1]. Finally we have a new emerging technology using features such as frame aggregation and MIMO in order to increase the data rate. This new wireless technology is called wireless `n`. [13]

In order to associate with an access point, the station node will have to perform a series of requests and replies. The station first picks up a beacon frame from the access point and makes the decision to connect. Thereafter it will send a Probe Request to obtain more information about the access point. The Probe Request contains information such as the SSID of the access point to which the station wants to connect, it also contains the supported rates.

The station later replies to this request by sending a Probe Response. This frame contains further information about the access point. Among this information is the SSID of the access point, the supported rates, capabilities and the channel on which communication should take place.

When the station receives the Probe Response and there are no problems with the access point's capabilities, the station proceeds by sending an Authentication message. This message contains any authentication information required by the wireless access point. If the provided authentication is valid, the access point responds with an Authentication Response message.

When the Authentication Response is received, the station sends an Association Request. The access point replies with an Association Response. Finally the station sends a `NULL Frame` as a final confirmation. The NULL frame has a wide variety of uses, one being to wake up the station if it is 'sleeping' [5]. Figure 1 is an overview of the association process.

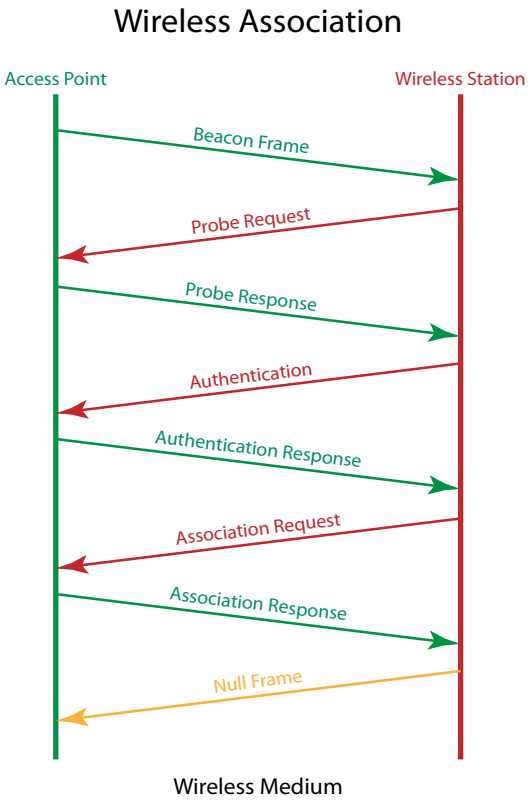


Figure 1: Wireless Association

2.4 Linux Kernel and Wireless Stack

As this report will be dealing extensively with the Linux Kernel and Wireless networking this section will describe it further in detail. The Linux Kernel is the backbone of any Linux system and a cornerstone of open source development. It is a *monolithic kernel*, which means that it places most of its services in the kernel level. This reduces the context switching required at the expense of an increase in the code that executes at the kernel level. The problem with executing code at kernel level is that a error in the code could prove fatal for the whole system, instead of just an isolated part of it. [11]

Linux uses modules to provide the link between devices and software. These modules are dynamically loaded into the kernel by the user. The user can do this by using the `insmod <module>.ko` or `rmmod <module>` commands. The `lsmod` command can be used to list the modules currently residing inside the kernel [11]. There is also the `modprobe` program which allows for even further control of kernel modules [4]. When a module is inserted into the kernel, it is granted the same rights as the Kernel itself. This means, when a module crash, it could cause a complete system failure.

In one end of the Linux Wireless Stack we have the device, which could be a USB or PCI device. The device communicates through an interconnect driver to the hardware drivers. These drivers are specifically programmed for each device. The hardware drivers provide information for the protocol layer. The specific implementation of this layer we will focus on is called mac80211. Applications can control the wireless card through system calls passed to mac80211. This allows the fine-grained control, over wireless packets, we require for the experiment. [14]

2.5 mac80211

mac80211 is, as stated above, a new framework which developers can utilize to write drivers for SoftMAC devices . It allows software to have greater control over the hardware, thus enabling more detail analysis and control over wireless communications. It also allows adding multiple virtual interfaces into a single wireless device. Today most linux wireless drivers are using the SoftMAC framework, instead of the previous FullMAC. SoftMAC supports most of the features provided by the wireless cards today. [18]

For our purposes these properties are very suitable, since our project require a fine control, over wireless network packets, at a low layer.

2.6 mac80211_hwsim

The mac80211_hwsim is a hardware simulator using the mac80211 framework. It can be used to simulate any number of wireless interfaces and it is built to be transparent to the software utilizing it. The wireless network cards created can communicate with each other, as if they were in range. This provided they are on the same channel/frequency. [19]

mac80211_hwsim creates N number of interfaces when inserted into the kernel, where N can be specified by the user. These interfaces are given regular wireless interface names (`wlanX` where X is an unused number). These interfaces can communicate to each other similar to how separate devices communicate. The simulator by default simulates perfect conditions, which means there is no packet loss or corruption.

The simulator also creates a master interface called `hwsimX`, where X is an unused number. This interface is used to listen to any communication that transpires between the simulated wireless interfaces. This interface is a standard Ethernet monitoring interface. This interface will receive any data sent to or between any of the simulated cards. The packets will all contain a radiotap header.

The goal of this tool is to provide a testing ground for new wireless software, without the requirement of hardware devices. However, this is not the purpose we will use it for. Instead we will use it as a virtual access point at the **Manager Node**.

2.7 Radiotap

Radiotap is a standard for 802.11 frame injection and reception. Radiotap is the format of the header appended to all packets received by a wireless interface. This header serves to supply additional information about the various parameters wireless frames can have. The Radiotap header first contains the version and the length of the header. Thereafter a bitmask specifying which fields are present in the header. Finally the data of these fields. [9]

2.8 Userspace Tools

Here is a brief description of the user-space tools we will utilize in our system.

2.8.1 hostapd

`hostapd` is an application for providing wireless access point logic and authentication for a Linux system. It has strong support for the mac80211 framework. `hostapd` works by putting a specified interface into master mode (Access Point Mode) and broadcast beacons onto it. It also creates an additional interface for handling any probe and/or authentication requests received from stations. [16]

`hostapd` starts by repeatedly sending wireless beacons onto its master interface. This interface is specified at the launch of the application. `hostapd` will also monitor for any incoming packets at its secondary interface, the monitoring interface. This interface is named `mon.X` where X is the name of the master interface. For example if the master interface is named `wlan0` the corresponding `hostapd` monitoring interface will be named `mon.wlan0`. When it receives a wireless management frame on this interface, it will reply accordingly. For example, should `hostapd` receive a *Probe Request*, it will respond with a *Probe Response*.

2.8.2 Capsulator

`capsulator` is a utility for tunneling data, from one or more interfaces on a machine, to one or more interfaces on a different machine. Essentially, it transparently connects any number of networks. For example, it can take wireless data received on a monitor interface and send this through the Ethernet wire to a different node, which in turn can replay the received data. You will require a `capsulator` at both ends, one retrieving the data from the wireless device and sending it onto the wire and one receiving it and replaying it.

2.8.3 iw

`iw` is a tool for managing and configuring wireless devices and an alternative for `iwconfig`. It can also be used to create multiple virtual interfaces on a single device [17]. For example creating a new interface in iw works as follows:

```
iw phy phy0 interface add mon type monitor
```

This will add a virtual monitoring called `mon` interface to the physical card `phy0`.

```
iw dev mon del
```

Will delete the same interface. We can also view detailed information about interfaces:

```
iw dev wlan1 info
```

`iw` can also be used to easily connect a network:

```
iw wlan0 connect CLOUDMAC
```

This example connects the wireless device wlan0 to any network with the SSID `CLOUDMAC`.

2.8.4 iperf

Iperf is a tool used for measuring performance in a network using the TCP or UDP protocol. More detailed information can be found in the Iperf Man Page [3].

2.8.5 mgen

Mgen is a tool for generating UDP packets and receiving them. It is used for accurately measuring the UDP performance. More information can be found in the MGEN User's and Reference Guide Version [8].

2.9 OpenWrt

OpenWrt is a very small and extensible GNU/Linux distribution specifically made for embedded devices. OpenWrt is very dynamic in that instead of providing a simple static firmware, it gives us a fully functional filesystem and a system for package management. It is excellent for developers because it is completely free and open-source, which means that any developer could examine the source code without restrictions, thus making it an excellent candidate to use in our system (where viewing and modifying the source code will be of key importance). [9]

2.10 Related Work

2.10.1 Virtual Machine

A Framework of Better Deployment for WLAN Access Point using Virtualization Technique [6] describes a way of creating virtual access points using Virtualization Techniques. The method described in the report involves deploying a Virtual Machine on all wireless access points in the network. Switching an access points logic is done through a migration process, which involves copying a new virtual machine image, the device memory and forwarding plane, to a different access point. When these three entities are copied, they are removed from the original access point.

These nodes are to be managed by a 'manager' node, which controls the system and continuously attempts to determine a better form of deployment. It also manages the migration process between the access points.

The solution which is specified in [6], is supposed to solve issues regarding deployment of access points, enabling administrators to easily reconfigure them on the fly. The approach they are using provides a way for access points to be fully programmable, at the cost of performance. One of the problems, with this approach, is a high complexity at the access

point, which may in turn cause a degradation in performance. Especially since they rarely have access to the resources required to virtualize an operating system. The architecture of the access points can also be a problem, since most access points run on an ARM architecture.

Virtual Wireless Network Urbanization [2] suggests a similar approach but focuses more on actual measurements of the network.

2.10.2 CAPWAP

Another related architecture is the CAPWAP architecture. CAPWAP enables a single Access Controller to manage multiple wireless access points. This architecture splits the processing of MAC layer frames between the access point and the Access Controller. It also introduces a new protocol for communication between the Access Controller and the access points. [7]

In CloudMAC we, instead of splitting the MAC layer processing between access points and virtual AP, put all processing in the virtual AP. This allows us to put the whole association state in the virtual AP rather than splitting it between physical and virtual AP.

2.11 Summary

This section presented a slight introduction to the Linux kernel, how it manages devices and what the wireless stack looks like. We briefly described the mac80211 wireless framework and how it can be useful in this project. We have also looked at the `mac80211_hwsim`, which is a driver for simulating a wireless network. Furthermore, we have also looked at the different tools the system can use, such as `hostapd` for access control and beacon generation, `capsulator` for creating tunnels between nodes and `iw` for managing the wireless network cards. We also looked on related work such as a virtualizing multiple operating systems in a single physical access point and we also looked at CAPWAP.

3 CloudMAC Architecture

The overall goal with CloudMAC is to separate the physical AP into two separate components; A virtual AP and a physical AP. The virtual interface should reside on a different machine and be transparently connected to the physical AP wireless interface.

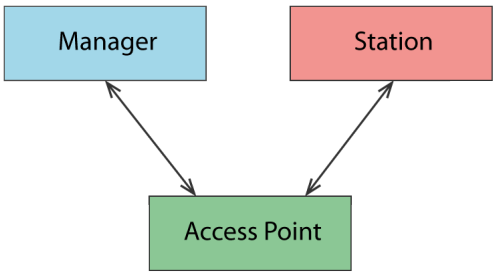


Figure 2: Basic overview of the system

The CloudMAC system consists of three essential core parts. The Manager, the Access Point and the Station. The Manager has one virtual machine running, which contains the actual processing and networking logic. The Manager is connected to the wired network, through which it communicates with the Access Point. The Station, on the other hand, is simply a wireless station, accessing the system. It could be for example a laptop, a cellphone or a desktop. The Access Point relays the communication between the Manager and Station nodes.

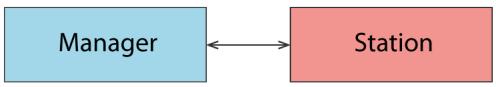


Figure 3: System as seen from the station

The relaying of data should be completely invisible to the Station. From its view, the system looks something like figure 3. The Access Point provides this kind of obfuscation due to its simplicity. The only thing the Access Point does, is forwarding data from the station to the

manager and data from the manager to the station. The data is forwarded through a tunnel in the wired network, to which both the Manager and the Access Point are connected.

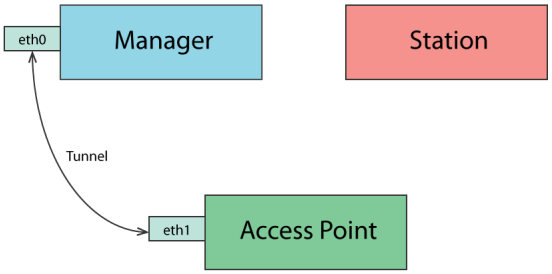
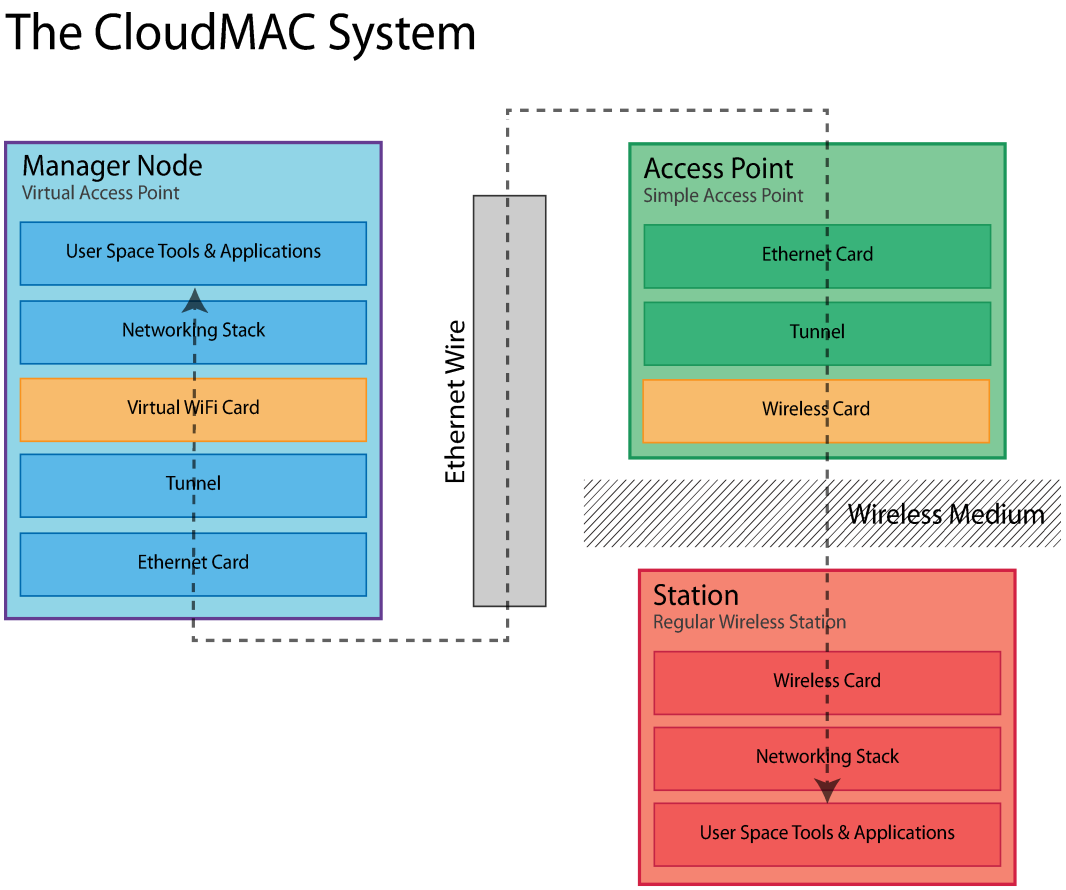


Figure 4: Ethernet tunnel between access point and manager

If we break the Manager and Access point apart into layers, we can show the flow of packets through the system. The User Space Tools & Applications layer is where any tools would reside. This could, for example, be network configuration tools like `iw` or network authentication tools like `hostapd`. These tools will send data through the Networking Stack to the Virtual WiFi Card of the Manager. This card in turn sends the data through the Ethernet wire using a tunnel. The access point will receive the data through the other end-point of the tunnel and send it into the air.

Any data received by the access point will be sent through the same tunnel. The manager will receive the data at its tunnel end-point, moving it up the networking stack to the user-space application. A normal wireless system would simply perform all the processing in the access point. The difference between CloudMAC and a normal wireless system can be seen in figure 5.



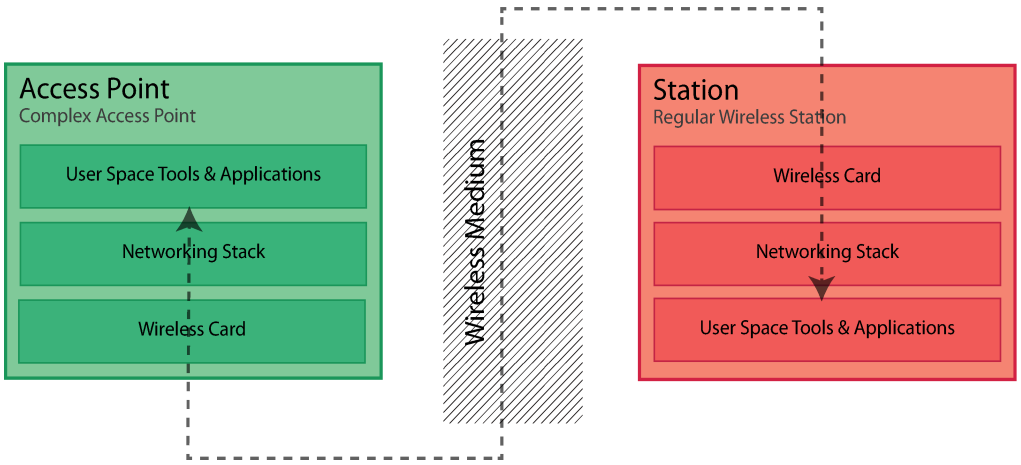


Figure 5: CloudMAC architecture compared to a regular wireless network

4 Design and Implementation of CloudMAC

4.1 Introduction

In this chapter the implementation of the CloudMAC architecture will be described. First a description of the different components in the CloudMAC architecture and their respective implementations, then an overview of the system as a whole.

4.2 Manager

The Manager node is a virtual machine which could be running at any system using any architecture. It is not in any way dependent on the architecture of the Access Point. The system uses a modified version (described in detail later in this chapter) of the mac80211_hwsim driver provided with the Linux Kernel (v2.6.27 and above). This driver creates two virtual interfaces called `wlan0` and `wlan1`. These two interfaces are connected through a virtual air interface and data can be sent between them. However, this is not the purpose we will use them for, rather we will utilize one of them as a virtual access point.

The hardware simulator also provides another important interface, called `hwsim0`. This interface replays any communication sent onto the simulated wireless interfaces (`wlan0` and `wlan1`). We will use this interface because we need some way to relay the communication on the simulated interfaces to the Access Point. This interface is not by itself a wireless interface, but instead a regular Ethernet interface.

Futhermore, we will later have another interface called `mon.wlan0`. This is an interface created by `hostapd,` which it uses to receive and transmit wireless management frames [16]. Finally the last interface would be the Ethernet interface `eth0`, connected to the same network as the Access Point. This interface will be used to tunnel packets we receive on the `hwsim0` interface and handle incoming packets. You can see a graphical representation of the whole Manager node in figure 6.

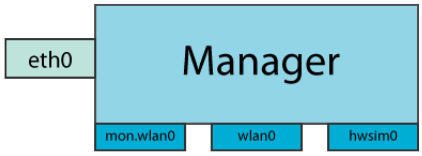


Figure 6: Overview of the manager

We will have a `hostapd` (Wireless Authenticator) running on the `wlan0` interface, which will generate wireless beacons and manage station association. `hostapd` will in turn create the `mon.wlan0` interface. `mon.wlan0` is the interface to which `hostapd` will listen for any incoming management frames on. `hostapd` will start sending beacon frames on the `wlan0` interface. These beacons will in turn also be received on the `hwsim0` interface.

We will use a modified version of the `capsulator` called `capsulemod`, to provide a tunnel from the `hwsim0` interface to the Ethernet interface `eth0`. Any packet received on the `hwsim0` interface is encapsulated in a tunnel frame, which adds a tag to the packet. This tag is used to make sure only the right tunnel endpoints accept the packet. The packet is tunnelled to the Access Point, as you can see in figure 7. What happens to the packet afterwards will be described in the Access Point section.

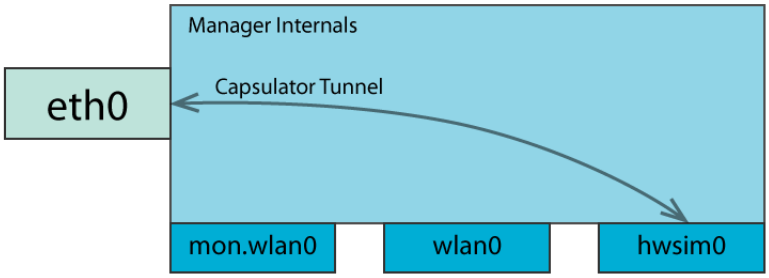


Figure 7: Packet capsulation

Receiving information is a bit more complex, since the tunnel will put any packets coming through `eth0` from the Access Point into the `hwsim0` interface. Since `hostapd` uses its own interface for monitoring incoming packets it cannot detect any packets coming through the tunnel. Thus we need to alter the behaviour of the `mac802_hwsim`, to allow for this. We chose to make any packet received on the `hwsim0` interface to be replayed to all regular interfaces (`wlan0` and `wlan1`). This will cause the `hostapd` to hear any packet received from the tunnel. Figure 8 is a graphical model of how the internals of the manager work.

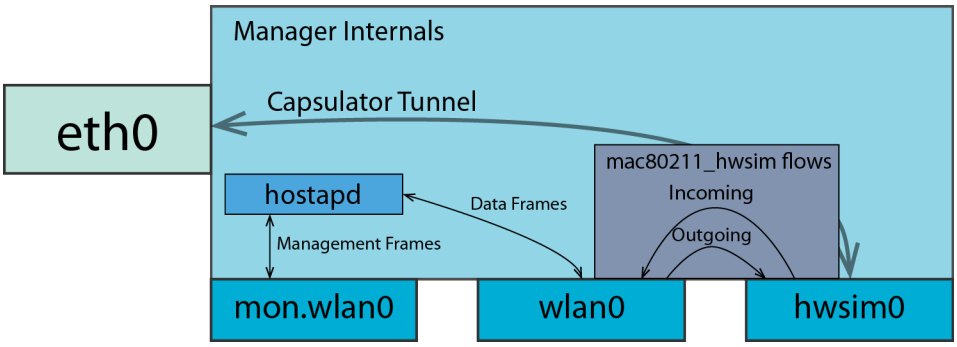


Figure 8: Manager internals

4.2.1 Detailed mac80211_hwsim Changes

The `mac80211_hwsim` is a driver for simulating wireless networks. This driver is used to provide a fake wireless environment for `hostapd` to work in. However in order to do this we need to make a small change to the driver. We need a way of allowing all information sent to the `hwsim0` interface to be replayed on all the simulated wireless cards. The function we need to change is called `hwsim_mon_xmit`. This function is responsible for any information transmitted on the `hwsim0` interface. In other words, any information coming from the tunnel will be processed by this function.

In the default driver, the function does nothing more, than simply freeing the packet data. We need to make it send the data to the wireless cards before this happens. Since the data received on monitoring interfaces carry a radiotap header, so will any data received on this

card. Frames with a radiotap header cannot be directly injected into the kernel, so the header needs to be stripped from the packet.

The packet received is enclosed within a Linux `skb` (socket buffer). The socket buffer provides functions for contracting and expanding the data, which we need to use in order to remove the radiotap header. The driver was modified to read the radiotap header length from the `skb->data` pointer. Later this information is used to remove as many starting bytes as we require from the packet using `skb_pull` function

After this, the function will reset the socket buffer. This is done exactly the same way as it is in the other receive functions (for example `mac80211_hwsim_tx_frame`). Thereafter it will trim the last 4 bytes, because those 4 are scrap bytes left from the kernel. This is done using the `skb_trim` method.

Subsequently, the packet is ready for replaying on the wireless interfaces. Before this is done, the spinlock must be activated, which essentially is a mutex lock (semaphore) preventing other multiple processes from simultaneously using the network card. Now the driver iterates over all the simulated network cards in the list called `hwsim_radios`. This is done using a macro called `list_for_each_entry`, which creates a loop over the list.

For each entry in the list, the driver checks that the radio is not idle and not started, it also checks to see if the channel is set. If any of these parameters are wrong, the driver continues to the next card. After this check, the socket buffer is copied using the `skb_copy` function. The driver also makes sure the command was successful, if not it continues to the next card. Thereafter, the duplicated packet is passed to the kernel using `ieee80211_rx_irqsafe`.

Finally the driver ends the loop and deactivates the spinlock to enable usage of the network radios. It also frees the memory allocated for the socket buffer as it is no longer required.

```
skb_pull(skb, radiotap_len);
skb_orphan(skb);
skb_dst_drop(skb);
skb->mark = 0;
secpath_reset(skb);
nf_reset(skb);
skb_trim(skb, skb->len - 4);
spin_lock(&hwsim_radio_lock);
list_for_each_entry(nic, &hwsim_radios, list) {
        struct skb_buff *nskb;
        if(nic->idle || !nic->started ||
                !nic->channel)
            continue;

        nskb = skb_copy(skb, GFP_ATOMIC);
        if(nskb == NULL)
            continue;

        ieee80211_rx_irqsafe(nic->hw, nskb);
}
spin_unlock(&hwsim_radio_lock);
dev_kfree_skb(skb);
```

Code 1: Hardware Simulator Changes

4.2.2 Manager Setup

Here we will briefly describe the process of setting up the manager and show the commands used to do it. These commands are entered using the linux shell (commandline interface). First we remove the old mac80211_hwsim module and insert our custom hardware simulator:

```
rmmod mac80211_hwsim
insmod mac80211_hwsim.ko
```

Code 2: Insert custom module

After this, we bring up the hardware simulator monitoring interface and set the IP address of the `wlan0` interface to a suitable one (192.168.1.1 in this example). Then we launch the `hostapd` program to listen on the wlan3 interface.

```
ifconfig hwsim0 up
ifconfig wlan0 192.168.1.1
../hostapd hostapd.conf -B
```

Code 3: Setup interface and launch `hostapd`

The `hostapd` configuration contains the code in code block 4.2.2. The first line specifies which driver we use, which is the nl80211 driver (specific for the machine we are running the system on). Thereafter the mode `a` and channel `56` is specified. The reason for using this mode and channel is to not interfere with any other transmissions in the building. Later the desired interface is specified, which in our case is the simulated interface `wlan0`. After that, the ssid of the network is specified, which in this case is `CLOUDMAC`. Finally the basic and supported rates are specified. These are set to 54 mbps, which is the highest rate available in wireless mode `a`.

```
driver=nl80211
hw_mode=a
channel=56
interface=wlan0
ssid=CLOUDMAC
supported_rates=540
basic_rates=540
```

Code 4: Manager hostapd configuration file

Finally, we need to set up the capsulator tunnel between the Manager and the Access Point. We do this using the `capsulator` command:

```
../capsulemod —t eth0 —f <Access Point IP> —b hwsim0#1 ——mode 2
```

Code 5: Starting the capsulator

The -t parameter is the target interface, -b is the source interface. The -f parameter is the destination IP address for the tunneled packet. -v enables verbose output (detailed output messages for debugging). –mode 2 is a custom parameter, which sets the forwarding mode. This prevents packets from being duplicated and is explained further in the Access Point section, where it is more applicable.

After this is done, the Manager will wait for a tunnel endpoint to be created at the Access Point.

4.3 Access Point

The Access Point is simpler than the Manager, but not less important. It has three interfaces of importance. The first one is the Ethernet interface `eth1`, which is connected to the same network as the Manager.

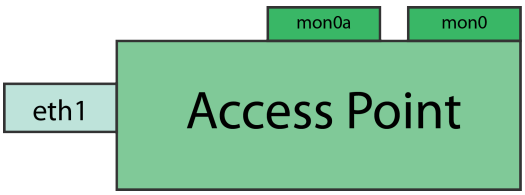


Figure 9: Access point interfaces

The next interface is `mon0`. This interface is simply a virtual monitoring interface (virtual interface in monitor mode) to the physical `wlan0` card. The reason we use a monitoring interface, is because we use the ath5k functionality, so that any information sent to a virtual monitoring interface, is sent onto the air from the respective physical card. This is important because it enables us to send information received from the tunnel endpoint onto the wireless medium. Another reason for putting it in monitoring mode is because we want to process the entire wireless frame, including the MAC layer information.

The third interface is mon0a which is used to reply with ACK packets to the station. This is required because without timely acknowledgments, the station will keep resending the packet, creating unnecessary traffic.

As can be seen in figure 10, the messages coming in from the air medium is simultaneously replied to by the `mon0a` interface, while being tagged and patched through the capsulator tunnel towards the Manager node.

4.3.1 Detailed capsulator changes

The standard capsulator works well with the Manager, but with the Access Point there will be a problem. Packets that are sent into the air, are going to be received by the same

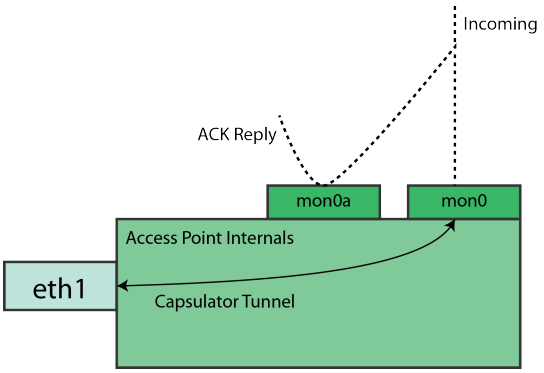


Figure 10: Access point internals

monitor interface they are sent from, which causes a feedback loop. This creates a duplicate for each packet sent by the Manager to be returned through the same path. This can be seen visualized in Figure 11

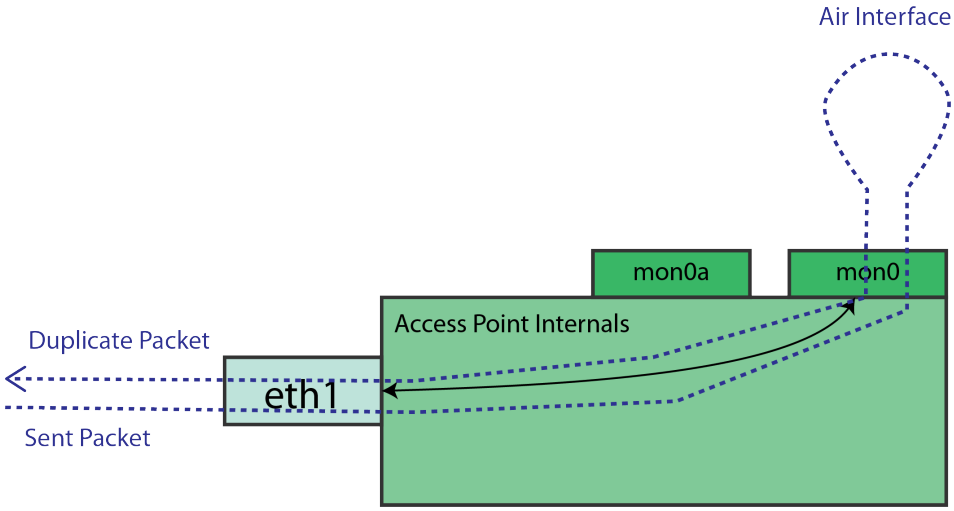


Figure 11: Access point capsulator problem

To solve this, we added the possibility of selecting a mode when running the capsulator. The mode can be either 0, 1 or 2. Mode 0 represents the ordinary capsulator. Mode 1 drops all packets (from the air interface), where the Source MAC address ends with `00:02`. mode 2 accepts only packets, where the Source MAC address ends with `00:02`. This address is the address for the Manager `hwsim0` card. This will also be the source address of any packet leaving the Manager node, which makes it an excellent way to check if a packet it destined

to the manager or not. The access point thus runs with the mode 1 option:

```
../capsulemod -t eth1 -f <Manager IP> -b mon0#1 --mode 1
```

Code 6: Capsulator mode example

This effectively prevents the feedback loop. In our implementation the MAC address is hard coded, but the capsulator could easily be modified to support a modifiable address. The code is only changed in the `capsulator.c` at the `capsulator_thread_main_for_border_port` function:

```
unsigned short *radiotap_start = (buf + 4 + 2); //Find radiotap header start
unsigned short radiotap_len = *radiotap_start;
radiotap_len = (radiotap_len>>8)|(radiotap_len<<8); //Find radiotap length
void * frame_start = (buf + 4 + radiotap_len); //Find Frame start
char * source_addr = (frame_start + 4 + 6); //Find source address
char * sa = source_addr; //Create shorthand variable
```

Code 7: Determine capsulated packet source MAC address

Now that we have the source address, we can simply check the mode and source address, and relay the packet accordingly.

```
if(sa[0]==2&&sa[1]==0)
{
    if(mode == 1)
    {
        verbose_println("CLOUDMAC: Ignoring Replayed Packet == 00:02");
        continue;
    }
}else
{
    if(mode == 2)
    {
        verbose_println("CLOUDMAC: Ignoring Relayed Packet != 00:02");
        continue;
    }
}
```

Code 8: Determine if to drop or tunnel packet

As can be seen in figure 12, the packets are dropped before they can enter a feedback loop.

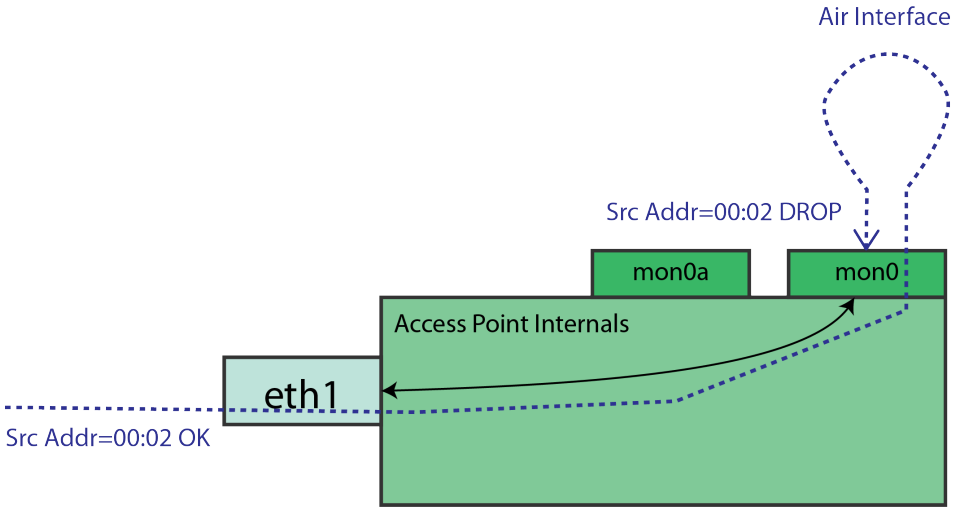


Figure 12: Access point capsulator problem solution

4.3.2 Access Point Setup

The access point setup is relatively simple and there are two scripts at work here. The first one is `apinit` which initializes all the interfaces on the Access Point. The first things it does is running a reload script to reload the network driver:

```
../reload.sh
```

Code 9: Run module reload script on access point

`reload.sh` removes all relevant modules and reloads them again. It also replaces the built-in ath and ath5k driver, with a custom built one. This is to enable debugging and modifying the transmit power (`txpower`). After reloading the modules, our `apinit` script removes unused interfaces:

```
iw dev wlan1 del
iw dev wlan2 del
```

Code 10: Delete unused interfaces from access point

This reduces the cluttering during testing and has no functional value. Thereafter it sets the channel of the physical card and creates a virtual monitoring card. Finally it will set the transmit power to 20dbm in order to increase the signal quality:

```
ifconfig wlan0 up
iwconfig wlan0 channel 56
ifconfig wlan0 down
iw phy phy0 interface add mon0 type monitor
iwconfig mon0 channel 56
ifconfig mon0 up
iwconfig mon0 txpower 20dbm
```

Code 11: Setup access point interfaces

Finally, the script creates the `mon0a` interface which, is set into `__ap` mode with a MAC address matching the one of the Manager:

```
iw phy phy0 interface add mon0a type __ap
ip link set dev mon0a address 02:00:00:00:00:00
ifconfig mon0a up
```

Code 12: Add ___ap interface to the access point

This causes the interface to reply, with a timely ACK, to any incoming packets, since the delayed ACK from the manager is too slow.

When this script has finished executing, the next script is started. This script is called `aprun` and has only one line of code. It starts the capsulator just as we have seen above:

```
../capsulemod –t eth1 –f <Manager IP> –b mon0#1 ––mode 1
```

Code 13: Start the capsulator on the access point

Now the Access Point is initialized and ready to receive data.

4.4 Station

The Station could be any wireless station with access to a 802.11a wireless network card. In our experiment, we will use one of the embedded systems available. The setup is a simple script:

This script basically enables a wireless card, sets the IP address to `192.168.1.2` and the essid to `CLOUDMAC`. Figure 13 shows us a graphical representation of the Station. Unremarkably, it

```
ifconfig wlan0 up
iwconfig wlan0 channel 56
ifconfig wlan0 192.168.1.2
iwconfig wlan0 txpower 20dbm
iwconfig wlan0 essid "CLOUDMAC"
```

Code 14: Setup the station interface

has no inner workings that are part of the CloudMAC system. Because of this, any Station with a wireless card should be able to connect. We tested and were able to successfully connect on both a MAC device and a Linux device.

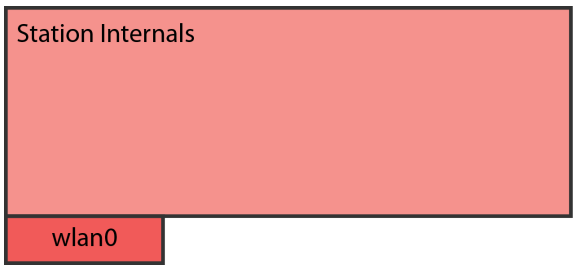


Figure 13: Station node

4.5 The Whole System Together

Now, when we have described the individual components of the system, let us take a look at the whole system combined. Previously, in figure 2 we provided a simple graphical representation of the system. Here we will provide a more detailed one. Thereafter we will show some figures of different packet flows. A picture of the entire CloudMAC system can be seen in figure 14.

4.6 Summary

We have had a look at the system as a whole. How the data travels through the access point in both directions and how the data is later processed by the manager. We have also seen detailed information about how the driver and capsulator was altered and what special

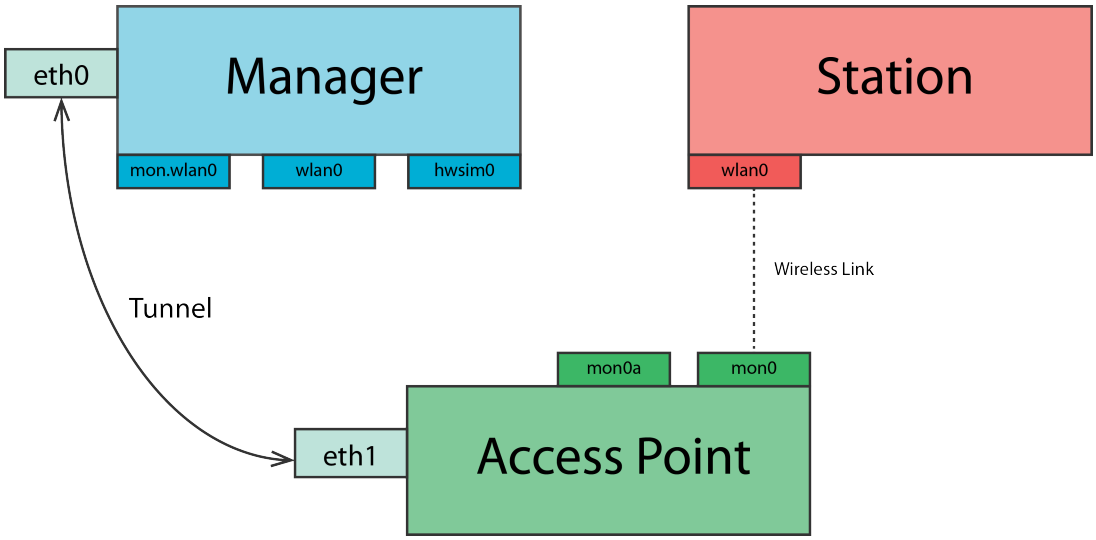


Figure 14: CloudMAC system overview

configuration options for setting up the CloudMAC system.

This new system have an increased flexibility at the cost of an extended packet path, with additional intermediate nodes. This could potentially increase the roundtrip time and throughput of the network.

There are a few issues with the current implementation. The first is that the tunnels are running in user space, which increases the processing time due to context switching. This could potentially be remedied by moving the tunnel to kernel space. Another existing issue is the hard coding of the MAC address. In a final system it would be preferable with an option to specify which mac addresses the Manager nodes use.

5 Evaluation

5.1 Introduction

This chapter will investigate the performance losses of the CloudMAC system as compared to a normal wireless system. This chapter is separated into multiple parts. The first chapter is a look how the machines are set up. Later we will look in detail at the specifications of all the machines used in the tests, as this greatly affects test results. There will also be descriptions of how the various test scripts look like, for validation of the results.

5.2 Machine Setup

5.2.1 CloudMAC Network Setup

The CloudMAC network setup is described in detail above. Here we will include some additional information on how the network is set up. The access point and station are both embedded machines, as described in the above section. These are connected through a wireless interface. The access point and the manager are connected by wire through a switch. This is depicted in figure 15.

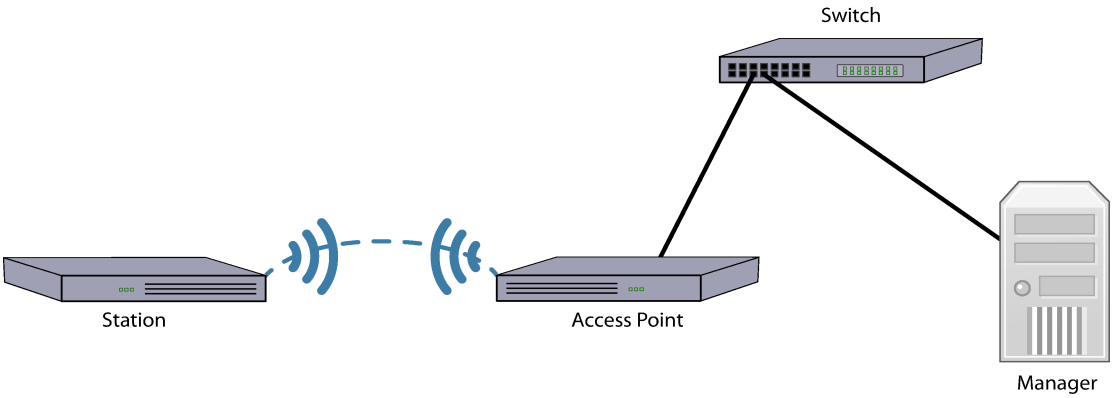


Figure 15: CloudMAC network

5.2.2 Reference Network Setup

The reference network is set up by having only two nodes, an access point and a station. The access point is set up with properties similar to the CloudMAC network. In the code excerpts below are the setup scripts for the reference network.

```
driver=nl80211
hw_mode=a
channel=56
interface=wlan0
ssid=CLOUDMACR
supported_rates=540
basic_rates=540
```

Code 15: Reference network hostapd configuration file

```
../reload.sh
ifconfig wlan0 up
ifconfig wlan0 192.168.1.1
nohup hostapd hostapd.conf -B
ifconfig wlan0 192.168.1.1
```

Code 16: Reference network access point setup script

```
../reload.sh
ifconfig wlan0 up
ifconfig wlan0 192.168.1.2
iwconfig wlan0 essid CLOUDMACR
```

Code 17: Reference network station setup script

Figure 16 provides a graphical representation of the reference network.

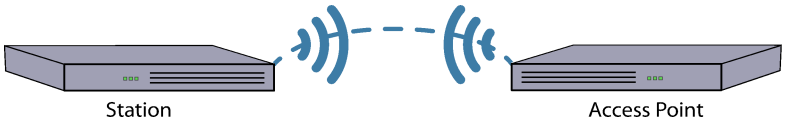


Figure 16: Reference network

5.2.3 Test Scripts

Here we will describe the test scripts to a small degree. We chose not to include the full scripts due to their size. The testing scripts use `expect` to set up the network and then executes the tests in sequence. `expect` is a tool for automating commandline entries. First the throughput tests, then the two-way delay tests. The association time was tested at a different time (but under similar conditions). All tests were run at night when the interference from surrounding network would be as small as possible. This is also the main reason for using the 5Ghz band, which assures that there is no interference from surrounding wireless traffic.

The network uses channel 56 at a sending power of 20db.

5.3 Machine Specifications

We will be measuring the performance of the CloudMAC network using a small test network. This network consists of nodes and a single virtual machine. The nodes are connected to a switch using Fast Ethernet. The virtual machine is connected to the same switch using Gigabit Ethernet. Here are detailed specifications of the machines:

Type	Node	Virtual Machine
Processor	Intel® IPX435 XScale® 667MHz	N/A
Memory	128 Mbyte DDRII-400 SDRAM	1024 MB

Table 1: Machine specifications

5.4 Results

5.4.1 TCP Throughput

Figure 17 shows the results of the TCP throughput test. This test was performed using `iperf` using the manager node as client and station node as server. The reason for this is

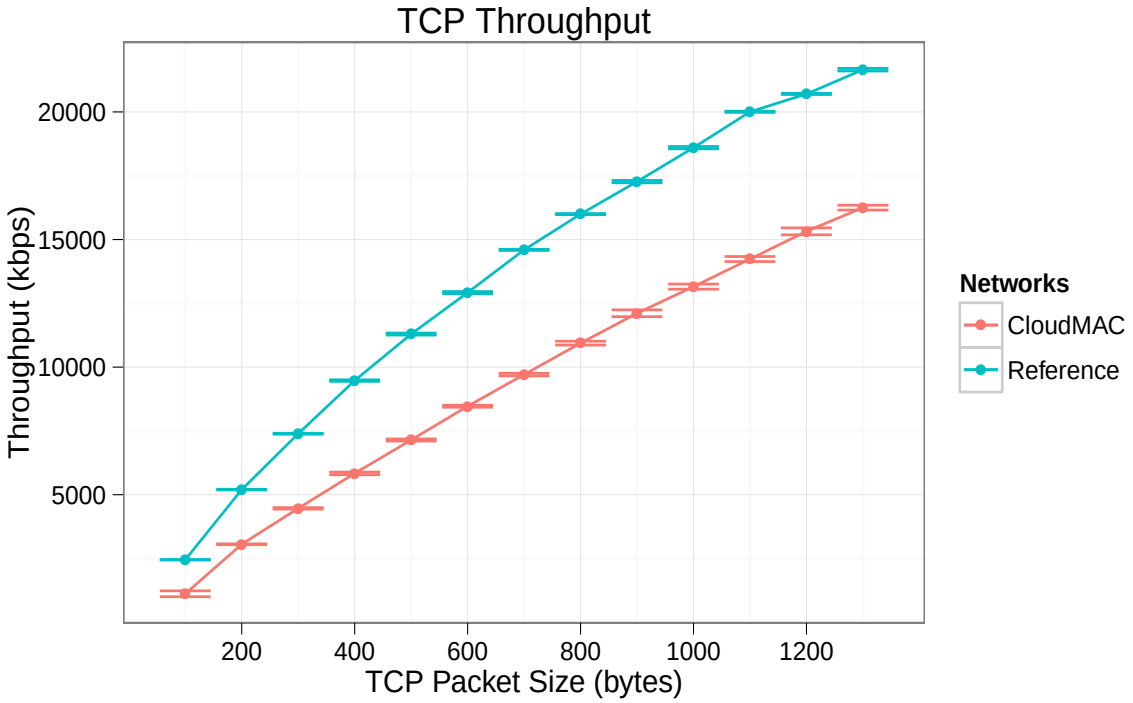


Figure 17: TCP throughput

that `iperf` sends data in the reverse direction, in other words, from the client to the server. Thus, data flows from the Manager to the Station. The test was run using different TCP packet sizes (changed with the `-M iperf` flag) for 60 seconds every 100 bytes of packet size. The test was run 20 consecutive times and the graph shows the average of the 20 results. The reason for the maximum packet size of 1300 is to prevent packet fragmentation. The CloudMAC system currently cannot handle packet fragmentation but such a feature could be developed in the future.

The dots in the graph represent the mean values from the results. The bars show the standard deviation.

As we can see in the graph produced above, the results show some performance loss. This is expected as the network have additional complexity in terms of increased network path and packet processing. The loss at the final packet size is $1 - \frac{16580}{21655} = 0.234 = 23.4\%$. We can

also see that the loss is greater for smaller packets as compared to larger packets. This is likely to be because a smaller packet size means that more packets needs to be transfered, in order to transfer the same amount of data. This also increases the number of ACK packets, which also decreases the throughput.

5.4.2 UDP Throughput

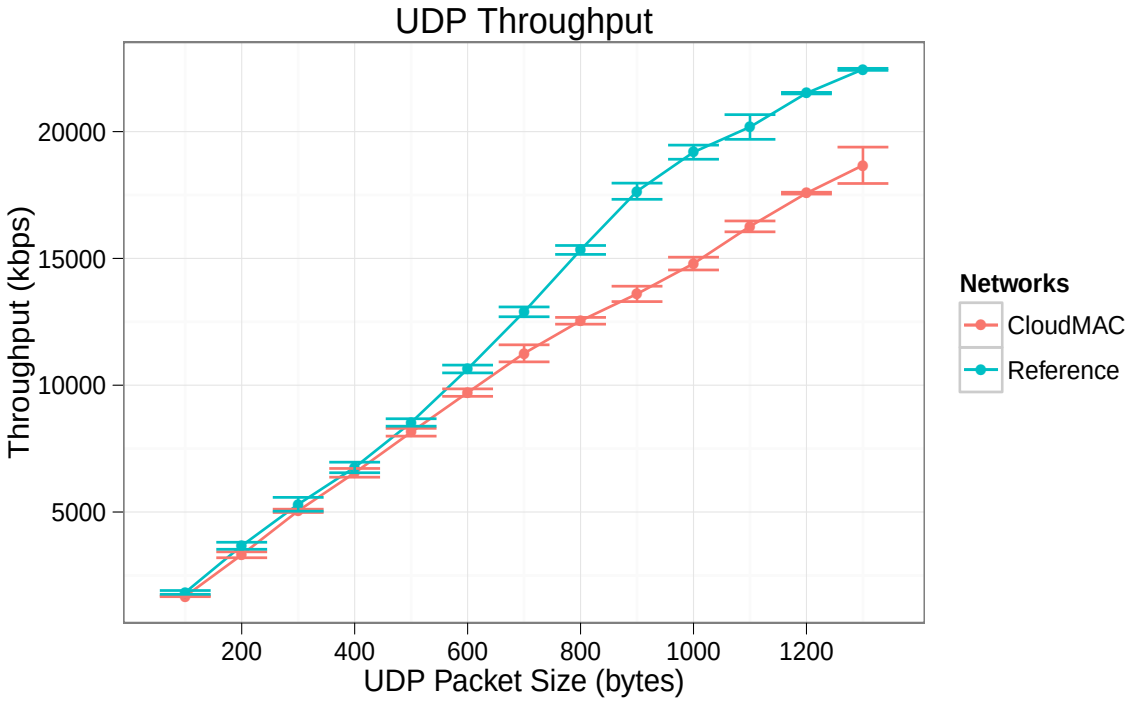


Figure 18: UDP throughput

Figure 18 shows the results of the UDP throughput test. This test was performed using `mgen` using the manager node for generating packets and the station node for packet reception. The test was run 10 seconds every 100 bytes of packet size. The test was run 30 consecutive times and the graph shows the average of the 30 results.

Like the TCP graph, the dots in the graph represent the mean values from the results. The bars show the standard deviation.

In these results we see some difference between the performance in CloudMAC as compared to the the Reference network. This is especially prominent for higher packet sizes. The performance loss at the final packet size is around $1 - \frac{18671.4}{22454.43} = 0.168 = 16.8\%$. A reason for better performance as compared to TCP is that UDP has no congestion control, thus packets are sent without regard for the network. There are also no ACK packets sent when using UDP, which could be the reason for good performance at lower packet sizes.

5.4.3 Throughput Variance

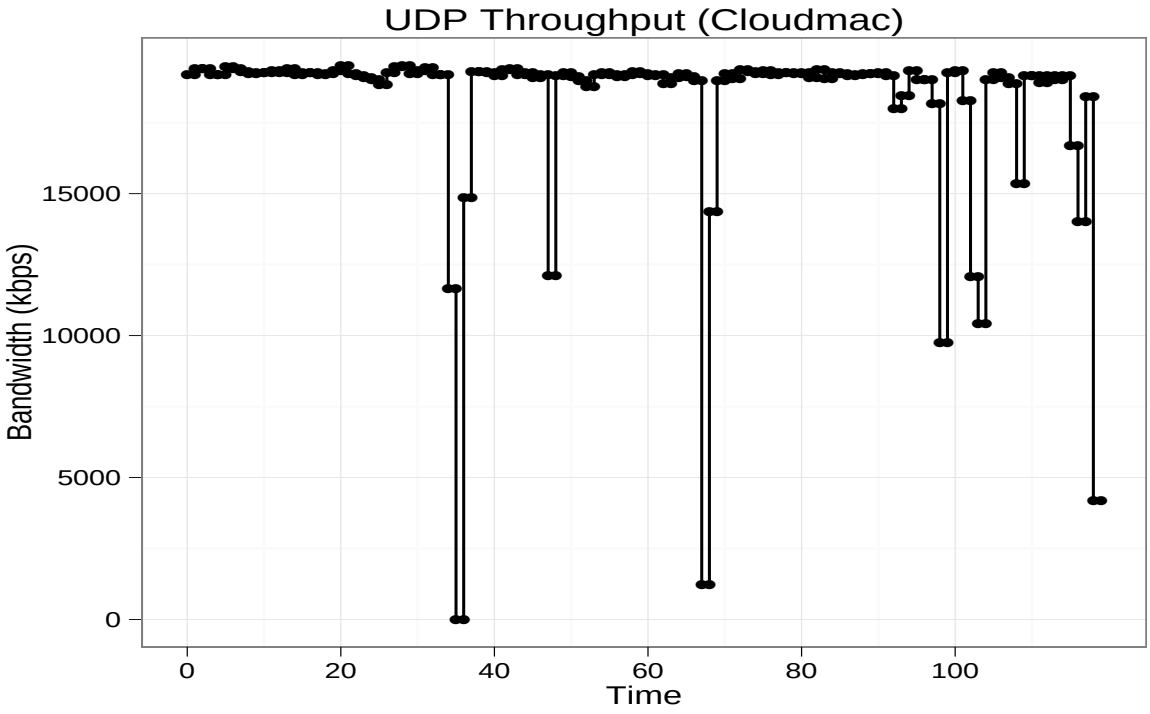


Figure 19: CloudMAC throughput variance

Figure 19 shows the results of the UDP bandwidth variance test on the CloudMAC system. This result was achieved by running `mgen` like the *UDP Throughput* test, but for 120 seconds to achieve higher resolution.

Figure 20 shows the results of the UDP bandwidth variance test on the reference system.

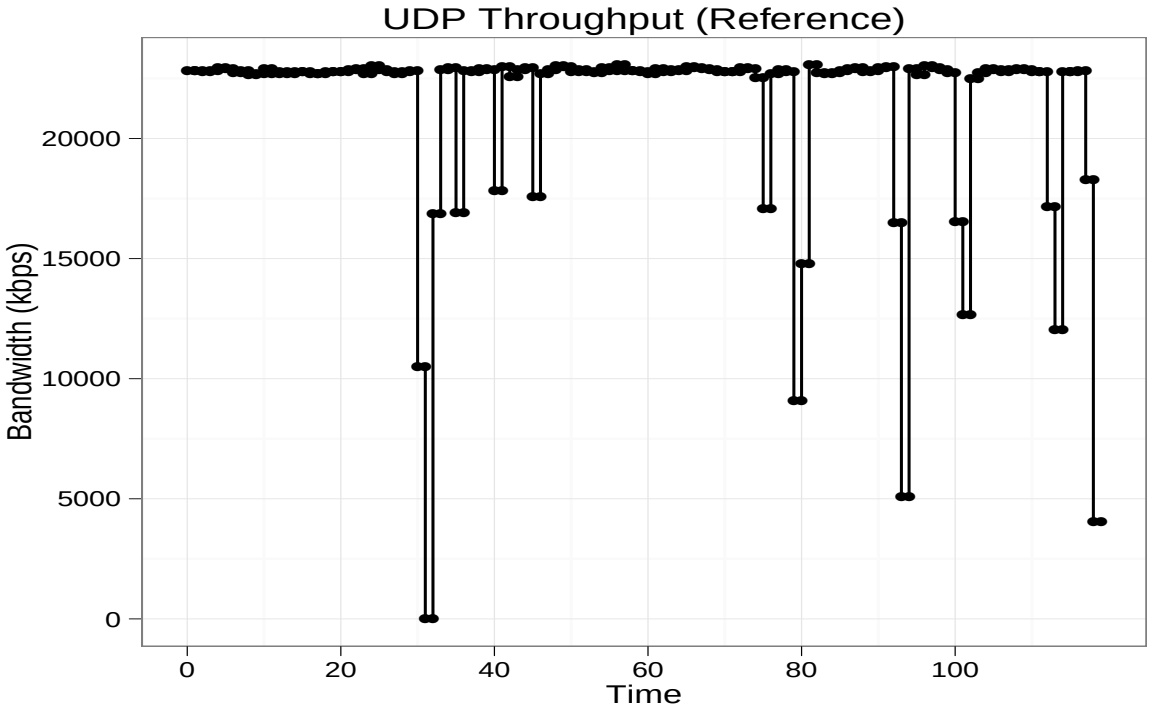


Figure 20: Reference throughput variance

As we can see comparing these two graphs, in both cases we have similar amount of performance drops, which could have several causes. One of these could be that some interference from the outside disrupts sending for a short time, causing slower throughput.

5.4.4 Two-Way Delay

Figure 21 shows the results of the two-way delay test. This test was run using the `ping` utility to ping the manager node from the station. The delays were measured for a five minute period and the data is displayed using a empirical distribution function. This is to easily show how the roundtrip variance differs between the CloudMAC and the reference system.

As we can see in figure 21 CloudMAC leads to increased delay. This is due to that the data needs to be sent through the Ethernet wire, back and fourth between the Manager node.

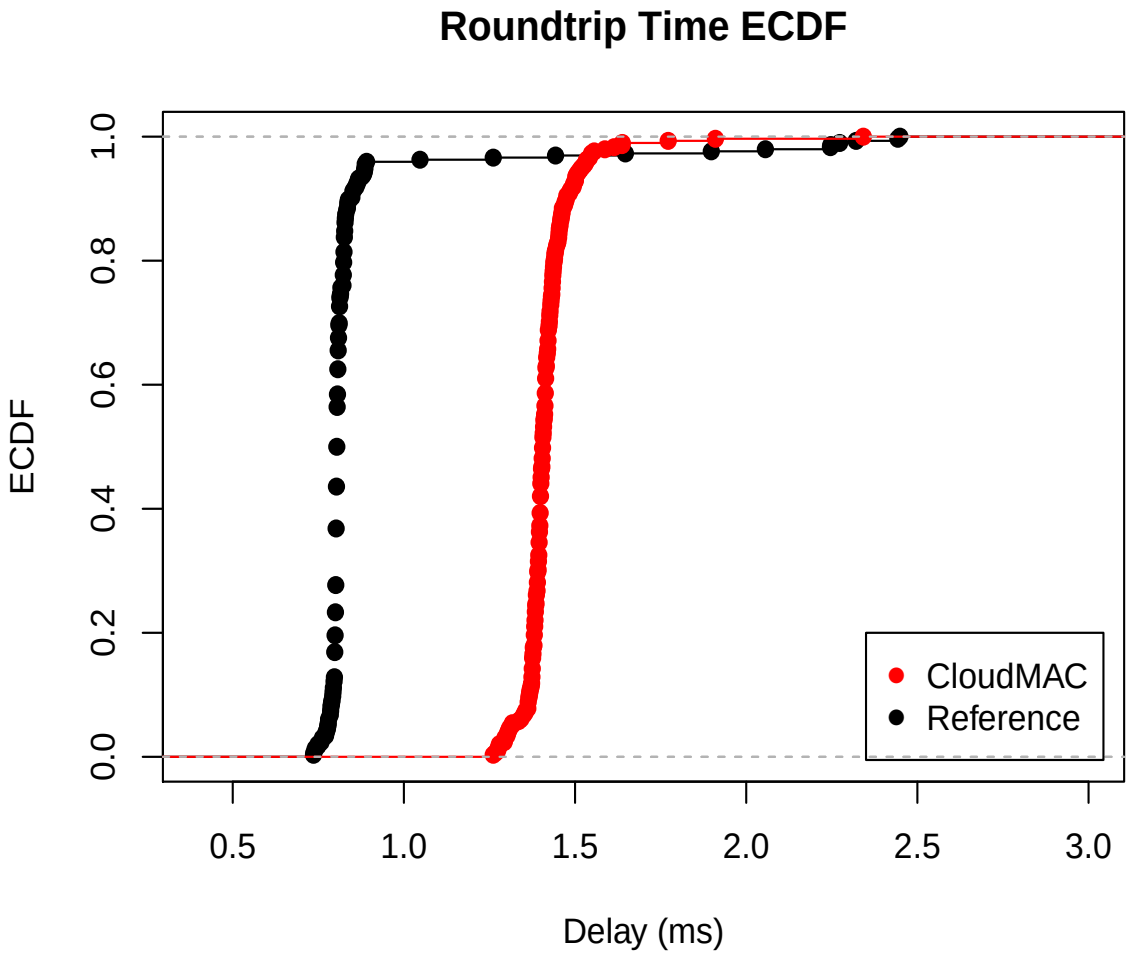


Figure 21: Roundtrip time ECDF

This causes a substantial increase in the delay of the system.

5.4.5 Association Time

We tested the association time of the system and compared it to a reference system. Association time is the time it takes for a station to become associatied to the wireless access point. The time is measured from the first Probe Request to the last Acknowledgment. This measurement is important because users often have expectations on the waiting time before

being connected to a wireless network. Various statistics on the results can be seen in table 2.

Statistic	CloudMAC	Reference
Minimum	1.436	1.437
1st Quantile	1.436	1.437
Median	1.436	1.437
Mean	1.477	1.439
3rd Quantile	1.436	1.445
Maximum	2.261	1.446

Table 2: Association time test results (in seconds)

As you can see from the measurements, there is little or no difference in association time between the CloudMAC and Reference network.

5.5 Summary

We have now described the tests we used and shown some results of the difference in performance between CloudMAC and a Reference network. We have for example seen that the UDP variance and wireless association time is almost the same between CloudMAC and the reference network. We have also seen that the UDP and TCP throughput and two-way delay differ slightly.

6 Conclusions

Wireless networking in large scale corporations can involve up to a hundred or even a thousand access points. Management and configuration of these access points can be very difficult. This is because the configuration interface changes heavily between manufacturers and models.

In this thesis we developed a alternate architecture for wireless networks. Instead of simply running all the packet processing logic in the access point, we move most of it to a separate machine. This separation in processing provides an additional flexibility at the cost of performance. Using this alternate architecture, one can prevent the difficulty in configuring large scale wireless networks by centralizing the configuration.

We were able to create this new architecture by using a modified hardware simulator driver and user space tunnels. We ran the hardware simulator at the Manager to provide a virtual wireless interface. This wireless interface would be, using the user space tunnels, seamlessly connected to the access point, thus providing a virtual access point to the Manager.

We can see, according to the test results, that a virtualization system like CloudMAC could be viable in real-world situations. The test results show us that there are performance losses, but they are still fairly small in relation to performance.

Our measurements show that the TCP performance losses are around 23%. This is within reasonable bounds. As the TCP protocol is used by the majority of Internet applications today, this metric is very important. An interesting aspect to investigate is the large variation in the CloudMAC performance.

The UDP performance (16.8%), approximately the same as the TCP performance, is substantial enough to be of use in a real-world network. However, the decline in performance with increased window size would be an interesting aspect to investigate in the future. The other tests such as roundtrip time and association time are all as expected.

There are many possible future developments for this system. First and foremost there are components that could be improved, such as making the modified capsulator more dynamic, by removing the static MAC address requirements. One could also add the functionality to support more than one network. This could involve multiple virtual machines using the same access point, or vice versa. Another change that could improve the CloudMAC architecture is to use kernel tunnels instead of user space tunnels. This allows the tunneling to be performed faster and thus can increase the performance of the CloudMAC network.

7 References

References

[1] Stefan Mangold & Lars Berlemann Bernhard H. Walke. *IEEE 802 Wireless Systems*. John Wiley & Sons, 2006.

[2] O. Braham and G. Pujolle. "Virtual wireless network urbanization". In: *Network of the Future (NOF), 2011 International Conference on the*. 2011, pp. 31 –34. DOI: 10.1109/NOF.2011.6126678.

[3] Linux Software Documentation. *Iperf Man Page*. URL: http://staff.science.uva.nl/~jblom/gigaport/tools/man/iperf.html.

[4] Linux Software Documentation. *Modprobe Man Page*. URL: http://linux.die.net/man/8/modprobe.

[5] Wenjun Gu et al. "On Security Vulnerabilities of Null Data Frames in IEEE 802.11 Based WLANs". In: *Distributed Computing Systems, 2008. ICDCS '08. The 28th International Conference on*. 2008, pp. 28 –35. DOI: 10.1109/ICDCS.2008.17.

[6] T. Hamaguchi et al. "A Framework of Better Deployment for WLAN Access Point Using Virtualization Technique". In: *Advanced Information Networking and Applications Workshops (WAINA), 2010 IEEE 24th International Conference on*. 2010, pp. 968 –973. DOI: 10.1109/WAINA.2010.61.

[7] Inc. M. Montemurro Research In Motion D. Stanley P. Calhoun Cisco Systems and Aruba Networks. *CAPWAP Protocol Specification*. 2007. URL: http://www.capwap.org/files/draft-ietf-capwap-protocol-specification-06.txt.

[8] pf.itd.nrl.navy.mil. *MGEN User's and Reference Guide Version 5.0*. URL: http://pf.itd.nrl.navy.mil/mgen/mgen.html.

[9] radiotap.org. *Radiotap - radiotap.org*. URL: http://www.radiotap.org/.

[10] James Kurose & Keith Ross. *Computer Networking: A Top-Down Approach*. 5th. Addison Wesley, 2009.

[11] David A Rusling. *The Linux Kernel*. URL: http://tldp.org/LDP/tlk/tlk.html.

[12] Jochen H. Schiller. *Mobile Communications*. 2nd. Addison Wesley, 2003.

[13] Vivek Shrivastava et al. "802.11n under the microscope". In: *Proceedings of the 8th ACM SIGCOMM conference on Internet measurement*. IMC '08. Vouliagmeni, Greece: ACM, 2008, pp. 105–110. ISBN: 978-1-60558-334-1. DOI: 10.1145/1452520.1452533. URL: http://doi.acm.org/10.1145/1452520.1452533.

[14] M. Vipin and S. Srikanth. "Analysis of open source drivers for IEEE 802.11 WLANs". In: *Wireless Communication and Sensor Computing, 2010. ICWCSC 2010. International Conference on*. 2010, pp. 1 –5. DOI: 10.1109/ICWCSC.2010.5415877.

[15] Linux Wireless. *Linux Wireless Glossary*. URL: http://linuxwireless.org/en/developers/Documentation/Glossary.

[16] Linux Wireless. *Linux Wireless: hostapd*. URL: http://linuxwireless.org/en/users/Documentation/hostapd.

[17] Linux Wireless. *Linux Wireless: iw*. URL: http://linuxwireless.org/en/users/Documentation/iw.

[18] Linux Wireless. *Linux Wireless: mac80211 Documentation*. URL: http://linuxwireless.org/en/developers/Documentation/mac80211.

[19] Linux Wireless. *Linux Wireless: mac80211_hwsim Documentation*. URL: http://linuxwireless.org/en/users/Drivers/mac80211_hwsim.