4. Implementation of QoS mechanism

4.1. Methodology

The initial stage of the implementation was creating a Click configuration and checking, debugging it in least complex methods. Only one new Click element had to be implemented (LeakyBucket) and all the others were taken from Click standard configuration. The main parts of the QoS mechanism implementation are: input, packet classifying, packet processing and output. Packet classifying is the most important part of the implementation. It classifies packets by their IP DSCP field.

4.1.1. Differentiated Services and IP DSCP field

Differentiated Services (DiffServ) is a new model in which traffic is treated by intermediate systems with relative priorities based on the type of services (ToS) field. DiffServ increases the number of definable priority levels by reallocating bits of an IP packet for priority marking. [10] The six most significant bits of the DiffServ field is called as the DSCP. The last two Currently Unused (CU) bits in the DiffServ field were not defined within the DiffServ field architecture; these are now used as Explicit Congestion Notification (ECN) bits. Routers at the edge of the network classify packets and mark them with either the IP Precedence or DSCP value in a DiffServ network. Other network devices in the core that support DiffServ use the DSCP value in the IP header to select a PHB behavior for the packet and provide the appropriate QoS treatment. In the implementation we use three types of DSCP codes recommended by Cisco [11]:

<table>
<thead>
<tr>
<th>Traffic</th>
<th>DSCP</th>
<th>Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>Voice</td>
<td>46</td>
<td>EF</td>
</tr>
<tr>
<td>Video</td>
<td>34</td>
<td>AF41</td>
</tr>
<tr>
<td>Best effort</td>
<td>0</td>
<td>Best effort</td>
</tr>
</tbody>
</table>

Figure 6: IP DSCP field codes used in implementation
4.1.2. Rate Controller problem

Although implementation is based on DRAP model (chapter 2), the offered rate controller algorithm is not used in it. The problem is that to implement this algorithm wireless device driver should have been changed, because the rate controller algorithm needs a feedback from a wireless device to count the delay of the packets. For that reason rate controller algorithm was simplified by fixing the bandwidth to 1/3 of the line bit rate for shaper’s and leakybucket’s rate.

4.2. The input part of configuration

4.2.1. The input used for debugging the implementation

In the beginning of the implementation, to check that packet classifying works imitative packet generating was used:

\[
\text{RatedSource}(<0800>, 100, 100) \\
\rightarrow \text{UDPIPEncap}(10.0.0.3, 200, 255.255.255.255, 200) \\
\rightarrow \text{SetIPDSCP}(34)
\]

Figure 7: Imitative packet generating

RatedSource(DATA, RATE, LIMIT,..) element generates packets at specified rate. It creates packets consisting of DATA, emitting at most LIMIT such packets out its single output at a rate of RATE packets per second. To be sure that the packets, that are generated by RatedSource(SADDR, SPORT, DADDR, DPORT) element, are needed ones for implementation (UDP), UDPIPEncap element was used. It encapsulates each incoming packet in a UDP/IP packet with source address SADDR, source port SPORT, destination address DADDR, and destination port DPORT. Since the traffic in the packet processing part is classified by IP dscp fields, it was easier to test classifier by setting IP packets’ DSCP fields with element SetIPDSCP(VAL). It expects IP packets as input and sets their DSCP to VAL. Then it incrementally recalculates the IP checksum and passes the packet to output 0.
4.2.2. Original input

Subsequently, the input part of configuration was changed with FromDevice, Strip and CheckIPHeader elements. FromDevice(DEVNAME) element is a push element that reads packets from network device. It reads packets from the kernel that were received on the network controller named DEVNAME. User-level FromDevice is like a packet sniffer. Packets emitted by FromDevice are also received and processed by the kernel. To get rid of the Ethernet header Strip(14) element is used. CheckIPHeader checks the IP header of the packets. It checks that the packet's length is reasonable and that the IP version, header length, length, and checksum fields are valid. Also it examines that the IP source address is a legal unicast address, or that the packet is destined for one of this machine's addresses. Original input is shown in the figure 8:

![Diagram](image)

*To packet classifying part*

Figure 8: The input part of the configuration

The loopback device was used as the input in FromDevice element. The atheros device (ath0) can be used for more realistic simulations.
4.3. Packet classifying part

In the packet classifying part firstly IPClassifier element classifies packets according to tcpdump-like patterns. In this case IP DSCP value is used where packets with IP DSCP 46 are audio packets, IP DSCP 34 are video packets and IP DSCP 0 is used for Best effort packets. All other packets are discarded. Audio packets goes directly into PacketProcessing part and then are put to the queue with the capacity of 20 packets (RTQueue::Queue(20)). The [0] output of the LeakyBucket element goes also the same way. LeakyBucket splits the flow of the packets at the specified rate (we use the bandwidth of 1/3 of the line bit rate). Another part of the splitted flow is discarded. Best effort
packets first must go to unspecified queue because BandwidthRatedUnqueue is pull-to-push element. It pulls packets at the given bandwidth RATE (we use the bandwidth of 1/3 of the line bitrate), and pushes them out its single output to PacketProcessing element class after which the packets put into NRTQueue::Queue(200).

4.4. Packet processing part

Element class PacketProcessing (figure 11) is used to set the correct source IP addresses and correct destination IP addresses to the packets, also encapsulating the packets with the wifi header. Element SetIPAddress sets the destination IP address annotation of incoming packets to the static IP address. StoreIPAddress(OFFSET) writes the destination IP address annotation into the packet at offset OFFSET. This element doesn't recalculate any checksums, so if you store the address into an existing IP packet, the packet's checksum will need to be set – for that reason SetIPChecksum element is used. It expects an IP packet as input and calculates the IP header's checksum and sets the checksum header field. For the elements SetTCPChecksum and SetUDPChecksum input packets should be TCP in IP and UDP in IP respectively. They calculate TCP/UDP header's checksum and set the checksum header field. They use the IP header fields to generate the pseudo-header. Just SetUDPChecksum checks if input packets are IP fragments, or the UDP length is longer than the packet, then pushes the input packets to the 2nd output, or drops them if there is no 2nd output. Before encapsulating the packets in the wi-fi header they must be encapsulated in the Ethernet header specified by it arguments. Element EtherEncap(ETHERTYPE, SRC, DST) does that. ETHERTYPE should be in host order. In the implementation SRC is used the MAC address of the computer on which packets are generated. WifiEncap(MODE, BSSID) - Click element that Converts ethernet packets to 802.11 packets with a LLC header. It strips the ethernet header off the front of the packet and pushes an 802.11 frame header and LLC header onto the packet. BSSID is an ethernet address. This usually the access point's ethernet address. MODE specifies which address field the BSSID field is located at. Element ExtraEncap copies the wifi_extra_header from Packet::anno() and pushes it onto the packet. There were some problems during testing work with this element, because it adds some extra bytes to the packets' headers.
4.5. Output part of configuration

The output part of configuration is made of PrioSched element which sets the priority to the packets from RTQueue andToDevice element:

![Diagram](image)

If the RTQueue is empty then second input port [1] of the PrioSched element is used. It is connected to NRTQueue. ToDevice(DEVNAME) element pulls packets and sends them out the named device using Berkeley Packet Filters (or Linux equivalent). Under Linux, a FromDevice element will not receive packets sent by a ToDevice element for the same device.
4.6. Description of new Click element

During implementation one new Click element LeakyBucket was implemented. LeakyBucket element is used to split the flow of the traffic at specified rate. It is written in C++ and compiled with click. LeakyBucket is a push element that works according the DGCRA algorithm (2.2.3). The element uses other elements from standard Click configuration. The .cc and .hh files of LeakyBucket element are attached in the appendix. The value Tau=10 was fixed in the LeakyBucket element and the bandwidth can be set simply writing router configuration file. If it is not described LeakyBucket displays an error.
5. Testing part and results

5.1. Testing methodology

For the testing part of implementation we were trying to model a hotspot.

![Traffic Flow Diagram](image)

*Figure 12: Topology of the simulation*

The results we were trying to achieve, appiel to inter arrival, end-to-end delays and throughput. The main point of implementation is reducing the TCP traffic so the UDP packets would show better performance. In order to calculate these parameters tcpdump traces of the traffic flow with and without Click implementation are needed. However, on purpose getting good results packets has been sent through Click using QoS implementation and without it. This is because Click processing takes time, and it delays packets. Another reason was that we couldn’t determine the line bit rate for wireless card and we assume that Click processing packets sets it to 1Mbps. Plenty of experiments have been prosecuted with the time of 60 seconds. During first 30 seconds only UDP packets are sent: Audio with bandwidth of 64 kbps, packets size 160 bytes and Video with bandwidth 50 kbps, packet size 512 bytes. On the 30st second TCP traffic interrupts with maximum intensity. Perl scripts were used to calculate the measurements and GNU Plot for drawing a graphs.

5.1.1. Packgen packet generator

To generate traffic for a simple tests, Packgen 0.1 packet generator was used. Packgen is a simple network packet generator handling diffserv markers, useful for testing network bandwidth and QoS. It can generate several flows of data, each having its own properties such as:
• name;
• destination;
• bandwidth;
• packet size;
• DSCP (Differentiated Services Code Point);
• time range.

The most important feature of the Packgen useful for Click implementation is that UDP and TCP packets can be sent with DSCP values at each time interval depending on the bandwidth to produce the size of the packets to generate) [3].

5.1.2. Hardware and the Madwifi driver

During the tests Proxim ORiNOCO 11a/b/g PCI Card Gold (8482-WD) wireless cards were used:

![Wireless cards used for implementation](image)

The MadWifi-ng driver was used to configure the WLAN. MadWifi is short for Multiband Atheros Driver for Wireless Fidelity. In other words: this project provides a Linux kernel device driver for Atheros-based Wireless LAN devices. The driver works such that WLAN card appears as a normal network interface in the system. Additionally there is support for the Wireless Extensions API. This allows to configure the device using common wireless tools (ifconfig, iwconfig and friends) [12]. We used one wireless card as an access point. To set that the following commands were used:

```
Wlanconfig ath0 destroy
Wlanconfig ath0 create wlan dev wifi0 wlanmode ap
Iwconfig ath0 essid "access"
```
5.1.3. Problems occurred during experiments

As it was mentioned before (chapter 4.4), ExtraEncap element that is used in the implementation adds some extra bytes to the packets’ headers. To solve this problem the Perl script correcting the traces was used. It obtains a tcpdump like dump from a trace having wrong additional bytes in the link layer header.

Another, unfortunately unsolved problem, was setting the line bit rate of wireless cards. The problem can depend on the Madwifi driver we used, because as we were trying to fix the rate of the cards at 2 Mbps with the command \texttt{iwconfig ath0 rate 2Mbps} the \texttt{iwconfig} command shows that we succeeded. However as we run some simple tests using saturated TCP connection we observed that line bitrate was about 12 mbps (figure 13, A):

![Figure 14: Throughput for TCP without using Click (A) and using it (B)](image)

Furthermore we have observed that Click setups the wi-fi card to the basic rate of 1 Mbps (figure 13, B). We were not able to figure it out why this happens. Therefore, in order to have a fair comparison, we have decided to use always Click in the experiments: with and without QoS implementation.
5.2. Results

5.2.1. Interarrival

The first experiments we were doing were to obtain a smaller jitter of interarrival. The most interesting point for these tests was receiver (the computer that obtains the packets). The graphs in figures 14 and 15 show the results without QoS implementation and with it. In both graphs the impact of the TCP traffic interruption is clearly visible. It is also seen that for audio and video traffics’ jitter decreased a little when using QoS implementation.

Figure 15: The jitter of audio traffic without QoS implementation (A) and with it (B)

Figure 16: The jitter of video traffic without QoS implementation (A) and with it (B)
5.2.2. End – To – End Delays

End – to – end delays introduce the packet delays between the sender (our access point) and receiver. The graphs in the figure 16 and figure 17 show the improvement of audio and video traffics when using the QoS implementation.

Figure 17: End-to-end delays for audio traffic without QoS implementation (A) and with it (B)

Figure 18: End-to-end delays for video traffic without QoS implementation (A) and with it (B)
5.2.3. The throughput

In the throughput graphs (figures 18 and 19) improvement using QoS implementation for both audio and video traffics are notably visible. The graphs that show UDP traffic without QoS implementation shows throughput of about 10 kbps and the graphs when using QoS shows the improvement for audio of about 4 times and the video about 3 times. However this shows that audio and video traffic decreases due to packet losses.

![Audio Throughput](image1)

**Figure 19:** The throughput for audio traffic without QoS implementation (A) and with it (B)

![Video Throughput](image2)

**Figure 20:** The throughput for video traffic without QoS implementation (A) and with it (B)
The TCP throughput was also captured just to show that improvement we have is because of reducing the TCP traffic to 200 kbps as it is shown in figure 20.

Figure 21: The throughput for TCP traffic without QoS implementation (A) and with it (B)
6. Conclusion

The objects of the project were implementing a Linux wi-fi hotspot with QoS support, without changing the Linux Kernel driver, taking the QoS measurements, demonstrating how audio and video behave with the implementation and without it. I have learned using new flexible software for building configurable routers Click modular router and implemented a QoS mechanism with it. The audio and video traffic delay, jitter and losses impacting by TCP connection was demonstrated. By the reason of having serious problems during the implementation and testing works, some of the objects were not achieved. The experiments that were prosecuted are so far not realistic. The implementation should be tested in more truthful environment. However the results of the tests described in the thesis show, that the implementation works. I maintain that further work is needed to finalize and study the Quality of Service Mechanism for a Hotspot. Additionally, more testing is needed to understand and adjust Click, also it’s interaction with the Madwifi drivers. There are problems to solve that were discovered during the implementation. It’s solving the extra bytes problem in packet’s headers created by ExtraEncap element, limiting the line bit rate.
List of references


[9] Click Modular router element list: http://read.cs.ucla.edu/click/elements


Appendix

1. Click configuration `conf_no_qos.click` file without QoS implementation:

```java
@elementclass PktProcessing {
    @input -> SetIPAddress(192.168.10.1) // src IP
    @input -> StoreIPAddress(12) // stores the src IP
    @input -> SetIPAddress(192.168.10.2) // dst IP
    @input -> StoreIPAddress(16) // stores the dst IP
    @input -> SetIPChecksum
    @input -> ipc :: IPClассifier(tcp, udp, -)
    @input -> SetTCPChecksum
    @input -> encap :: EtherEncap(0x8000, 00:20:A6:58:56:40, 00:20:A6:58:55:B8)
       -> WifiEncap(0x00, 00:20:A6:58:56:40)
       -> ExtraEncap()
       -> output;
    @output -> ipc[1] -> SetUDPChecksum -> encap;
    @output -> ipc[2] -> encap;
}

FromDevice(lo) -> Strip(14)
    -> CheckIPHeader()
    -> PktProcessing
    -> Queue
    -> ToDevice(ath1);
```

2. Click configuration `conf_qos.click` file with QoS implementation:

```java
@elementclass PktProcessing {
    @input -> SetIPAddress(192.168.10.1) // src IP
    @input -> StoreIPAddress(12) // stores the src IP
    @input -> SetIPAddress(192.168.10.2) // dst IP
    @input -> StoreIPAddress(16) // stores the dst IP
    @input -> SetIPChecksum
    @input -> ipc :: IPClассifier(tcp, udp, -)
    @input -> SetTCPChecksum
    @input -> encap :: EtherEncap(0x8000, 00:20:A6:58:56:40, 00:20:A6:58:55:B8)
       -> WifiEncap(0x00, 00:20:A6:58:56:40)
       -> ExtraEncap()
       -> output;
    @output -> ipc[1] -> SetUDPChecksum -> encap;
    @output -> ipc[2] -> encap;
}

RTqueue::Queue(20);
NRTQueue::Queue(200);

FromDevice(lo) -> Strip(14)
    -> CheckIPHeader()
    -> classify::IClassifier(ip dscp 46,
                              ip dscp 34,
                              ip dscp 0,
                              -);
    classify[0]
```

38
-> Print("audio")
-> PktProcessing
-> RTqueue
-> sched::PrioSched;

classify[1]
  -> leakybucket::LeakyBucket(204800);  
  leakybucket[0]  
    -> Print("video")
    -> PktProcessing
    -> RTqueue;
  leakybucket[1]  
    -> Print("leakybucket_drop_out")
    -> Discard();

classify[2]  
  -> Queue  
  -> BandwidthRatedUnqueue(200 kbps)
  -> Print("shaper_output")
  -> PktProcessing
  -> NRTQueue
  -> [1] sched;

classify[3]  
  -> Print("shit")
  -> Discard();

sched -> ToDevice(ath1);

3. The new Click element’s LeakyBucket .cc/.hh files:

**Leakybucket.cc:**

```cpp
#include <click/config.h>
#include <click/confparse.hh>
#include <click/error.hh>
#include <click/glue.hh>
#include "leakybucket.hh"
CLICK_DECLS

LeakyBucket::LeakyBucket() { lvst=0; tau=10; }
LeakyBucket::~LeakyBucket(){}

int LeakyBucket::configure(Vector<String> &conf, ErrorHandler *errh){
    CpVaParseCmd cmd = (is_bandwidth() ? cpBandwidth : cpUnsigned);
    if(cp_va_parse(conf,this,errh,cmd,"maximum rate",&rate,cpEnd)<0) return -1;
    rate.set_rate(rate,errh);
    if(rate<=0) return errh->error("rate must be larger than 0");
    return 0;
}

void LeakyBucket::configuration(Vector<String> &conf) const
{
    conf.push_back(is_bandwidth() ? cp不解parse_bandwidth(rate.rate()) : String(_rate.rate()));
}
```

39
void
LeakyBucket::push(int, Packet *p){
    t=Timestamp::now();
    ck=lvst+((p->length())/(rate/8));
    yk=ck-t;
    if (yk<=tau) {
        if (ck>t){
            lvst=ck; }
        else lvst=t;
        output(0).push(p);
    } else
        output(1).push(p);
}

CLICK_ENDDECLS
EXPORT_ELEMENT(LeakyBucket)

LeakyBucket.hh:

#ifndef CLICK_LEAKYBUCKET_HH
#define CLICK_LEAKYBUCKET_HH
#include<click/element.hh>
#include<click/gaprate.hh>
CLICK_DECLS

class LeakyBucket : public Element { public:
    LeakyBucket();
    ~LeakyBucket();

    const char *class_name() const { return "LeakyBucket"; }
    const char *port_count() const { return "1/2"; }
    const char *processing() const { return PUSH; }
    bool is_bandwidth() const { return class_name()[0] == 'B'; }
    int configure(Vector<String>&, ErrorHandler*);
    void configuration(Vector<String>&) const;
    bool can_live_reconfigure() const { return true; }
    void push(int port, Packet *);

private:
    int lvst;
    uint32_t ck;
    int yk;
    int tau;
    uint32_t rate;
    uint32_t_t t;
    GapRate _rate;
};

CLICK_ENDDECLS
#endif
Exte cheap
1 Mbps
threw a typ.