Implementation of Explicit Delay Control for Networked Computer Games

Studienarbeit

von

cand. inform. Dirk Markwardt

Institut für Betriebssysteme und Rechnerverbund
Technische Universität Braunschweig

Implementation of Explicit Delay Control for Networked Computer Games

Studienarbeit

von

cand. inform. Dirk Markwardt

Institut für Betriebssysteme und Rechnerverbund
Technische Universität Braunschweig

Betreuer der Arbeit: M. Sc. Xiaoyuan Gu
Prof. Dr.-Ing. Lars Wolf

Abstract

The usage of Networked multiplayer games increased during the last years all over the world due to the availability of broadband Internet links. Most of these games are sensitive against delays, therefore support from the network is required. An important aspect is that packets which are out of delay boundary become useless for the application. The implementation of an application-initiated control of packet lifetime called Explicit Delay Control (EDC) will be introduced in this treatise. With EDC a Maximum Tolerable Delay (MTD) will be embedded within the IP header options. At each hop the MTD will be examined and updated according to the time the packet spent at this hop. When the MTD expires the packet will be dropped because it became useless for the application. In this case an ICMP EDC Time Exceeded message will notify the sender of the packet. One benefit of using EDC is that the network will not transport packets that became already obsolete, thus saving bandwidth and processing time. Another benefit is that it becomes impossible that packets which are already out of time still arrive at their destination.

This work introduces the first implementation of EDC within the Linux Kernel. The implementation has been tested using User-Mode-Linux virtual hosts and a network emulator. Several test-cases of different distances have been simulated using appropriate timings on these virtual hosts.

The tests of this implementation shows that saving nearly 5% of bandwidth can be achieved using EDC.

Contents

1 Introduction

The distribution of Network computer games rised due to the popularity of the Internet. The availability of fast Internet connections even increased this development. Many games, especially First Person Shooters (FPS), require very quick reaction times. Therefore they demand near-real-time support from the network they use.

The architecture of many Network computer games is that there is one dedicated server and several clients connecting to it. The distance between the clients and the server they use has a direct impact on the delay for the particular connection. The delay even increases if the path in the network contains segments of smaller bandwidth like interconnection links between two providers. In these segments the probability of congestion is higher.

The working group around Xiaoyuan Gu proposed a paper in which they introduced a new way of support through the network: Explicit Delay Control (EDC) [1].

The idea of EDC is to enable the application to determine the Maximum Tolerable Delay (MTD) of its packets.

In Chapter 2 the algorithm of Explicit Delay Control will be presented, Chapter 3 describes the implementation of EDC within the Linux Kernel and the test application. The test environment and the tests accomplished are described in Chapter 4. Finally the conclusions and an outlook to future work is presented in Chapter 5.

2 Algorithm of Explicit Delay Control

In the following chapter the algorithm of Explicit Delay Control will be outlined.

2.1 Current Packet Lifetime Control in IP

The Time To Live (TTL) Field of IPv4 was designed with two functions in mind: one is to limit the lifetime of TCP segments, the other is to terminate Internet routing loops. The TTL value is expressed in seconds and each hop is supposed to decrement it by at least one. When TTL reaches 0 the packet will be dropped. The main purpose of the TTL Field is counting the hops the packed passes. Therefore it has been renamed to "HopLimit" in IPv6.

As the time unit of this field is second it is obviously that it is unsuitable for limiting the lifetime of packets of time-critical interactive multimedia applications which often require a one-way maximum packet delay of 100 ms.

2.2 Explicit Delay Control

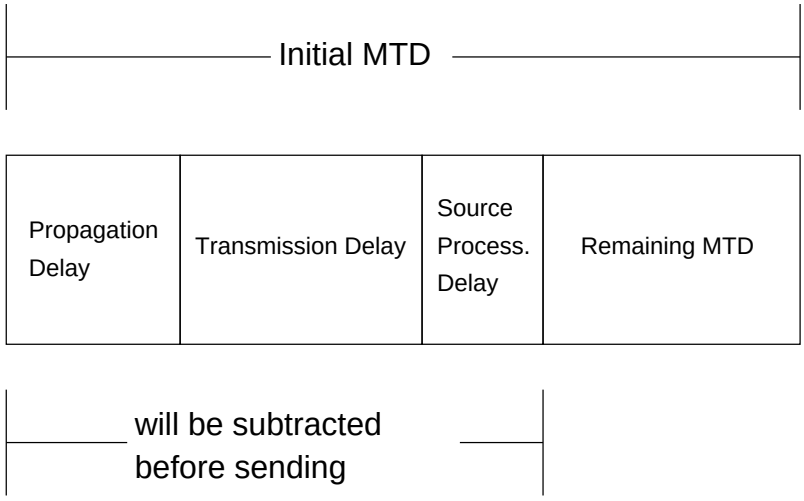
The TTL Field is unusable for satisfying the need of high resolution packet lifetime control. Therefore another mechanism must be taken into account: Explicit Delay Control (EDC).

2.2.1 EDC in brief

The basic idea is to include an additional field within the options of each IP packet which contains a one way Maximum Tolerable Delay (MTD) value in milliseconds (ms).

The initial MTD value is given by the application. Each application has its own MTD depending on its type. A First Person Shooter normally has a lower MTD than a Voice-over-IP (VoIP) application.

This initial MTD is subtracted by three values: An estimated propagation delay from source to destination host, an estimated transmission delay of the source host and the processing delay used until the data is ready to be sent. The result will be put into each IP packet of the interactive real-time flow. See picture 2.1.



Picture 2.1: Calculation of remaining MTD

Each router on the path between source and destination will decrement this value by its own processing and queueing delay. It will then write back the new value into the IP packet.

If the MTD reaches zero before the packet arrives at its destination the IP packet will be dropped and an ICMP EDC TTL Exceeded notification will be sent back to the source of this packet. This notification will help the sender to adjust the possibly wrong guessed initial MTD values.

2.2.2 Maximum Tolerable Delay (MTD) Definition

The MTD is the one-way maximum tolerable delay for data packets of a multimedia data stream. Different applications can have different MTD values. The unit of these values is millisecond.

This initial MTD value will be subtracted by three values: Propagation Delay, Transmission Delay and Source Processing Delay.

The Propagation Delay is the time the electrical or optical signals will run on the cables between source and destination. For an estimation of the Propagation Delay the locations of source and destination must be known. Most of the internet infrastructure consists of optical fiber. So the speed of light in the media glass (200000 km/second) must be taken as propagation speed. Equation 2.1 shows the calculation of the estimated Propagation Delay (est_prop_delay).

$$est\_prop\_delay = \frac{Estimated\,distance}{Propagation\,speed} \tag{2.1}$$

The Transmission Delay is defined by the slowest link between source and destination. Usually this is the link on the client side, whose bandwidth will be taken into account. This delay is the time from sending the first bit to sending the last bit of the data packet.

For estimating this delay the typical packet size of the application must be considered. Equation 2.2 shows the calculation of the estimated Transmission Delay (est_tran_delay).

$$est\_tran\_delay = \frac{Typical\,packet\,size * 8}{Client\,bandwidth} \tag{2.2}$$

The initial MTD given by the application will be subtracted by these values to get the MTD value (equation 2.3).

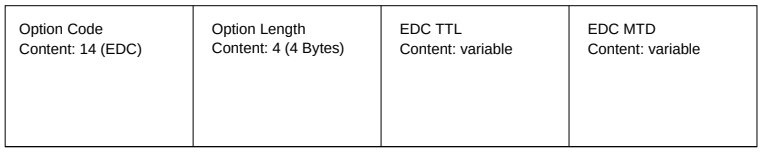
$$MTD = initial\,MTD - (est\_prop\_delay + est\_tran\_delay) \tag{2.3}$$

The Source Processing Delay (src_proc_delay) is the time the source needs to process the data until it is ready to be sent. This time is considered for each packet individually. The MTD calculated in equation 2.3 will be decremented by this delay as shown in equation 2.4. After that the MTD will be put into the IP options of this packet.

$$MTD = MTD - src\_proc\_delay \tag{2.4}$$

2.2.3 MTD placement within the IP header

The MTD will be placed within the IP header. A reuse of the already existing packet lifetime control fields TTL (IPv4) or HopCount (IPv6) might cause incompatibilities and confusion with non-real-time data. So the IP Options will be used for EDC purposes. The MTD is defined as a 4-byte IP Option with the type 14 as shown in figure 2.2.



Picture 2.2: EDC IP Option Format

The first byte indicates the option type. The option type 14 is used temporarily until the Internet Assigned Numbers Authority (IANA) assigns a final type for MTD.

The second byte denotes the length of this IP Option of 4 bytes.

The third byte contains the TTL value at the hop where the last MTD update took place. It is needed to detect EDC incompatible hops and in that case to adjust the remaining MTD.

The fourth byte is the remaining MTD value. It is stored as a single byte value and has the unit milliseconds. Therefore MTD can have values between 0 and 255.

2.2.4 Update and Per-hop behavior on MTD

The MTD value is the remaining lifetime of the data packet. Each router on the path from source to destination needs its own amount of processing time on this packet. This amount of time is variable and depends on several factors: receiving, eventually some sanity checks, taking the routing decision, queueing and finally sending. This implies that each router must update the MTD field according to the needed processing time mentioned above.

There are two approaches for finding the needed processing time and updating the MTD value. Each one has its advantages and disadvantages. The first one is Absolute Tolerable Delay Update. In this approach the router records the arrival time of the packet. Then the packet will be processed normally. The second time value will be taken in the moment where the packet has been taken out of the send queue for delivery. The difference is the absolute processing time (APT) which contains all work to be done and is therefore very accurate. If the MTD value is smaller than this APT the packet will be discarded. Otherwise a new MTD value will be calculated as MTD=MTD-APT. This new MTD will be put back into the IP header. The disadvantage of this approach is that all work on the packet will be done before finally detecting that this packet must be discarded due to exceeded MTD.

The other approach is Proactive Tolerable Delay Update which tries to avoid the disadvantage mentioned above at the cost of being less accurate. At the time of receiving the packet the router estimates a possible processing time (PPT). This PPT contains times for all steps of work to be done. If this PPT value is greater than the MTD value of the packet received, it will be discarded. Otherwise a new MTD value will be calculated as MTD=MTD-PPT. This new MTD will be put back into the IP header. Estimating a relatively accurate PPT is possible because tasks like sanity checks and the routing decision have nearly constant execution times. The queueing delay can also be estimated rather accurate because the router knows best about the state of the send queues. The advantage of this approach is that no processing time will be wasted on packets with exceeded lifetime. The disadvantage is the

possible inaccuracy in the estimated PPT. A reason for that could be that data packets with higher priority arrive after the considered packet. These packets will be put into the send queue in front of this specific packet.

The EDC_TTL Field will be handled similar to the TTL Field within the IP header. Each EDC-enabled router will decrement this value by one. Before doing that the router checks if there is a difference between the EDC_TTL and the normal TTL value. In this case an adjustment of the MTD value must take place because the packet passed an EDC incompatible hop. The exact procedure for this is described in the next part.

2.2.5 Dealing with Incompatibilities

It is possible that routers exist which don't know the code for EDC option within the IP options. The IP options will be parsed by these routers but they won't modify options they don't know. Anyway they will forward these packets but don't modify the MTD and EDC_TTL values. These routers will be detected because of the difference between TTL and EDC_TTL. This detection takes place before modifying the EDC and MTD values according to the own used processing time.

In this case a correction of the MTD value must take place because the real MTD value is lower than it seems to be. There could be several EDC incompatible routers one after another. This would raise the difference and also the needed correction for the MTD value. The difference must be calculated as in equation 2.5. The EDC_TTL is the TTL value at which hop the last EDC update took place and the current TTL (curr_TTL) is the TTL value after decrementing due to forwarding.

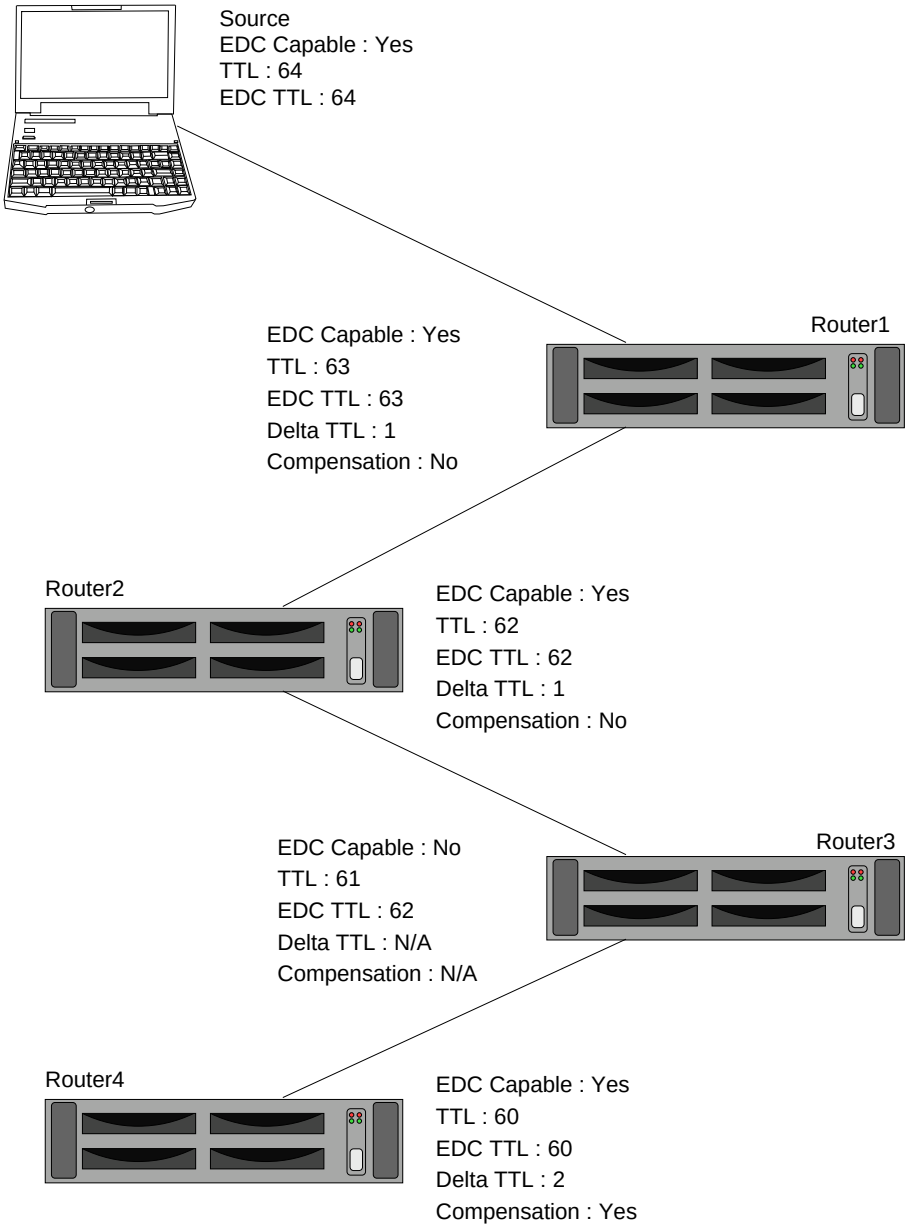
$$\Delta TTL = EDC\_TTL - curr\_TTL \tag{2.5}$$

If the ΔTTL value is equal or greater than 2 it is detected that the packed passed one or more EDC incompatible hops. In this case the error on the MTD value must be compensated using equation 2.6

$$MTD = MTD - Current\_processing\_time * (1 + \beta * (\Delta TTL - 1)) \tag{2.6}$$

with the compensation factor β. The use of the value 1.0 is recommended. In this case it is assumed that the EDC incompatible router(s) has/have used the same processing time than the actual router. This correction of the MTD value is an empirical approach.

Figure 2.3 shows the detection and compensation of an EDC incompatible hop. The TTL and EDC_TTL values are shown at the time the packet leaves the hop. The Delta TTL value is shown as being calculated after TTL update but before EDC option update as mentioned earlier.



Picture 2.3: Detection and Compensation of an EDC incompatible hop

2.2.6 Explicit Delay Control Notification (EDCN)

In the case of a packet drop due to exceeded MTD the particular hop shall send an ICMP Time to Live Exceeded packet (Type 11) back to the source. This procedure is called Explicit Delay Control Notification (EDCN). EDCN introduces a new code for the TTL Exceeded. Until IANA assignes a code number for this purpose "2" will be used because 0 and 1 are already assigned for "TTL exceeded in Transit" and "Fragment Reassembly Time Exceeded". Additionally the original IP header and the first eight data bytes will be included in this ICMP packet.

RFC 1122 says that it is required to send back the original IP header and the first eight bytes of the datagram. It is possible to send back more up to a maximum ICMP packet size of 576 bytes but one main idea of EDC is saving bandwidth so EDCN packets will be limited to the minimum possible size.

The first eight bytes of the datagram contain the source and destination ports of the original UDP packet which makes it possible to associate an incoming EDCN to a specific application on the source host.

2.2.7 Sender's behavior on Reception of EDCN

When the sender receives a single EDCN message it is not yet necessary to change any parameters of EDC. But in the case of receiving more than one (typically three) EDCN messages within one given time slot, typically 4 Round-Trip-Times (RTTs), it might be required to adapt the initially estimated values for the different delays. This adaption will be performed in small steps.

The reason for these EDCN messages could be too conservative estimations for the propagation und transmission delay values. It is difficult to estimate the propagation delay value without knowledge of how far away the sender is from the receiver from the point of view the data packet has.

For adapting the estimated values two cases must be distinguished: A loss because of a misestimated propagation delay or a misestimated transmission delay. These cases can be differentiated by comparing the initial TTL value and the TTL value of the original IP header within the EDCN message.

If this difference is below 10 the probability of a too big value for the propagation delay is high. Otherwise the probability of a too big value for the transmission delay is high.

For adapting the initially misestimated delays two compensation factors δ (for the transmission delay) and λ (for the propagation delay) will be used. They will both be initialized as 0. Equation 2.7 will be used to calculate the MTD taking these compensation factors into account.

If the need for compensation is detected and the difference between the two TTL values is below 10 a value of 0.25 will be added to λ. After this compensation the application must wait for another 4*RTT before the next compensation can take place. This must be done to avoid over-compensation because there might be several packets with the old MTD value still on the wire. If needed the procedure of compensation will be repeated until a maximum allowed estimated propagation delay threshold (est_prop_delay_thd) will be reached.

$$MTD = initial\,MTD - (\frac{est\_prop\_delay}{1+\delta} + \frac{est\_tran\_delay}{1+\lambda}) \tag{2.7}$$

If the need for compensation is detected and the difference between the two TTL values is equal or above 10 a value of 0.25 will be added to δ. After this compensation the application must wait for another 4*RTT before the next compensation can take place. This must be done to avoid over-compensation because there might be several packets with the old MTD value still on the wire. If needed the procedure of compensation will be repeated until a maximum allowed estimated transmission delay threshold (est_tran_delay_thd) will be reached.

When a maximum threshold is reached no further compensation takes place. In this case it is improbable that a misestimation is still the reason for the packet losses. The compensation factors and thus the MTD value will be restored to their original values.

3 Implementation of Explicit Delay Control

In the following sections the implementation of EDC within the Linux Kernel and the realization of the test programs are explained step-by-step.

3.1 EDC Handling in the Linux Kernel

The initial implementation of EDC packet handling was realized using Linux. This Operating System has been chosen because the source code is freely available. The add-on "User-Mode-Linux" (UML) makes it possible to develop in kernel-space without involving the main host. After compiling the UML-Kernel it can simply be started within user-space of the main host. Even coding bugs cannot crash the whole system.

3.1.1 Changes within the Kernel

In order to implement EDC within the Linux Kernel it was necessary to make changes in different source files. In the following these changes will be presented.

3.1.1.1 icmp.h

An additional code for ICMP packet type "Time Exceeded" has been defined for EDC with the value 2.

3.1.1.2 icmp.c

When generating ICMP packets the Linux Kernel includes the IP header and data up to an ICMP packet size of 576 bytes. RFC 1122 demands that only the IP header and the first eight data bytes are included. As EDC is intended to save bandwidth the size of ICMP EDC Time Exceeded packets is limited to the minimum which is required.

3.1.1.3 inetdevice.h, sysctl.h and devinet.c

The structure of parameters for a network interface has been extended by two values: real_speed and mtd_decrement. The latter is for testing purposes and can be dropped later.

3.1.1.4 ip.h

A new IP option code for EDC has been added to the existing IP options. Until an exact number for this purpose has been assigned by the Internet Assigned Numbers Authority (IANA) the code number 14 will be used.

The internal IP options structure has been modified to additionally store the MTD value and the EDC TTL Field.

3.1.1.5 ip_forward.c

The test code for the parameter "mtd_decrement" is located in the source file ip_forward.c. An IP packet with EDC option will be dropped if its MTD value is smaller or equal than the mtd_decrement value of the network interface the packet would leave. Otherwise the decremented MTD value will be written back into the packet, a new IP checksum will be calculated and the packet will be enqueued for sending.

These code changes are needed for testing and can be removed later.

3.1.1.6 ip_options.c

In the source file ip_options.c the additional fields of the internal IP options structure needed for EDC get filled.

3.1.1.7 ip_output.c

Every IP packet the Linux kernel receives gets a timestamp. To measure how long a locally created packet stayed in the send queue it was necessary to implement this timestamp also at packet creation time.

3.1.1.8 sch_generic.c

If an IP packet with EDC option is dequeued from the network interface send queue the Absolute Processing Time (APT) will be calculated. Additionally the amount of time which will be needed to send out this packet on the given network interface regarding packet size and real upstream bandwidth (with the parameter "real_speed") will also be calculated.

If the difference between the IP TTL and the EDC TTL values is greater than 1 the packet passed one or more EDC incompatible hops. In this case the EDC TTL Field will be set to the value of the IP TTL Field. The processing time calculated above will be multiplied with the number of EDC incompatible hops. The missing update of the MTD Field caused by the EDC incompatible router(s) will be compensated by assuming the same processing delay measured above.

If the MTD value of the packet is less or equal than the sum of these two times the packet will be dropped and an ICMP EDC Time Exceeded packet will be sent back to the source of the original packet. Otherwise the MTD Field within the packet will be updated considering the number of EDC incompatible hops and the IP checksum will be recalculated.

3.1.2 Configuration of an EDC-enabled Kernel

Configuration of the EDC parameters can be done via sysconfig interface. There are two parameters which affect EDC for every network interface. They can be found within proc-filesystem:

```
/proc/sys/net/ipv4/conf/*/{mtd_decrement,real_speed}
```

3.1.2.1 mtd_decrement

The "mtd_decrement" parameter is needed for testing purposes only. If an IP packet with EDC option leaves the system through a network interface for which a value bigger than 0 is set, the MTD value of this packet will be additionally decremented by this value. If the MTD reaches 0 the packet will be dropped and an ICMP EDC Time Exceeded notification will be sent back to the source of the original IP packet. The default value of this parameter is 0 which is correct for normal operation.

Example:

```
echo "10" > /proc/sys/net/ipv4/conf/eth0/mtd_decrement
```

will decrement the MTD value by an additional amount of 10 milliseconds if the IP packet leaves the system via eth0.

3.1.2.2 real_speed

In the parameter "real_speed" the actual upstream link speed of a network interface must be set. The unit is bits per second (bit/s). It is possible that the actual link speed of a network interface is less than the speed which the hardware might reach. An example: A DSL line from German Telekom has an upstream link speed of 128 kBit/s but the DSL modem is connected via ethernet at a link speed of 10 MBit/s. In this case the parameter must be set to 128000.

Example:

```
echo "128000" > /proc/sys/net/ipv4/conf/eth0/real_speed
```

will set the real upstream speed of eth0 to 128000 bit/s or 128 kBit/s.

3.2 User-Space Application

The user-space test application consists of a server and a client. Both are written in C.

3.2.1 Server

The implementation of the server is rather simple. Its main task is binding to a UDP socket and listen for data packets. Due to the fact that EDC shall drop a packet at the end of its lifetime there is no need to care about EDC when it arrives at its destination.

The Administrator of the server knows where the hardware is located. Clients can connect this server from all over the world. So it is advisable to leave the task of estimating the propagation delay to the server because only one location database must be maintained.

If the server receives an UDP packet containing the string "getpropdelay:xxx" where "xxx" is the name of the city the client is located, it calls the helper program "distance.pl" passing its own location and the location it received from the client as parameters. The estimated propagation delay will be sent back to the client within a single byte.

3.2.1.1 Starting the Server

The server process must be started with its location as an argument:

```
./server "New York"
```

starts the server process with its location "New York".

3.2.1.2 Output of the Server

The Output of the Server looks like this:

```
lxd:/mnt/dirk/uni# ./server "New York"
Search for distance / propagation delay between New York and Braunschweig
New York <--> Braunschweig
Distance=6223    Propagation delay=31
got packet from 192.168.188.1
packet is 25 bytes long
packet contains "getpropdelay:Braunschweig"
got packet from 192.168.188.1
packet is 75 bytes long
packet contains "Hello World ... 0"
got packet from 192.168.188.1
packet is 75 bytes long
packet contains "Hello World ... 1"
got packet from 192.168.188.1
packet is 75 bytes long
packet contains "Hello World ... 2"
got packet from 192.168.188.1
packet is 75 bytes long
packet contains "Hello World ... 3"
got packet from 192.168.188.1
packet is 75 bytes long
packet contains "Hello World ... 4"
```

3.2.2 Client

The client program named ”udptest” has to do a substantial greater amount of tasks than the server. Its main purpose is to send test data to the server using IP packets with EDC option enabled.

Before it can do that the following tasks must be done: Determining RTT between client and server by querying the helper program ”rttometer”, estimating propagation delay by asking the server, estimating transmission delay using the given bandwidth and opening a raw socket for receiving ICMP EDC Time Exceeded notification packets.

Test data is being generated using three profiles: ”audio”, ”video” and ”game”. Profile ”video” assumes that a normal video stream consists of 25 frames per second, each frame has 200 bytes. The other two profiles habe been measured by examining the data streams of normal applications (Table 3.1).

For profile ”game” the data stream of the First Person Shooter ”Unreal Tournament 2004” (UT2004) has been examined. During normal gameplay the client sends 20 packets per second with an average size of 75 bytes to the server.

For profile ”audio” the data stream of the Internet Telephony application ”Skype” has been examined. A call to the Echo Test Service resulted in 23 Packets per second with an average size of 136 bytes.

Table 3.1: Packets per second and average packet sizes in bytes of different multimedia applications

	Video	Audio (Skype)	Game (UT2004)
Packets/s	25	23	20
Avg. size	200	136	75

During sending test data according to one of the three profiles the client application takes care of analyzing ICMP packets the client host receives. This is necessary because an ICMP raw socket receives all ICMP packets addressed to the particular host even if they are not related to the running application. Furthermore the estimated transmission and propagation delay values will be adapted using the appropriate correction factors if the circumstances apply to the description in the EDC algorithm.

3.2.2.1 Starting the Client

The client program expects 4 parameters: destination host, own location, own upstream bandwidth and profile to be used. The parameter "own location" must be the name of the city where the client is located, it must be enclosed with quotation marks if the name contains spaces. The "own upstream bandwidth" must be given in bit/s, "profile" can be one of "audio","video" or "game".

```
./udptest 192.168.191.2 Braunschweig 33600 game
```

starts the client with the destination host 192.168.191.2, the own location "Braunschweig", an upstream bandwidth of 33600 bit/s and the profile "game".

3.2.2.2 Output of the Client

The output of the client program looks like this:

```
fraggle:/home/dirk/uni# ./udptest 192.168.191.2 Braunschweig 33600 game
Hostname       : 192.168.191.2
IP-Address     : 192.168.191.2
Found min RTT ...
Min RTT: 31.968 msec, confidence 0.006 (CI), epsilon 2.000 msec

Buffer[9]:31.968 msec, confidence 0.006 (CI), epsilon 2.000 msec

RTT=31
using 31 as RTT value
Initial MTD=100ms ,est. prop. delay=31ms ,est trans. delay=26
=> resulting MTD=43.000000

Thresholds:
Propagation delay : 15
Transmission delay : 13
.............+......+..+timediff result=415
last 3 EDCN Packets not within 4*RTT
..+timediff result=207
last 3 EDCN Packets not within 4*RTT
```

```
....
Received SIGINT ... aborting !
Closing Socket ! BYE !

Statistics:
===========
Sent packets         : 28
Received EDC-Packets : 4

Last 10 EDC-Packets received at:
1105394735.581251
1105394735.477322
1105394735.373273
1105394735.61336
```

3.3 Helper Programs

In order to keep the server and client simple it was necessary to use several helper programs.

3.3.1 distance.pl

The small Perl script "distance.pl" calculates distance and propagation delay between two locations on earth. It is called from the server process with the two locations as arguments. It fetches the latitude and longitude values of these locations from the Cities-database from Geobytes [2]. The distance is calculated using equation 3.1 from the website of C. Michaels [3].

$$\begin{aligned} distance \;=\; \arccos\big(\;& \cos(lat1)*\cos(lon1)*\cos(lat2)*\cos(lon2) + \\ & \cos(lat1)*\sin(lon1)*\cos(lat2)*\sin(lon2) + \\ & \sin(lat1)*\sin(lat2)\;\big)*radius \end{aligned} \tag{3.1}$$

The "radius" variable has the value of 6378 km which is the radius of the earth. Locations to the west of the zero meridian and to the south of the equator have got negative latitude / longitude values. Equation 3.1 gives the shortest possible distance between the two locations.

For the calculation of propagation delay (equation 3.2) it is assumed that the network media is fiber optics. Lightspeed in vacuum or copper wire is ca. 300000 km/s, but in the medium glass it is only ca. 200000 km/s.

$$propagation\,delay = \frac{200000\,km/s}{distance} \tag{3.2}$$

In case of an error the estimated propagation delay is set to 0. This can happen when one or both locations can not be found in the database. So it is not possible to subtract a too big estimated propagation delay from the initial MTD which could cause wrongful packet drops.

One problem of "distance.pl" is that it can not distinguish between multiple cities which have the same name. If there are multiple cities which have the same name (e.g. "Hamburg" exists 10 times) the skript simply takes the first one it finds. This limitation is acceptable since these test programs are only a proof of concept. Another way to get the correct location of the client is pointed out in chapter "Outlook".

3.3.2 rttometer

The Program "rttometer" measures Round Trip Times using TCP. This avoids problems with firewalls blocking ICMP Echo Requests.

The program uses a raw socket so send TCP SYN packets to the destination host. After that it measures the time until either a TCP ACK or a TCP RST packet arrives for this connection. This measurement is repeated several times to get an accurate value.

The client program calls "rttometer" to get the Round Trip Time from client to server. It will need this value to correct a misestimated propagation or transmission delay.

The homepage of this program is http://idmaps.eecs.umich.edu/rttometer/ [4].

4 Tests of the Implementation

The implementation of kernel- and user-space code has been tested in a test environment realized in this work to ensure that it works as expected. Additionally it has been verified that it is possible to save bandwidth and processing time by using EDC.

4.1 Test Environment

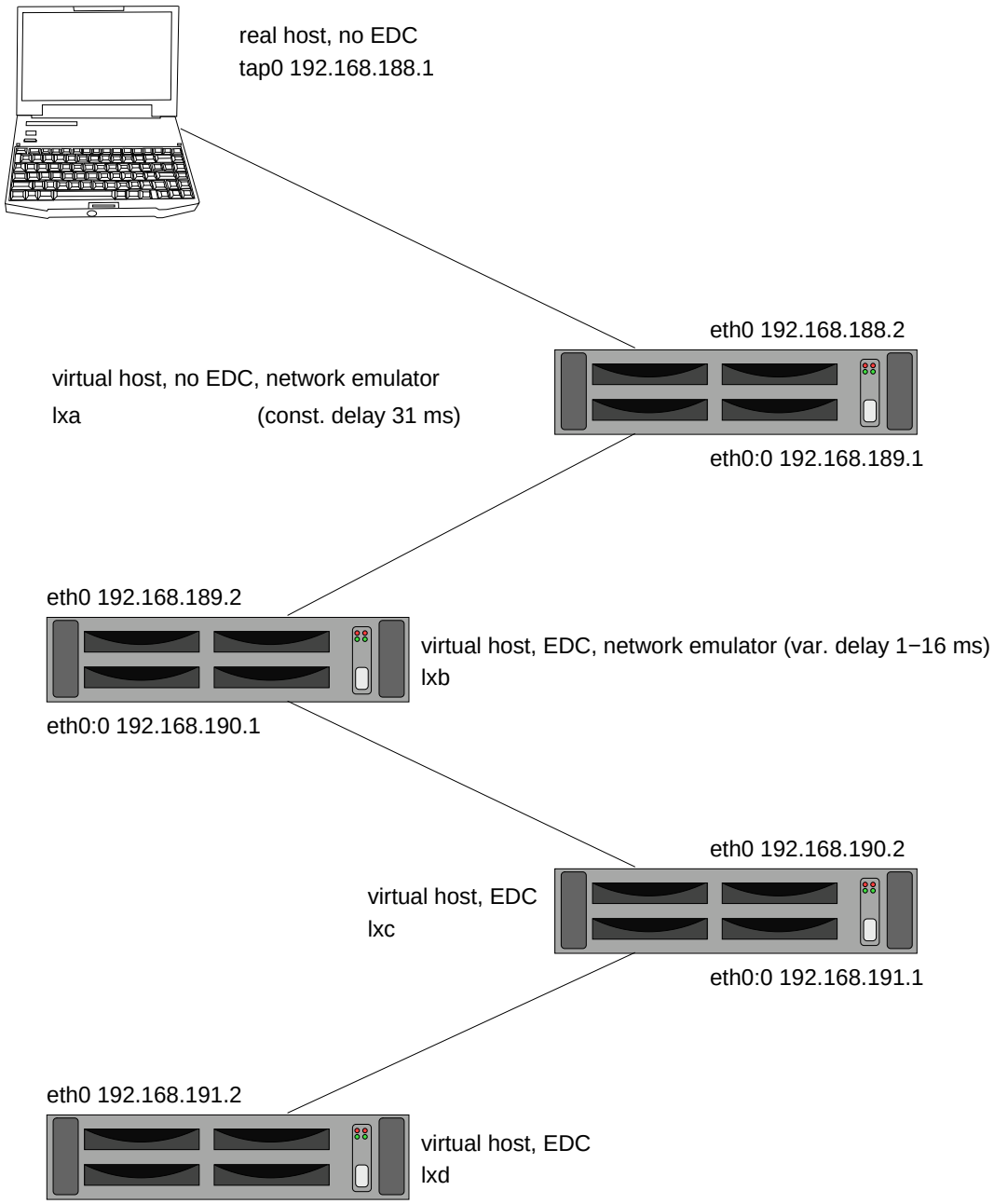
The test environment has been realized using User-Mode-Linux (UML) [5]. UML enables the user to run several virtual machines on one real linux host using a different approach than VMWare [6] or Bochs [7]. While the latter ones emulate a complete X86 PC including graphics hardware, memory and interfaces UML shares these resources with the real host. The kernel of the UML virtual host will be compiled for the special hardware platform "UM" and runs as a normal user-space program without needing root privileges. Image files on the real host will be used as virtual hard disks. Networking is done via the TAP device and a "Switch"-Daemon which emulates an ethernet switch device.

The advantage of using UML for Linux Kernel development is that the developer is able to make changes to the kernel source, recompile it and run it as a virtual machine. In the case of a software bug crashing the kernel the virtual machine will simply be terminated like any other user-space program. This speeds up kernel development enormously.

All virtual machines in this test environment run Debian Linux 3.0 with Kernel 2.4.27.

Packet forwarding is enabled on every virtual host. Network routes are set so that every host can reach all other hosts using the path shown in picture 4.1. The necessary test programs are stored on the real host and exported via NFS.

On the real host it is required to run the UML switch daemon for networking with the virtual hosts. Normally this daemon runs in "switch mode" which means that it acts like a real network switch. Normal packets (no broadcasts) are sent to the host where they belong to. If necessary it is possible to run the daemon in "hub mode" where every network packet is visible to every host. This makes it possible to trace network traffic with programs like "tcpdump" or "ethereal".



Picture 4.1: User-Mode-Linux Test configuration

The development of the Network Emulator [8] within the Linux Kernel was inspired by the network emulator "NISTNet". Network emulation will be done by the special "netem"-scheduler. The emulation parameters will be assigned by the "tc" command from the iproute2 package. It is possible to emulate constant and variable delay, random packet loss and duplication and packet reordering. It has been included in the mainstream Linux Kernel since version 2.4.27.

Picture 4.1 shows that virtual hosts "lxa" and "lxb" run as a network emulator. Host "lxa" is not EDC enabled and has a constant delay of 31 ms, "lxb" is EDC enabled and has a variable delay which is randomly chosen between 1 ms and 16 ms.

EDC and the network emulator both rely on the timestamp which every IP packet gets assigned at receive time. The network emulator keeps back the received packet for the given time. After that the EDC part decrements the MTD for this amount of time. This makes it possible to emulate a congestion at host "lxb" in this test environment.

Host "lxa" is not EDC enabled because of two reasons. Primary reason is that the helper program "rttometer" must be able to measure a certain delay which shall not be subtracted from MTD when passing this host. The mentioned delay is already considered within the estimated propagation delay. The second reason is the existence of an EDC incompatible host within this test. So dealing with EDC incompatibilities can be verified.

4.2 Running the Test

At first the server process must be started on host "lxd" within the test environment. The server gets the parameter "Braunschweig" as the virtual location. When the server is ready the client can be started on another host with the appropriate parameters for the connection.

Two tests have been taken during this work: Test 1 simulates a Voice-over-IP connection over transatlantic distance (Braunschweig - New York, 6223 km). The timings used in this test correspond to the values shown in picture 4.1.

Test 2 again simulates a Voice-over-IP connection. But in this test the simulated distance is much smaller: While using the same composition of virtual hosts like in test 1, the timings have been adapted to reflect the smaller distance between Barcelona and Braunschweig (1366 km).

The tests ended after 5000 packets have been sent.

4.2.1 Test 1: Relaxed timing on simulated transatlantic distance

The client on the real host has been started with the following parameters: ”192.168.191.2” as destination host, ”New York” as its own location, 64000 bit/s local upstream and profile ”audio”. It showed the following output at the beginning of the test:

```
fraggle:/home/dirk/uni# ./udptest 192.168.191.2 "New York" 64000 audio
Hostname       : 192.168.191.2
IP-Address     : 192.168.191.2
Found min RTT ...
Min RTT: 72.151 msec, confidence 0.191 (CI), epsilon 4.000 msec

Buffer[9]:72.151 msec, confidence 0.191 (CI), epsilon 4.000 msec

RTT=72
using 72 as RTT value
Initial MTD=100ms ,est. prop. delay=31ms ,est trans. delay=21
=> resulting MTD=48.000000
Thresholds:
Propagation delay : 15
Transmission delay : 10
.....+.........+....+timediff result=584
last 3 EDCN Packets not within 4*RTT
.+timediff result=224

Change Prop delay: new lambda 25
Last adapt: 1112729949.304584
..............+timediff result=629
timediff result=674
last 3 EDCN Packets not within 4*RTT
..+timediff result=719
timediff result=719
last 3 EDCN Packets not within 4*RTT
+timediff result=719
timediff result=90

Change Prop delay: new lambda 50
```

```
Last adapt: 1112729950.24501
...............+timediff result=719
timediff result=719
last 3 EDCN Packets not within 4*RTT
....................+timediff result=1709
timediff result=1709
last 3 EDCN Packets not within 4*RTT
.....................+timediff result=2654
timediff result=1934
last 3 EDCN Packets not within 4*RTT
.......................+timediff result=3689
```

The client found a minimum Round Trip Time of 72 milliseconds. The initial MTD value was 100 ms. After subtracting the estimated delays the resulting MTD value was 48 ms. The thresholds have been set to 50% of the original estimations. The estimation of the propagation delay has been adapted once. The new lambda value of 25 means 0.25 but internally the client uses integer calculations due to performance reasons. Every dot means a packet which has been sent out. Every "+" stands for an EDCN message which has been received.

The complete output of the test program has been shortened a bit. Here is the end:

```
last 3 EDCN Packets not within 4*RTT
.....................+timediff result=1890
last 3 EDCN Packets not within 4*RTT
.........................+timediff result=2114
last 3 EDCN Packets not within 4*RTT
..........Closing Socket ! BYE !

Statistics:
===========
Sent packets         : 5000
Received EDCN-Packets : 230

Last 10 EDCN-Packets received at:
1112730498.772039
1112730497.557226
```

```
1112730496.657360
1112730495.666509
1112730494.586671
1112730493.551816
1112730492.606967
1112730492.155026
1112730490.625264
1112730489.722400
```

The program sent out 5000 data packets and received 230 ICMP EDC Time Exceeded notifications. This means that 4.6% of the packets were dropped due to MTD reached 0. The time values of the last ten EDCN packets are UNIX time stamps in the format "seconds.microseconds". The whole test was running for 3 minutes and 49 seconds.

4.2.2 Test 2: Strict timing on shorter simulated distances within Europe

For this test a much stricter timing has been defined: The initial MTD value has been set to 40 ms. The delays on the virtual hosts were set to simulate a network path from Barcelona to Braunschweig. This is about 22% of the distance of test 1. Host "lxa" has a constant delay of 6 ms to reflect the propagation delay. Host "lxb" has a random variable delay between 1 and 15 ms to simulate changing network traffic conditions.

The client on the real host has been started with the following parameters: "192.168.191.2" as destination host, "Barcelona" as its own location, 64000 bit/s local upstream and profile "audio". It showed the following output at the beginning of the test:

```
fraggle:/home/dirk/uni# time ./udptest 192.168.191.2 "Barcelona" 64000 audio
Hostname       : 192.168.191.2
IP-Address     : 192.168.191.2
Found min RTT ...
Min RTT: 20.546 msec, confidence 0.167 (CI), epsilon 2.000 msec

Buffer[9]:20.546 msec, confidence 0.167 (CI), epsilon 2.000 msec

RTT=20
using 20 as RTT value
Initial MTD=50ms ,est. prop. delay=6ms ,est trans. delay=21
```

```
=> resulting MTD=23.000000
Thresholds:
Propagation delay : 3
Transmission delay : 10
...+.+...+timediff result=179
last 3 EDCN Packets not within 4*RTT
..+timediff result=224
last 3 EDCN Packets not within 4*RTT
+timediff result=90
last 3 EDCN Packets not within 4*RTT
.+timediff result=44

Change Prop delay: new lambda 25
Last adapt: 1112735648.875115
...............+timediff result=719
timediff result=764
last 3 EDCN Packets not within 4*RTT
....................+timediff result=1754
```

The client found a minimum Round Trip Time of 20 milliseconds. The initial MTD value was 40 ms. After subtracting the estimated delays the resulting MTD value was 23 ms. The thresholds have been set to 50% of the original estimations. The estimation of the propagation delay has been adapted once. The new lambda value of 25 means 0.25 but internally the client uses integer calculations due to performance reasons. Every dot means a packet which has been sent out. Every "+" stands for an EDCN message which has been received.

The complete output of the test program has been shortened a bit. Here is the end:

```
....................+timediff result=223315
timediff result=2024
last 3 EDCN Packets not within 4*RTT
....................+timediff result=224349
timediff result=2069
last 3 EDCN Packets not within 4*RTT
.....Closing Socket ! BYE !
```

```
Statistics:
===========
Sent packets         : 5000
Received EDCN-Packets : 236

Last 10 EDCN-Packets received at:
1112735873.224996
1112735872.190140
1112735871.155312
1112735870.165462
1112735869.175611
1112735868.182767
1112735867.237904
1112735866.203152
1112735865.123214
1112735864.133377
```

The program sent out 5000 data packets and received 236 ICMP EDC Time Exceeded notifications. This means that 4.72% of the packets were dropped due to MTD reached 0. The time values of the last ten EDCN packets are UNIX time stamps in the format "seconds.microseconds". The whole test was running for 3 minutes and 48 seconds.

4.2.3 Interpretation of the measured data

By using EDC it was possible to achieve a bandwidth saving of nearly 5% after the point of packet-drop because of exceeded MTD. Additionally the load on the routers along the path after the hop where the packet was dropped has been lowered although this is difficult to measure.

The bandwidth-saving comes into effect only in situations where the MTD value is low. This will happen on large-distance links or source hosts with small bandwidth. But on a segmented 100 MBit ethernet LAN with one or two routers there will likely never be any EDCN messages.

5 Conclusion and Outlook

Below the conclusion of this work is presented. Some aspects of possible further proceeding will be outlined.

5.1 Conclusion

The tests show that saving bandwidth and processing time is possible by using EDC. In both tests a saving of nearly 5% was reached.

As real-time interactive multimedia applications take up more and more space in the internet the inclusion of EDC features in routers and such applications makes sense.

Internet providers and big IP carriers might be interested in EDC technology. While many "internet end users" have a flat rate these companies must pay for the amount of data which leaves their own network. Therefore they are interested in any form of bandwidth saving.

5.2 Outlook

The active components of Internet infrastructure do not only run on Linux Operating System. Many routers are from Cisco and therefore run on Cisco IOS. In order to take an advantage of EDC the majority of routers should be EDC compatible. So EDC should be ported to the Cisco IOS and possibly other router operating systems. Microsoft Windows does not play a role here because it mainly runs on workstations and not routers.

Before starting the process of standardization of EDC it should be taken into consideration to use the more precise MTD definition pointed out in "Possible Improvements".

5.3 Possible Improvements

During work on this implementation of EDC two possible improvements became obvious. The first point "More precise MTD" has not been implemented because the initial specification in [1] should not be changed. The second point "Determine Location based on IP

Address" requires a connection to the Internet to query the online service. As the test environment consists of virtual hosts an Internet connection would be difficult. Furthermore the test environment uses private IP adresses which are not locateable.

5.3.1 More precice MTD

The MTD is actually stored in milliseconds within a single byte. This allows to store values from 0 to 255 with no decimal places.

Unfortunately IP options must be 32 bit aligned and EDC needs 4 bytes to store all necessary information. IP options must end with the "Option-End" opcode 0. Therefore the number of bytes needed for IP options containing only EDC is 5. The next 32 bit boundary would be 8 bytes which makes it necessary to fill in three padding bytes (Picture 5.1). This is a waste of space.

Option Code Content: 14 (EDC)	Option Length Content: 4 (4 Bytes)	EDC TTL Content: variable	EDC MTD Content: variable
Option End Content: 0 (End)	Padding Content: 0 (Fill Byte)	Padding Content: 0 (Fill Byte)	Padding Content: 0 (Fill Byte)

Picture 5.1: Actual structure of IP options with only EDC used

A possible improvement could be to raise the resolution of the MTD Field from one to four bytes which would make it possible to store a range of 2^{32} instead of 2^8 values. The MTD could be stored as nanoseconds instead of milliseconds which would be a very significant resolution improvement. The size of IP options would not raise (Picture 5.2) assuming that EDC is the only option used.

5.3.2 Determine Location based on IP Address

Determining locations based on user data might be uncertain. The user could enter wrong data accidently or on purpose. Another point of uncertainty might be multiple occurences of one city name (e.g. "Hamburg" exists 10 times).

The company "Geobytes Inc." [2] (possibly others too) offers a service with which it is possible to find the region where an IP address is located. This can be used to estimate distance and

Option Code Content: 14 (EDC)	Option Length Content: 4 (4 Bytes)	EDC TTL Content: variable	EDC MTD Content: variable
EDC MTD Content: variable	EDC MTD Content: variable	EDC MTD Content: variable	Option End Content: 0 (End)

Picture 5.2: Possible improvement: better accuracy of MTD and no padding bytes

propagation delay without needing further data from the user.

It must be kept in mind to achieve the correct IP address in cases where server or client are located behind a router which does Network Address Translation (NAT). An often used private IP address behind such a router can not be located.

6 Appendix

The source code of the test programs and the changes within the Linux Kernel in ”diff” format.

6.1 server.c

```
/*
server.c -- Test Server for EDC
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define MYPORT 4711             // the port users will be connecting to

#define MAXBUFLEN 1024

int stopit;

void handler(int signal)
{
  printf("\nReceived SIGINT ... aborting !\n");
  stopit = 1;
}

int getpropdelay(char *city1, char *city2)
{
```

```
FILE *pipe;
fd_set pipefds;
struct timeval timeout;
char command[200];
char buffer[1000];
int fd;
int distance, propdelay;

sprintf(command, "/usr/bin/perl distance.pl \"%s\" \"%s\" 2>&1",
        city1, city2);

/* open pipe, in case of error return 0=no delay */
if ((pipe = popen(command, "r")) == NULL) {
  printf("Could not open pipe !\n");
  return 0;
}

fd = fileno(pipe);
FD_ZERO(&pipefds);
FD_SET(fd, &pipefds);
timeout.tv_sec = 5;           /* Wait 5 seconds for external programm */
timeout.tv_usec = 0;

if (select(fd + 1, &pipefds, NULL, NULL, &timeout) == -1) {
  perror("select");
  exit(1);
}
if (FD_ISSET(fd, &pipefds)) {
  /* read distance or error / not found */
  fgets(buffer, 999, pipe);
  if ((strncmp(buffer, "fileerror", 9) == 0)
      || (strncmp(buffer, "not found", 9) == 0)) {
    printf("Error-Case ... %s\n", buffer);
    return 0;
  }
  distance = strtol(buffer, NULL, 10);
  /* when we get here the second line is propagation delay */
  fgets(buffer, 999, pipe);
  fclose(pipe);
  propdelay = strtol(buffer, NULL, 10);
  printf("%s <--> %s\n", city1, city2);
  printf("Distance=%u    Propagation delay=%u\n", distance,
         propdelay);
  return propdelay;
```

```
  }
  /* Timeout */
  fclose(pipe);
  return 0;
}


int main(int argc, char *argv[])
{

  int sockfd;
  fd_set readfds, master;
  struct sockaddr_in my_addr;   // my address information
  struct sockaddr_in their_addr;        // connector's address information
  int addr_len, numbytes;
  struct timeval timeout;
  char buf[MAXBUFLEN];
  char city[100];
  int propdelay;
  unsigned char pd;

  if (argc != 2) {
    printf("Usage: %s Name_of_City\n", argv[0]);
    exit(1);
  }



  stopit = 0;

  signal(SIGINT, handler);

  if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
    perror("socket");
    exit(1);
  }

  my_addr.sin_family = AF_INET; // host byte order
  my_addr.sin_port = htons(MYPORT);     // short, network byte order
  my_addr.sin_addr.s_addr = INADDR_ANY; // automatically fill with my IP
  memset(&(my_addr.sin_zero), '\0', 8); // zero the rest of the struct

  if (bind(sockfd, (struct sockaddr *) &my_addr,
           sizeof(struct sockaddr)) == -1) {
```

```
  perror("bind");
  exit(1);
}

addr_len = sizeof(struct sockaddr);


FD_ZERO(&readfds);
FD_ZERO(&master);
FD_SET(sockfd, &master);
readfds = master;
/* 2 Sec Timeout */
timeout.tv_sec = 2;
timeout.tv_usec = 0;

while (!stopit) {
  if (select(sockfd + 1, &readfds, NULL, NULL, &timeout) == -1) {
    perror("select");
    exit(1);
  }
  if (FD_ISSET(sockfd, &readfds)) {
    if ((numbytes = recvfrom(sockfd, buf, MAXBUFLEN - 1, 0,
                             (struct sockaddr *) &their_addr,
                             &addr_len)) == -1) {
      perror("recvfrom");
      exit(1);
    }
    if (strncmp(buf, "getpropdelay:", 13) == 0) {
      buf[numbytes] = 0;
      strncpy(city, &buf[13], 99);
      city[99] = 0;
      printf
          ("Search for distance / propagation delay between %s and %s ...\n",
           argv[1], city);
      propdelay = getpropdelay(argv[1], city);
      if (propdelay > 255)
        propdelay = 255;
      pd = (unsigned char) propdelay;
      if ((sendto
           (sockfd, &pd, 1, 0, (struct sockaddr *) &their_addr,
            addr_len)) == -1) {
        perror("sendto");
        exit(1);
      }
```

```
      }
      printf("got packet from %s\n", inet_ntoa(their_addr.sin_addr));
      printf("packet is %d bytes long\n", numbytes);
      buf[numbytes] = '\0';
      printf("packet contains \"%s\"\n", buf);
    }

    readfds = master;
    timeout.tv_sec = 2;
    timeout.tv_usec = 0;

  }

  close(sockfd);

  return 0;

}
```

6.2 udptest.c

```
#include <stdio.h>
#include <string.h>
#include <signal.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <netinet/udp.h>
#include <netdb.h>
#include <errno.h>

#define h_addr h_addr_list[0]

#define DEST_PORT 4711

#define COPY 128
#define CLASS_CONTROL 0
```

```
#define CLASS_DEBUG_MEASURE 64
#define INITIAL_MTD 100



/* global variables */
int stopit, edcchange;
unsigned int orig_ttl;          /* original TTL of packets sent from us      */
unsigned int rtt;
unsigned long int edccount;
unsigned int delta, lambda;     /* for adaption of estimations of prop_delay */
                                /* and trans_delay                           */
unsigned int est_prop_delay, est_trans_delay;
unsigned int est_prop_delay_thd, est_trans_delay_thd;

struct timeval edctv[10];       /* last 10 times where ICMP-EDC-Packets arrived */
struct timeval last_adapt;

/* handle Interrupt signal */
void handler(int signal)
{
  printf("\nReceived SIGINT ... aborting !\n");
  stopit = 1;
}

/* timediff_ms calculates the no. of milliseconds between two timeval values */
/* time2 should be bigger than time1                                         */
unsigned long timediff_ms(struct timeval *time1,
                          struct timeval *time2)
{
  unsigned long result;
  result =
      ((time2->tv_sec - time1->tv_sec) * 1000000 + time2->tv_usec -
       time1->tv_usec) / 1000;
  printf("timediff result=%u\n", result);
  return result;
}



/* EDC Delay adaption routine                                                */
/* ttl contains the TTL field from EDCN. So we see at which hop the packet died. */
int edcadapt(unsigned char ttl)
{
  /* wait for min. 4 RTTs since last adaption */
```

```
if (last_adapt.tv_sec == 0
    || (timediff_ms(&last_adapt, &edctv[edccount % 10]) >
        (4 * rtt))) {

  /* now at least 3 EDCN must be received within 4*RTT to trigger adaption */
  if (timediff_ms
      (&edctv[(edccount - 2) % 10],
       &edctv[edccount % 10]) > (4 * rtt)) {
    /* smaller than 4*RTT -> abort */
    printf("last 3 EDCN Packets not within 4*RTT\n");
    return 0;
  }

  /* check whether to adapt est_prop_delay or est_trans_delay */
  if ((orig_ttl - ttl) < 10) {
    /* less than 10 hops -> adapt est_prop_delay ! */
    delta = 0;
    lambda = lambda + 25;
    /* test if new est_prop_delay > est_prop_delay_thd */
    if (((est_prop_delay * 100) / (100 + lambda)) <
        est_prop_delay_thd) {
      lambda = (est_prop_delay_thd / est_prop_delay) - 1;
      printf("Threshold reached !!!\n");
    }
    printf("\nChange Prop delay: new lambda %u\n", lambda);
  } else {
    /* more or equal 10 hops -> adapt est_trans_delay ! */
    lambda = 0;
    delta = delta + 0.25;
    /* test if new est_trans_delay > est_trans_delay_thd */
    if ((est_trans_delay * 100 / (100 + delta)) <
        est_trans_delay_thd) {
      delta = (est_trans_delay_thd / est_trans_delay) - 1;
    }
    printf("\nChange Trans delay: new delta %u\n", delta);
  }
  /* save time of last adaption */
  gettimeofday(&last_adapt, NULL);
  printf("Last adapt: %u.%u\n", last_adapt.tv_sec,
         last_adapt.tv_usec);

  /* set adaption marker */
  edcchange = 1;
}
```

```
  return 0;
}


/* decode_icmp gets all inbound ICMP-Packets and tests for ICMP-EDCN-Packets */
/* related to our UDP-Socket connection                                      */
int decode_icmp(char *buf, int len, struct sockaddr_in *from,
                struct sockaddr_in *to)
{
  struct ip *ip;
  struct icmp *icp;
  struct udphdr *up;
  int hlen, icphlen;
  unsigned char ip_ttl;

  ip = (struct ip *) buf;
  hlen = ip->ip_hl << 2;

  len -= hlen;
  icp = (struct icmp *) (buf + hlen);
  // up = (struct udphdr *) (buf+hlen+sizeof(struct icmp));

  /* ICMP Time Exceeded ? */
  if (icp->icmp_type != ICMP_TIME_EXCEEDED) {
    // printf("/");
    return 0;
  }
  /* ICMP Time Exceeded Code 2 */
  if (icp->icmp_code != 2) {
    // printf("/");
    return 0;
  }

  /* here we have an ICMP-EDC-Packet. Is it ours ? */
  icphlen = icp->icmp_dun.id_ip.idi_ip.ip_hl << 2;
  up = (struct udphdr *) (buf + hlen + icphlen +
                          sizeof(struct icmphdr));

  /* Check source- and destination port */
  if (up->source != from->sin_port) {
    // printf("/");
    return 0;
  }
```

```
  if (up->dest != to->sin_port) {
    // printf("/");
    return 0;
  }

  /* check source and destination address */
  if ((icp->icmp_dun.id_ip.idi_ip.ip_src.s_addr) !=
      (from->sin_addr.s_addr)) {
    return 0;
  }
  if ((icp->icmp_dun.id_ip.idi_ip.ip_dst.s_addr) !=
      (to->sin_addr.s_addr)) {
    return 0;
  }

  /* Yes, this is an ICMP-EDCN-packet related to our UDP-connection */
  printf("+");
  edccount++;

  /* save system time */
  gettimeofday(&edctv[edccount % 10], NULL);

  /* test if est_prop_delay or est_tran_delay need adaption   */
  /* save received TTL so we know @ which hop the packet died */
  ip_ttl = icp->icmp_dun.id_ip.idi_ip.ip_ttl;

  /* start adaption if possible (enough EDCN packets received) */
  if (edccount > 2) {
    edcadapt(ip_ttl);
  }

  return 0;
}


/* getrtt tries to find round trip time (RTT) from client to server. It calls     */
/* external program "rttometer" which measures RTT via TCP, not ICMP Echo Request */
int getrtt(char *hostname)
{
  FILE *pipe;
  unsigned int drtt;
  fd_set pipefds;
  struct timeval timeout;
  char command[200];
```

```
char buffer[1000];
int fd;

sprintf(command, "/usr/local/sbin/rttometer %s 2>&1", hostname);

/* open pipe, in case of error return 0=no delay */
if ((pipe = popen(command, "r")) == NULL) {
  printf("Could not open pipe !\n");
  return 0;
}

drtt = 0;

fd = fileno(pipe);
FD_ZERO(&pipefds);
FD_SET(fd, &pipefds);
timeout.tv_sec = 10;          /* Wait 5 seconds for external programm */
timeout.tv_usec = 0;

while (!feof(pipe)) {
  if (select(fd + 1, &pipefds, NULL, NULL, &timeout) == -1) {
    perror("select");
    exit(1);
  }
  if (FD_ISSET(fd, &pipefds)) {
    /* read one line of output from rttometer */
    fgets(buffer, 999, pipe);
    if (strncmp(buffer, "Min RTT:", 8) == 0) {
      printf("Found min RTT ... \n%s\n", buffer);
      printf("Buffer[9]:%s\n", &buffer[9]);
      drtt = atoi(&buffer[9]);
      break;
    }
    FD_ZERO(&pipefds);
    FD_SET(fd, &pipefds);
    timeout.tv_sec = 10;      /* Wait 10 seconds for external programm */
    timeout.tv_usec = 0;
  } else {
    /* Timeout */
    break;
  }
}
fclose(pipe);
printf("RTT=%u\n", drtt);
```

```
  return drtt;
  // return 80;
}



int main(int argc, char *argv[])
{
  unsigned long i, j;
  int sockfd, icmpfd;
  long flags;
  struct sockaddr_in dest_addr, sin, srcaddr;
  struct hostent *hostaddr;
  char buffer[256];
  char icmpbuf[4096];           /* 4k Buffer for ICMP (enough?)   */
  int icmpread;
  unsigned long int count;
  unsigned long int waitusec;
  unsigned char tos;
  unsigned char ttl;
  double mtd;
  unsigned int sourceport = 0;
  unsigned int packetsize;
  unsigned int packets_per_second;
  socklen_t len, srclen;
  struct ip *ippkt;
  struct icmp *icmppkt;
  unsigned long bandwidth;
  struct timeval timeout;
  fd_set readfds;
  int numbytes, addr_len;

  /* Testdata to send over network */
  struct {
    char text[250];
    unsigned long int count;
  } data;

  /* IP-Options */
  struct {
    unsigned char optioncode;
    unsigned char optionlength;
    unsigned char ext_ttl;
    unsigned char mtd;          /* maximum tolerable delay */
```

```
  unsigned char option_end;
  unsigned char padding[3];
} myoptions;

if (argc != 5) {
  printf
      ("Usage: %s hostname own_location upstream_bandwidth profile\n\n",
       argv[0]);
  exit(1);
}

/* Initialize compensation factors */
delta = 0;
lambda = 0;

/* set last adaption time to 0 */
last_adapt.tv_sec = 0;
last_adapt.tv_usec = 0;

if (strncasecmp(argv[4], "video", 5) == 0) {
  /* Profile "Video":   25 Packets (Frames) of 200 Bytes per Second */
  packetsize = 200;
  packets_per_second = 25;
} else if (strncasecmp(argv[4], "game", 4) == 0) {
  /* Profile "Game":    20 Packets of 75 Bytes per Second (UT2004, client to server) */
  packetsize = 75;
  packets_per_second = 20;
} else {
  /* Default Profile "Audio":   23 Packets of 136 Bytes per Second  */
  /* (Skype-Session, client to server) */
  packetsize = 136;
  packets_per_second = 23;
}

waitusec = 1000000 / packets_per_second;


/* name res */
if ((hostaddr = gethostbyname(argv[1])) == NULL) {
  perror("gethostbyname");
  exit(1);
}

printf("Hostname       : %s\n", hostaddr->h_name);
```

```
printf("IP-Address     : %s\n",
       inet_ntoa(*((struct in_addr *) hostaddr->h_addr)));


rtt = getrtt(argv[1]);
printf("using %u as RTT value\n", rtt);

memset(&myoptions, '\0', sizeof(myoptions));

myoptions.optioncode = 14;    /* festgelegt in Doku */
myoptions.optionlength = 4;
myoptions.mtd = INITIAL_MTD;
myoptions.option_end = '\0';


count = 0;
stopit = 0;

/* set signal handler */
signal(SIGINT, handler);


/* create UDP socket */
sockfd = socket(AF_INET, SOCK_DGRAM, 0);

/* Data structure for source port. */
srcaddr.sin_family = AF_INET;
srcaddr.sin_port = INADDR_ANY;
srcaddr.sin_addr.s_addr = INADDR_ANY;
memset(&(srcaddr.sin_zero), '\0', 8);
srclen = sizeof(struct sockaddr_in);


dest_addr.sin_family = AF_INET;
dest_addr.sin_port = htons(DEST_PORT);
memcpy(&(dest_addr.sin_addr.s_addr), hostaddr->h_addr,
       hostaddr->h_length);
memset(&(dest_addr.sin_zero), '\0', 8);

/* Connect Socket for finding our source IP later */
if ((connect
     (sockfd, (struct sockaddr *) &dest_addr,
      sizeof(struct sockaddr_in))) < 0) {
  perror("Connect");
```

```
  exit(1);
}


/* retrieve TOS and TTL */
len = sizeof(tos);
getsockopt(sockfd, SOL_IP, IP_TOS, &tos, &len);
getsockopt(sockfd, SOL_IP, IP_TTL, &ttl, &len);

/*  printf("DEBUG: len after getsockopt: %u    TOS = %u\n",len,tos); */

/* add IPTOS_LOWDELAY for multimedia application */
tos = tos | IPTOS_LOWDELAY;

/* set IP-Options TTL to real TTL of IP Packet */
myoptions.ext_ttl = ttl;

orig_ttl = ttl;

/* find est. propagation delay and calculate est. transmission delay */
bandwidth = strtol(argv[3], NULL, 10);

/* assume packet_size=packetsize and calculate delay in ms */
/* 20+8+8 -> IP-Header + Options + UDP-Header */
est_trans_delay =
    ((packetsize + 20 + 8 + 8) * 8 * 1000) / bandwidth;

/* send request to server */
FD_ZERO(&readfds);
FD_SET(sockfd, &readfds);
timeout.tv_sec = 10;          /* 10 sec timeout */
timeout.tv_usec = 0;

/* send our location to the server */
snprintf(buffer, 255, "getpropdelay:%s", argv[2]);
if ((sendto
     (sockfd, buffer, strlen(buffer), 0,
      (struct sockaddr *) &dest_addr,
      sizeof(struct sockaddr))) == -1) {
  perror("sendto ... getpropdelay");
  exit(1);
}

/* wait max. 10 sec for an answer or timeout */
```

```
if (select(sockfd + 1, &readfds, NULL, NULL, &timeout) == -1) {
  perror("select");
  exit(1);
}
if (FD_ISSET(sockfd, &readfds)) {
  if ((numbytes =
       recvfrom(sockfd, buffer, 255, 0,
                (struct sockaddr *) &dest_addr, &addr_len)) == -1) {
    perror("recvfrom");
    exit(1);
  }
  if (numbytes == 1) {
    est_prop_delay = buffer[0];
  } else {
    /* we got junk so we assume zero ... */
    printf("junk received ...\n");
    est_prop_delay = 0;
  }
} else {
  /* timeout, so we assume propagation delay = 0 */
  est_prop_delay = 0;
}

/* calculate MTD */
mtd = (double) (INITIAL_MTD - (est_prop_delay + est_trans_delay));
printf
    ("Initial MTD=%ums ,est. prop. delay=%ums ,est trans. delay=%u  => resulting MTD=%f\n",
     INITIAL_MTD, est_prop_delay, est_trans_delay, mtd);

/* set threshold values to 50% of initial values */
est_prop_delay_thd = est_prop_delay / 2;
est_trans_delay_thd = est_trans_delay / 2;
printf("Thresholds:\n");
printf("Propagation delay : %u\nTransmission delay : %u\n",
       est_prop_delay_thd, est_trans_delay_thd);

/* set calculated MTD into IP Options structure */
myoptions.mtd = (unsigned char) mtd;

/* set new TOS (lowdelay) for our socket */
if (setsockopt(sockfd, SOL_IP, IP_TOS, &tos, sizeof(tos)) != 0) {
  printf("setsockopt IP_TOS failed. errno=%u\n", errno);
}
```

```
/* set new IP Options for our socket */
if (setsockopt
    (sockfd, SOL_IP, IP_OPTIONS, &myoptions,
     sizeof(myoptions)) != 0) {
  perror("setsockopt IP_OPTIONS failed.");
}

/* clear test data */
memset(&data, '\0', sizeof(data));


/* create ICMP RAW Socket for later receiving of ICMP-EDC Packets */
if ((icmpfd = socket(PF_INET, SOCK_RAW, IPPROTO_ICMP)) < 0) {
  perror("ICMP-socket");
  exit(1);
}

/* prepare sockaddr_in for ICMP */
bzero((char *) &sin, sizeof(sin));
sin.sin_port = INADDR_ANY;

/* bind ICMP RAW socket */
if ((bind(icmpfd, (struct sockaddr *) &sin, sizeof(sin))) < 0) {
  perror("ICMP-bind");
  exit(1);
}


/* set ICMP RAW Socket to Nonblocking                  */
/* we do not want to wait if there are no ICMP Packets */
flags = (long) fcntl(icmpfd, F_GETFL, 0);
if (fcntl(icmpfd, F_SETFL, flags | O_NONBLOCK) < 0) {
  perror("ICMP fcntl nonblock");
  exit(1);
}


/* send packets while stopit==0                                     */
/* signal handler sets stopit=1 when process receives SIGINT signal */
while (!stopit) {
  sprintf(data.text, "Hello World ... %u", count);
  data.count = count;
  sendto(sockfd, &data, packetsize, 0,
         (struct sockaddr *) &dest_addr, sizeof(struct sockaddr));
```

```
printf(".");

/* read from ICMP socket until all data is read */
do {
  if ((icmpread = read(icmpfd, icmpbuf, 4096)) < 0) {
    /* test for other errors than EAGAIN */
    if (errno != EAGAIN) {
      perror("ICMPRead");
      exit(1);
    }
  } else {
    /* get source IP and source port */
    if (srcaddr.sin_port == INADDR_ANY) {
      if ((getsockname
           (sockfd, (struct sockaddr *) &srcaddr, &srclen)) < 0) {
        perror("getsockname");
        exit(1);
      }
    }
    /* decode ICMP packet. Due to raw socket we get ALL ICMP packets */
    decode_icmp(icmpbuf, icmpread, &srcaddr, &dest_addr);
    // printf ("ICMP-Paket empfangen. Größe: %u\n", icmpread);
  }
} while (icmpread > -1);

/* adapt values if necessary */
if (edcchange == 1) {
  edcchange = 0;
  /* calculate new MTD using compensation factors */
  mtd =
      INITIAL_MTD - ((est_prop_delay * 100 / (100 + lambda)) +
                     (est_trans_delay * 100 / (100 + delta)));
  /* set new IP Options */
  myoptions.mtd = (unsigned char) mtd;
  if (setsockopt
      (sockfd, SOL_IP, IP_OPTIONS, &myoptions,
       sizeof(myoptions)) != 0) {
    perror("setsockopt IP_OPTIONS failed.");
  }
}


/* flush STDOUT */
fflush(NULL);
```

```
    /* wait calculated no. of microseconds */
    usleep(waitusec);
    count++;
  }

  /* close sockets */
  printf("Closing Socket ! BYE !\n\n");
  close(sockfd);
  close(icmpfd);

  /* statistics */
  printf("Statistics:\n");
  printf("===========\n");
  printf("Sent packets          : %u\n", count);
  printf("Received EDC-Packets : %u\n\n", edccount);
  printf("Last 10 EDC-Packets received at:\n");
  if (edccount < 10) {
    j = edccount;
  } else {
    j = 10;
  }
  for (i = edccount; i > edccount - j; i--) {
    printf("%u.%u\n", edctv[i % 10].tv_sec, edctv[i % 10].tv_usec);
  }
  return 0;

}
```

6.3 distance.pl

```
#!/usr/bin/perl

$CITIES="//root//geodata//Cities.txt";

# Radius Earth in km
$radius=6378;

# Lightspeed in km/s
# Media: Glass
$lightspeed=200000;
```

```
# define pi as 3.14.....
$pi = atan2(1,1) * 4;

$f1=0;
$f2=0;

# subroutine acos
#
# input: an angle in radians
#
# output: returns the arc cosine of the angle
#
# description: this is needed because perl does not provide an arc cosine function

sub acos {
    my($x) = @_;
    my $ret = atan2(sqrt(1 - $x**2), $x);
    return $ret;
}


open(CF,"< $CITIES") or die "fileerror";
while (defined ($line = <CF>) && ($f1==0 || $f2==0)) {
    chomp $line;
    if( !$f1 && $line =~ m/,\"$ARGV[0]\",/ ) {
        @array=split( /,/ , $line);
        # Name of City
        $array[3] =~ s/\"//g;
        # Latitude / Breite
        $array[4] =~ s/\"//g;
        $a1=$array[4]*($pi/180);
        # Longitude / Länge
        $array[5] =~ s/\"//g;
        $b1=$array[5]*($pi/180);
        # print ("$array[3]   $array[4]   $array[5]   $a1   $b1\n");
        $f1=1;
    }
    elsif ( !$f2 && $line =~ m/,\"$ARGV[1]\",/ ) {
        @array=split( /,/ , $line);
        # Name of City
        $array[3] =~ s/\"//g;
        # Latitude / Breite
        $array[4] =~ s/\"//g;
        $a2=$array[4]*($pi/180);
```

```
        # Longitude / Länge
        $array[5] =~ s/\"//g;
        $b2=$array[5]*($pi/180);
        # print ("$array[3]   $array[4]   $array[5]   $a2   $b2\n");
        $f2=1;
    }
}

if($f1==1 && $f2==1) {
    $distance=&acos(cos($a1)*cos($b1)*cos($a2)*cos($b2) + \
    cos($a1)*sin($b1)*cos($a2)*sin($b2) + sin($a1)*sin($a2)) * $radius;
    print "$distance\n";
    $propdelay=$distance / ($lightspeed / 1000);
    print "$propdelay\n";
}
else {
    print "not found\n";
}
```

6.4 Changes within the Linux Kernel (diff-format)

```
diff -urN -X dontdiff linux-2.4.27-umlorig/include/linux/icmp.h linux-2.4.27-uml/include/linux/icmp.h
--- linux-2.4.27-umlorig/include/linux/icmp.h 2001-04-12 21:11:39.000000000 +0200
+++ linux-2.4.27-uml/include/linux/icmp.h 2004-12-28 12:32:49.000000000 +0100
@@ -61,7 +61,7 @@
 /* Codes for TIME_EXCEEDED. */
 #define ICMP_EXC_TTL           0       /* TTL count exceeded          */
 #define ICMP_EXC_FRAGTIME      1       /* Fragment Reass time exceeded */
-
+#define ICMP_EXC_EDC           2       /* EDC Time exceeded           */

 struct icmphdr {
   __u8 type;
diff -urN -X dontdiff linux-2.4.27-umlorig/include/linux/inetdevice.h linux-2.4.27-uml/include/linux/inetdevice.h
--- linux-2.4.27-umlorig/include/linux/inetdevice.h 2004-04-14 15:05:40.000000000 +0200
+++ linux-2.4.27-uml/include/linux/inetdevice.h 2004-12-28 12:32:49.000000000 +0100
@@ -21,7 +21,9 @@
        int     arp_announce;
        int     arp_ignore;
        int     medium_id;
-       int     force_igmp_version;
+        int    force_igmp_version;
+        int     prop_delay;
+        long    real_speed;
        void    *sysctl;
 };

diff -urN -X dontdiff linux-2.4.27-umlorig/include/linux/ip.h linux-2.4.27-uml/include/linux/ip.h
--- linux-2.4.27-umlorig/include/linux/ip.h 2004-12-29 21:09:15.000000000 +0100
+++ linux-2.4.27-uml/include/linux/ip.h 2004-12-28 12:32:49.000000000 +0100
@@ -61,6 +61,8 @@
 #define IPOPT_RR       (7 |IPOPT_CONTROL)
 #define IPOPT_SID      (8 |IPOPT_CONTROL|IPOPT_COPY)
```

```
 #define IPOPT_SSRR     (9 |IPOPT_CONTROL|IPOPT_COPY)
+/* Explicit delay control */
+#define IPOPT_EDC       (14|IPOPT_CONTROL)
 #define IPOPT_RA       (20|IPOPT_CONTROL|IPOPT_COPY)

 #define IPVERSION      4
@@ -105,8 +107,10 @@
                 ts_needtime:1,                /* Need to record timestamp             */
                 ts_needaddr:1;                /* Need to record addr of outgoing dev  */
   unsigned char router_alert;
-  unsigned char __pad1;
-  unsigned char __pad2;
+  unsigned char edc_ttl;                       /* EDC TTL-Field                        */
+  unsigned char edc_mtd;                       /* EDC Maximum tolerable delay          */
+/*  unsigned char __pad1;  */
+/*  unsigned char __pad2;  */
   unsigned char __data[0];
 };

diff -urN -X dontdiff linux-2.4.27-umlorig/include/linux/sysctl.h linux-2.4.27-uml/include/linux/sysctl.h
--- linux-2.4.27-umlorig/include/linux/sysctl.h 2004-12-29 21:08:40.000000000 +0100
+++ linux-2.4.27-uml/include/linux/sysctl.h 2004-12-28 12:39:24.000000000 +0100
@@ -375,6 +375,8 @@
        NET_IPV4_CONF_FORCE_IGMP_VERSION=17,
        NET_IPV4_CONF_ARP_ANNOUNCE=18,
        NET_IPV4_CONF_ARP_IGNORE=19,
+       NET_IPV4_CONF_PROP_DELAY=20,
+       NET_IPV4_CONF_REAL_SPEED=21,
 };

 /* /proc/sys/net/ipv4/netfilter */
@@ -837,3 +839,39 @@
 #endif /* __KERNEL__ */

 #endif /* _LINUX_SYSCTL_H */
diff -urN -X dontdiff linux-2.4.27-umlorig/net/ipv4/devinet.c linux-2.4.27-uml/net/ipv4/devinet.c
--- linux-2.4.27-umlorig/net/ipv4/devinet.c 2004-08-08 01:26:06.000000000 +0200
+++ linux-2.4.27-uml/net/ipv4/devinet.c 2004-12-28 12:32:49.000000000 +0100
@@ -1151,7 +1151,7 @@
 static struct devinet_sysctl_table
 {
        struct ctl_table_header *sysctl_header;
-       ctl_table devinet_vars[20];
+       ctl_table devinet_vars[22];
        ctl_table devinet_dev[2];
        ctl_table devinet_conf_dir[2];
        ctl_table devinet_proto_dir[2];
@@ -1209,6 +1209,12 @@
         {NET_IPV4_CONF_FORCE_IGMP_VERSION, "force_igmp_version",
          &ipv4_devconf.force_igmp_version, sizeof(int), 0644, NULL,
          &proc_dointvec},
+        {NET_IPV4_CONF_PROP_DELAY, "propagation_delay",
+         &ipv4_devconf.prop_delay, sizeof(int), 0644, NULL,
+         &proc_dointvec},
+        {NET_IPV4_CONF_REAL_SPEED, "real_speed",
+         &ipv4_devconf.real_speed, sizeof(long), 0644, NULL,
+         &proc_dointvec},
         {0}},

         {{NET_PROTO_CONF_ALL, "all", NULL, 0, 0555, devinet_sysctl.devinet_vars},{0}},
diff -urN -X dontdiff linux-2.4.27-umlorig/net/ipv4/icmp.c linux-2.4.27-uml/net/ipv4/icmp.c
--- linux-2.4.27-umlorig/net/ipv4/icmp.c 2004-04-14 15:05:41.000000000 +0200
+++ linux-2.4.27-uml/net/ipv4/icmp.c 2004-12-28 12:32:49.000000000 +0100
@@ -496,6 +498,9 @@
        room = rt->u.dst.pmtu;
        if (room > 576)
                room = 576;
+       /* for bandwidth saving @ EDC Time exceeded return only IP-hdr + 8 data bytes */
+       if (type==ICMP_TIME_EXCEEDED && code==ICMP_EXC_EDC)
+               room=72;
        room -= sizeof(struct iphdr) + icmp_param.replyopts.optlen;
        room -= sizeof(struct icmphdr);

diff -urN -X dontdiff linux-2.4.27-umlorig/net/ipv4/ip_forward.c linux-2.4.27-uml/net/ipv4/ip_forward.c
--- linux-2.4.27-umlorig/net/ipv4/ip_forward.c 2001-04-12 21:11:39.000000000 +0200
```

```
+++ linux-2.4.27-uml/net/ipv4/ip_forward.c 2004-12-28 12:32:49.000000000 +0100
@@ -40,6 +40,7 @@
 #include <net/checksum.h>
 #include <linux/route.h>
 #include <net/route.h>
+#include <linux/inetdevice.h>

 static inline int ip_forward_finish(struct sk_buff *skb)
 {
@@ -77,6 +78,9 @@
        struct rtable *rt;        /* Route we use */
        struct ip_options * opt = &(IPCB(skb)->opt);
        unsigned short mtu;
+        unsigned char edc_mtd;
+        unsigned char *ihdr;
+        struct in_device *in_dev;

        if (IPCB(skb)->opt.router_alert && ip_call_ra_chain(skb))
                return NET_RX_SUCCESS;
@@ -121,6 +130,24 @@
        if (skb_cow(skb, dev2->hard_header_len))
                goto drop;
        iph = skb->nh.iph;
+        ihdr = skb->nh.raw;
+        opt=&(IPCB(skb)->opt);
+
+        if(opt->edc_mtd>0) {
+                if ((in_dev = __in_dev_get(dev2)) != NULL) {
+                        edc_mtd=(unsigned char) *(ihdr+opt->edc_mtd);
+                        /* printk("edc_mtd before: %u\n",edc_mtd);  */
+                        if (edc_mtd != 0) {
+                                if (in_dev->cnf.prop_delay > 255 || in_dev->cnf.prop_delay > edc_mtd) {
+                                        goto edc_time_exceeded;
+                                }
+                        edc_mtd = edc_mtd - in_dev->cnf.prop_delay;
+                        (unsigned char) *(ihdr+opt->edc_mtd)=edc_mtd;
+                        opt->is_changed=1;
+                        /*  printk("edc_mtd after: %u\n",edc_mtd);  */
+                        }
+                }
+        }

        /* Decrease ttl after skb cow done */
        ip_decrease_ttl(iph);
@@ -157,6 +184,11 @@
          icmp_send(skb, ICMP_DEST_UNREACH, ICMP_SR_FAILED, 0);
          goto drop;

+edc_time_exceeded:
+        /* Tell the sender its packet died... */
+        icmp_send(skb, ICMP_TIME_EXCEEDED, ICMP_EXC_EDC, 0);
+        goto drop;
+
 too_many_hops:
         /* Tell the sender its packet died... */
         icmp_send(skb, ICMP_TIME_EXCEEDED, ICMP_EXC_TTL, 0);
diff -urN -X dontdiff linux-2.4.27-umlorig/net/ipv4/ip_options.c linux-2.4.27-uml/net/ipv4/ip_options.c
--- linux-2.4.27-umlorig/net/ipv4/ip_options.c 2002-11-29 00:53:15.000000000 +0100
+++ linux-2.4.27-uml/net/ipv4/ip_options.c 2004-12-28 12:32:49.000000000 +0100
@@ -424,6 +424,16 @@
                                }
                        }
                        break;
+                      case IPOPT_EDC:
+                        if (optlen != 4) {
+                          pp_ptr = optptr +1;
+                          goto error;
+                        } else {
+                          opt->edc_ttl=optptr - iph + 2;
+                          opt->edc_mtd=optptr - iph + 3;
+                           /* printk("IPOPT_EDC: ttl=%u , ms=%u\n",(unsigned char) *(iph+opt->edc_ttl), (unsigned char) *(iph+opt->edc_mtd));  */
+                        }
+                        break;
                      case IPOPT_RA:
                        if (optlen < 4) {
```

```
                                pp_ptr = optptr + 1;
@@ -617,3 +627,31 @@
        }
        return 0;
 }
diff -urN -X dontdiff linux-2.4.27-umlorig/net/ipv4/ip_output.c linux-2.4.27-uml/net/ipv4/ip_output.c
--- linux-2.4.27-umlorig/net/ipv4/ip_output.c 2003-11-28 19:26:21.000000000 +0100
+++ linux-2.4.27-uml/net/ipv4/ip_output.c 2005-01-01 17:02:55.000000000 +0100
@@ -546,6 +553,10 @@
                skb->dst = dst_clone(&rt->u.dst);
                skb_reserve(skb, hh_len);

+                /* EDC: write packet creation time to skb */
+                if (skb->stamp.tv_sec == 0)
+                        do_gettimeofday(&skb->stamp);
+
                /*
                 *      Find where to start putting bytes.
                 */
diff -urN -X dontdiff linux-2.4.27-umlorig/net/sched/sch_generic.c linux-2.4.27-uml/net/sched/sch_generic.c
--- linux-2.4.27-umlorig/net/sched/sch_generic.c 2004-02-18 14:36:32.000000000 +0100
+++ linux-2.4.27-uml/net/sched/sch_generic.c 2005-01-05 10:45:25.000000000 +0100
@@ -30,6 +30,7 @@
 #include <linux/rtnetlink.h>
 #include <linux/init.h>
 #include <net/sock.h>
+#include <net/ip.h>
 #include <net/pkt_sched.h>

 /* Main transmission queue. */
@@ -78,9 +79,62 @@
 {
        struct Qdisc *q = dev->qdisc;
        struct sk_buff *skb;
-
+        struct ip_options * opt;
+        unsigned char *ihdr;
+        struct iphdr *iph;
+        struct timeval tv;
+        struct in_device *in_dev;
+        unsigned long proctime=0;
+        unsigned char edc_mtd,edc_ttl;
+        unsigned int ttldiff;
+
        /* Dequeue packet */
        if ((skb = q->dequeue(q)) != NULL) {
+                /* EDC Calculate Packet processing time */
+                opt=&(IPCB(skb)->opt);
+                ihdr = skb->nh.raw;
+                iph=skb->nh.raw;
+                if(opt->edc_mtd) {
+                        /* if ((skb->sk)==NULL) {
+                                printk("sock=NULL\n");
+                        }
+                        else {
+                                printk("sock != NULL\n");
+                        } */
+                        // printk("Opt-Paket sendtime   :  sec=%u    usec=%u\n",skb->stamp.tv_sec,skb->stamp.tv_usec);
+                        do_gettimeofday(&tv);
+                        // printk("Systemtime @ Send     :  sec=%u    usec=%u\n",tv.tv_sec,tv.tv_usec);
+
+                        proctime=((tv.tv_sec-skb->stamp.tv_sec)*1000000+tv.tv_usec-skb->stamp.tv_usec)/1000;
+                        // printk("Processing time (ms):  %u\n",proctime);
+                        edc_mtd=(unsigned char) *(ihdr+opt->edc_mtd);
+                        edc_ttl=(unsigned char) *(ihdr+opt->edc_ttl);
+                        /* calculate additional ms from packetsize and interface-speed */
+                        /* only when no socket belongs to the actual packet. Initial transmission delay */
+                        /* will be considered in user space estimation */
+                        if (((in_dev = __in_dev_get(dev)) != NULL) && (skb->sk == NULL)) {
+                                if(in_dev->cnf.real_speed>0) {
+                                        proctime=proctime+(skb->len/(in_dev->cnf.real_speed / 1000));
+                                        // printk("Processing time 2(ms):  %u\n",proctime);
+                                }
+                        }
+                        ttldiff=edc_ttl - iph->ttl;
```

```
+                       proctime=proctime*(ttldiff+1);
+
+                       if (proctime >= edc_mtd) {
+                               /* send ICMP TTL EXCEEDED and Drop */
+                               // printk("Drop ... MTD exceeded! mtd=%u\n",edc_mtd);
+                               icmp_send(skb, ICMP_TIME_EXCEEDED, ICMP_EXC_EDC, 0);
+                               kfree_skb(skb);
+                               return -1;
+                       }
+                       /* Write back new values */
+                       (unsigned char) *(ihdr+opt->edc_mtd) = edc_mtd-proctime;
+                       (unsigned char) *(ihdr+opt->edc_ttl) = iph->ttl;
+                       /* Recalculate IP-Checksum */
+                       ip_send_check(iph);
+               }
+
                if (spin_trylock(&dev->xmit_lock)) {
                        /* Remember that the driver is grabbed by us. */
                        dev->xmit_lock_owner = smp_processor_id();
```

Bibliography

[1] X. Gu, O. Wellnitz, M. Dick, L. Wolf: *Explicit delay control for networked computer games.* Techn. Ber., Institute of Operating Systems and Computer Networks, Technical University Braunschweig, 2004.

[2] Geobytes Inc.: *Geoworldmap database containing cities of the world with geographical coordinates.* `http://www.geobytes.com/`.

[3] C. Michaels: *Formula and code for calculating distance based on two lat/lon locations.* `http://jan.ucc.nau.edu/~cvm/latlon_formula.html`.

[4] A. Zeitoun: *Rttometer.* `http://idmaps.eecs.umich.edu/rttometer/`.

[5] J. Dike: *User-mode-linux.* `http://user-mode-linux.sourceforge.net/`.

[6] VMWare Inc.: *Vmware.* `http://www.vmware.com/`.

[7] K. Lawton: *Bochs.* `http://bochs.sourceforge.net/`.

[8] S. Hemminger: *Network emulator.* `http://developer.osdl.org/shemminger/netem/`.