

Marie Camier – Development of a DCCP-Based Adaptive VoIP Client – Aug 2005



**DUBLIN CITY UNIVERSITY
SCHOOL OF ELECTRONIC ENGINEERING**

**Development of an Adaptive
DCCP-based VoIP Client**

Marie Camier
August 2005

**MASTER OF ENGINEERING
IN
TELECOMMUNICATIONS ENGINEERING**

Supervised by Sean Murphy

Acknowledgements

First, I would like to thank Sean Murphy for his guidance and availability, as well as for his commitment to the project.

I would also like to thank Ian McDonald and Arnaldo Carvalho de Melo for their immediate and detailed responses to the questions I asked on the DCCP mailing list, and Laurent Cottureau for the “Linux-Hotline”.

Declaration

I hereby declare that, except where otherwise indicated, this document is entirely my own work and has not been submitted in whole or in part to any other university.

Signed:

Date:

Abstract

In a context where voice over IP and wireless LANs are growing, this document deals with the investigation on the interest and the implementation of an adaptive VoIP client. It was undertaken as my Masters project during the summer months of 2005.

The objective of my project was to develop an adaptive VoIP client based on the DCCP transport protocol.

The interest of implementing adaptive clients is stretched out in this document, as well as possible ways to adapt to networks conditions : through the use of a suitable protocol, DCCP, or by varying the bit rate of the application according to metrics giving information on the state of the network. Specifications on how the adaptivity should be implemented and a frame to this development as well as methods are given, but to date the implementation has not yet been achieved, and should be the object of future work.

Table of Contents

Chapter 1 – Introduction.....	9
1.1. Objective.....	10
1.2. Structure of the document.....	10
Chapter 2 : Issues related to Internet Telephony.....	12
2.1. Transmitting data over wireless networks.....	12
2.1.1. The IEEE 802.11 set of standards	12
2.1.2. WLAN technical issues.....	15
2.1.3. Heterogeneous WANS.....	15
2.2. Challenges of VoIP.....	16
2.2.1. VoIP specific requirements.....	16
2.2.2. Network behaviours affecting VoIP.....	17
2.3. Combining VoIP and WLANs : Interest of an Adaptive Client.....	17
2.3.1. Issues with wireless Internet telephony.....	18
2.3.2. Interest of an Adaptive client in wireless Internet telephony.....	19
Chapter 3 - Datagram Congestion Control Protocol (DCCP).....	20
3.1. Problem Statement.....	20
3.1.1. The need for congestion control.....	20
3.1.2. TCP deficiencies in Internet telephony.....	21
3.1.3. Non-TCP congestion control mechanisms.....	21
3.2. Congestion Control mechanisms.....	22
3.2.1. TFRC : AIMD congestion control.....	22
3.2.2. TFRC : an Equation-Based Congestion Control	24
3.2.3. Smoothness in a steady-state : Comparison of AIMD and TFRC.....	25
3.3. DCCP main features.....	26
3.3.1. CCID2 usage.....	27
3.3.2. CCID3 usage.....	28
Chapter 4 – Adaptive VoIP Client over DCCP.....	30
4.1. Variable Bit-Rate Applications.....	30
4.2. The Speex codec.....	31
4.2.1. Introduction to Speex.....	31
4.2.2. Speex main features.....	32
4.2.3. The Libspeex API.....	34
4.3. An adaptive VoIP client based on DCCP.....	35
4.3.1. Accessing information contained in the DCCP module.....	35
4.3.2. Feedback information to retrieve from the DCCP module.....	36
4.4. Linphone : an extensible VoIP client.....	37

4.4.1. Linphone	37
4.4.2. Implementing DCCP in Linphone	38
Chapter 5 – DCCP Implementations.....	39
5.1. DCCP prototype implementations.....	39
5.1.1. Kernel DCCP for Linux 2.6.....	39
5.1.2. Kernel DCCP for Linux 2.4.....	40
5.1.3. Lulea University's FreeBSD implementation.....	41
5.2. Choice of an implementation.....	41
Chapter 6 – Implementing an adaptive client : Feedback retrieval from DCCP kernel module.....	42
6.1. Linux Kernel architecture.....	42
6.1.1. Introduction to Linux.....	42
6.1.2. Structure of the Linux Operating System.....	42
6.1.3. The Linux kernel.....	43
6.1.4. Splitting the Linux kernel.....	44
6.1.4. Concrete structure of the kernel.....	46
6.2. Linux device drivers.....	47
6.2.1. Classes of devices and modules.....	47
6.2.2. User mode versus kernel mode.....	49
6.3. Kernel modules.....	49
6.3.1. Loading and unloading modules.....	49
6.3.2. The DCCP module.....	50
6.4. Ioctl calls.....	50
6.4.1. ioctl() user function vs ioctl driver method.....	51
6.4.2. Choosing an ioctl command number.....	52
6.5. Implementation of an ioctl call.....	52
Conclusion.....	55
Future work.....	55
References.....	56
Appendix A : Code structure of the DCCP 2.6 module in the 2.6 DCCP Kernel.....	57
A.1. Source tree of the DCCP driver.....	57
A.2. Source tree of the 2.6 DCCP kernel.....	58
Appendix B : Linphone.....	60
B.1. The linphone code structure.....	60
B.2. Compiling, installing and running Linphone.....	62
B.3. Implementing DCCP support in Linphone.....	63
Appendix C : How to compile a Debian Kernel.....	65
C.1. How to compile a Debian kernel.....	65

Marie Camier – Development of a DCCP-Based Adaptive VoIP Client – Aug 2005

C.2. Issues specific to compiling the 2.6 DCCP Debian kernel.....	66
Appendix D : Example of an ioctl driver method.....	68

Table of Figures

Figure 1 : TFRC Throughput	25
Figure 2 : TCP(2/5,1/8) Throughput	26
Figure 3 : TCP Throughput	26
Figure 4 : The four main subsystems of Linux OS	43
Figure 5 : A split view of the kernel	45

Chapter 1 – Introduction

Over the last decade, the Internet has become a major means of sharing data, in both our professional and personal lives.

The recent emergence of Wireless technologies has led to a fast deployment of Wireless Local Area Networks (WLANs) which offer better mobility, scalability and flexibility than wired LANs. At the same time, Voice over IP, a low-cost alternative to traditional telephony, that can offer a wider range of services has started to replace traditional telephony. The growth of both VoIP and of WLANs has led to the emergence of a new application : Wireless Internet Telephony.

Network conditions vary over time, especially in Wireless networks. On the other hand, real-time multimedia applications such as Internet telephony or video conferencing introduced new quality of service (QoS) constraints.

WLAN environments raise specific issues, since wireless channels are lossy and have a narrow bandwidth. Packet loss in WLANs is not necessarily due to congestion and should not be automatically interpreted as such.

A lot of issues related to Voice over IP transmission must also be solved in order to provide good quality of service to VoIP communications : The nature of voice traffic is very specific, and raises issues due to its burstiness, the small size of VoIP packets and the need for timeliness over reliability.

Because of the unpredictable nature of the wireless medium, it is interesting to have VoIP applications which can adapt to the channel conditions. Variable Bit-Rate codecs offer flexibility for these application to adapt.

To date there is no transport-layer protocol that can satisfactorily meet the specific requirements of real-time multimedia streaming applications such as VoIP. The IETF is currently working on a new standard, called Datagram Congestion Control Protocol (DCCP) which implements a new congestion control mechanism more suitable for such applications.

1.1. Objective

The objective of this project is to combine the benefits of DCCP congestion control mechanism for VoIP quality with these of Variable Bit-Rate codecs, in an adaptive DCCP-based VoIP client.

The aim of this report is to explain how the congestion control mechanisms implemented in DCCP adapt the sending rate to the network congestion, and how Variable Bit-Rate codecs allows applications to adapt the encoding and decoding rate to the network conditions.

The implementation of such a client requires to choose a suitable codec and to find a way to retrieve information on the network conditions in order to adapt to them.

1.2. Structure of the document

This document is composed of five other chapters. Chapter 2 gives the issues that wireless Internet telephony raises. It first presents the QoS issues of wireless LANs and the VoIP specific requirements before presenting the issues related to the combination of both these technologies into Wireless VoIP.

Chapter 3 then gives an overview of the DCCP transport protocol on which my adaptive client will be based, and explains how this protocol's congestion control mechanisms make it more suitable for Internet telephony.

Marie Camier – Development of a DCCP-Based Adaptive VoIP Client – Aug 2005

Chapter 4 deals with the influence of codecs on the quality of a voice application, and how Variable Bit Rate codecs provide a way to adapt to changing network conditions. It gives an overview of the Speex codec and presents the feedback information to retrieve from DCCP according to which the client will adapt.

Chapter 5 is a quick presentation of three different DCCP implementations that I have experimented. It explains why I have chosen to work with one of them in particular.

Finally, chapter 6 deals with the first steps of the client implementation, and more specifically with the feedback retrieval from DCCP module. It contains an overview of the Linux kernel structure that shows that retrieving information from the kernel is a complicated issue, and gives a mechanism to access this information.

Chapter 2 : Issues related to Internet Telephony

This chapter presents the new QoS challenges that wireless networks and VoIP have raised, which leads to the idea of using adaptive clients to overcome the difficulty of combining both these technologies in a new application : wireless Internet telephony.

2.1. Transmitting data over wireless networks

This section presents the challenges of transmitting data over wireless networks, starting with a brief overview of the 802.11 set of standards.

2.1.1. The IEEE 802.11 set of standards

As described in [802.11] this standard is part of the 802.x (IEEE LAN / MAN) family of standards. This family of standards deals with the Physical and the Data Link layers of the OSI model.

802.11 standard was accepted by the IEEE in 1997, revised in 1999. It specifies the wireless Medium Access Control (MAC) and Physical Layer (PHY), to provide wireless connectivity for fixed, portable and moving devices within a LAN.

There are several specifications in the 802.11 set of standards :

- **802.11** is the first WLAN standard created by the IEEE. It provides up to 2Mbps in the 2.4GHz band. 802.11 wireless products are no longer manufactured and this specification has been extended into 802.11b. The modulation scheme used is either frequency hopping spread spectrum (FHSS) or direct sequence spread spectrum (DSSS). It supports Wired Equivalent Privacy (WEP) and Wi-Fi Protected Access (WPA) security protocols.

- **802.11b** [802.11b] (also referred to as *802.11 High Rate* or *Wi-Fi*) is an extension to 802.11 that provides up to 11Mbps in the 2.4GHz band. 802.11b uses only DSSS encoding scheme, with Complementary Code Keying. 802.11b allows wireless functionality comparable to Ethernet. It is not interoperable with the 802.11a standard. It supports WEP and WPA security protocols.
- **802.11a** [802.11a] is an extension to 802.11. It was created at the same time as 802.11b. It provides up to 54 Mbps in the 5 GHz band. It uses an orthogonal frequency division multiplexing (OFDM) encoding scheme. It is not interoperable with 802.11b as it operates at a different frequency. It is better than 802.11b for multimedia applications in densely populated environments. WEP and WPA security protocols are supported.
- **802.11g** [802.11g] provides up to 54 Mbps in the 2.4 GHz band. Its modulation scheme is DSSS with CCK below 20Mbps and OFDM above above 20Mbps. It supports WEP and WPA security protocols. This specification may replace 802.11b for it has an improved security. It is compatible with the 802.11b standard.
- **802.11e** is a forthcoming standard for QoS over Wireless networks, on which the IEEE 802.11 TaskGroup E is working. It is more adapted for VoIP on wireless networks. It enhances the 802.11 standard so as to expand support for applications with QoS requirements, to meet multimedia streaming applications.

b) Mode of operation

Two different operation modes have been specified in the 802.11 standard, the Infrastructure mode and the Ad-Hoc mode. Most WLANs today utilize "infrastructure" mode.

In the infrastructure mode, the wireless network uses at least one Access Point connected to the wired network architecture. In the Ad-Hoc mode, end-stations spontaneously form a wireless network, without the need of an access point, or any connection to a wired network.

c) Channel access mechanisms

The 802.11 standard has two channel accessing mechanisms : the distributed coordination function (DCF) and the point coordination function (PCF).

• ***The Distributed Coordination Function***

DCP is based on the carrier sense multiple access with collision avoidance CSMA/CA, though it is a little more complex and also used duration fields sent by end-stations to make the decision of when a station is going to transmit data. This duration field enables stations to reserve the medium for frames to be sent subsequently.

• ***The Point Coordination Function***

This access method is optional in the 802.11 standard. It enables the transmission of time-sensitive information. It allows stations to transmit data synchronously by letting each station transmit data at a given time : Within a time period called the contention free period, the point coordinator steps through all the stations in PCF

mode and let them transmit data during a specific period of time.

PCF is thus a contention-free protocol, which allows all stations in PCF mode to transmit data with regular time delays between their frames.

PCF is more appropriate than DCF for VoIP applications, since it enables the transmission of time-sensitive information, but much more complex.

2.1.2. WLAN technical issues

Wireless technologies raise specific issues due to the nature of the medium and to the protocols used, that affect the quality of service of WLAN.

Wireless channels are unpredictable, and bursty channel errors can occur, due to the different paths that packets may take to reach the destination. These channel errors lead to corruption or loss of packets.

The bandwidth in WLANs is also narrower than in most wired networks. Congestion is thus more likely to happen, especially at the Access Points.

RF interference is also a big problem. A source of RF interference can block the WLAN as long as the interference signal is present if DCF is used, since the stations sensing energy on the medium wait until they sense that the medium is free.

2.1.3. Heterogeneous WANs

In most cases, communication are established between two devices from distinct LANs. Data is routed through a wired Wide Area Network. In the case of a device belonging to a Wireless LAN, data will be passed through both a wireless link and a wired link. This is referred as a heterogeneous WAN.

The bottleneck of such a heterogeneous networks is generally the wireless link, as the wired linked can be considered as more reliable and more lightly loaded.

In such systems we will thus only study the packet loss and congestion problems on the wireless part, considering that it is where such problems.

2.2. Challenges of VoIP

Voice over IP (VoIP) is the routing of voice conversations over any IP network. Data is transmitted over a packet-switched network, whereas traditional telephony uses dedicated circuit-switched networks. Signalling protocols, such as SIP or H.323, establish, control and terminate phone calls.

VoIP is being widely adopted and is in rapid expansion, due to its low cost and the extra-functionalities VoIP applications can offer.

Quality of VoIP applications is variable, and there is no guaranteed QoS.

2.2.1. VoIP specific requirements

Voice traffic has specific requirements due to its nature.

Telephony traffic is bursty : It is composed by alternating periods of talk spurts and silence. Voice quality is also very sensitive to delay and jitter, and packets have to be display in order for speech to be understood.

VoIP applications require timeliness over reliability, as a packet arriving too late is of no use and may as well never arrive. It is more important that most packets arrive on time, than that all packets arrive at some time. Many voice codecs are robust to packet loss, to a certain extent.

Delay depends to the transmission duration of a packet : smaller packets are transcoded and transmitted faster. This is why VoIP packets are usually small. A compromise must be found in order to reduce transmission and transcoding delays while still have a satisfactory payload.

2.2.2. Network behaviours affecting VoIP

IP networks do not offer any guarantee on the order of the packet delivery or on the quality of service. Voice over IP applications can thus face severe problems due to latency, and must restructure the received streams that may have packets missing, or coming in any order. For a low-latency, high quality voice, the network provider should ensure that there is a sufficient end-to-end bandwidth.

VoIP is susceptible to the following network behaviours : delay, jitter and packet loss.

- **Delay** is the time taken for a packet to go from a point to another point in the network. It can be measured in one-way or in round-trip-time delay. VoIP tolerates delay up to 150ms before the quality of the call becomes unacceptable.
- **Jitter** is the variation of the delay over time from one point to another point in the network. The call can be very degraded if this variation – or jitter – is too wide. The amount of jitter that VoIP applications can tolerate depends on the size of the jitter buffer on the network.
- **Packet loss** can also affect VoIP applications.

2.3. Combining VoIP and WLANs : Interest of an Adaptive Client

Internet telephony has become increasingly used, but adapting it to lossy wireless LANs raises has become a new challenge.

In VoIP, delay is critical. Moving VoIP to wireless technologies raises new

performance issues, as wireless networks have a different behaviour than wired networks. Combining VoIP and WLANs is even more challenging. The delay and jitter are accentuated by the lossy nature of the wireless links. Access point congestion, as well as RF interference can dramatically affect link quality.

The Ethernet was not designed for real-time streaming but however, wired LANs with ample bandwidth can satisfactorily meet VoIP quality of service requirements, and Voice can more easily cohabit with data on wired LANs.

2.3.1. Issues with wireless Internet telephony

Internet telephony on WLANs introduce several new concerns :

- **Congestion**

Congestion in wireless LANs can cause problems. A 802.11 group is working on the 802.11e standard which QoS requirements make delay acceptable for multimedia streams.

- **Mobility**

Most wireless telephone users tend to walk or travel while speaking on the phone. A support for mobility could be needed, which is not an issue in wired LANs. Wireless coverage may need to be extended to places where it was not needed considering a static user (like stairs, corridors, etc.).

- **Security**

Eavesdropping is easier to achieve on wireless networks, for the transmission media is the air. Security must be implemented, so that conversations can stay

confidential. Security should be added without affecting VoIP quality of service by introducing longer delays.

- **Hardware**

Hardware delays should be reduced to the minimum.

This chapter concentrates on solving the congestion problems in IP telephony on WLANs. A client that dynamically adapts to the networks conditions is a interesting way of avoiding congestion.

2.3.2. Interest of an Adaptive client in wireless Internet telephony

An adaptive client is a client that tries to adapt to the current network conditions so as to use the available bandwidth in an intelligent way. In VoIP applications delay and jitter are particularly critical, and in wireless networks network conditions can vary very fast. This is why it is interesting to think how an adaptive VoIP client can operate in this wireless context.

There are several ways of adapting to the network condition. DCCP offers control mechanisms that allow to vary the transmission rate in packets/second. Variable Bit-Rate codecs allow to vary the transcoding bit rate by changing the size of the encoded packets.

Chapter 3 - Datagram Congestion Control Protocol (DCCP)

This chapter describes the new requirements of internetwork communications that led the IETF (Internet Engineering Task Force) to acknowledge the need for a new protocol and to define DCCP [DCCP]. It gives an overview of the benefits of TCP Friendly protocols in Internet telephony, as well as a description of DCCP features.

3.1. Problem Statement

Recently, there has been an emergence of new types of applications, like Internet telephony or all types of multimedia streaming, that share a preference for timeliness over reliability. Since TCP introduces arbitrary delay due to its reliability, these application sometimes use UDP instead of TCP, like Linphone, a free VoIP phone. But UDP does not provide built-in congestion control, and the growth of this non congestion-controlled traffic may cause problems to the stability of the Internet.

3.1.1. The need for congestion control

For the overall stability of the Internet, it is necessary that most traffic implements congestion control. A great part of the traffic uses congestion-controlled TCP, UDP making up most of the remainder. UDP applications can implement their own congestion control mechanisms, but most of them tend not to, since it is hard to implement congestion control for unreliable flows. Up to recently, since UDP traffic was relatively small, the effect of non congestion controlled traffic was limited. But with the emergence of new applications using UDP rather than TCP for timely

reasons, there is some concern about how the growth of non congestion controlled traffic may threaten the stability of the Internet.

3.1.2. TCP deficiencies in Internet telephony

Because of the delay that its reliability introduces, TCP is not optimal for Internet telephony and in general for applications that prefer timeliness over reliability.

In the case of telephony, if a datagram is lost, the application may not wish to send it again, since it may no longer be useful : It is more important that datagrams averagely arrive in time for their play-out than that they all arrive some time. That is why TCP reliability, which introduces some delay, is not suitable. As we have seen in the previous part, UDP can be used, but no congestion control is implemented within this protocol.

Another major problem of TCP is that it abruptly halves its congestion window in response to a single packet loss. This unnecessarily severe change in the sending rate can noticeably reduce the perceived quality of the voice call. A smoother adaptation to the network capabilities is required.

3.1.3. Non-TCP congestion control mechanisms

A lot of research has been carried out to overcome the problem of congestion control for such applications. Though applications using UDP can implement their own congestion control, it is not an easy task.

A number of TCP-friendly transport-layer mechanisms have also been proposed, that provide smoother congestion control : The next section describes two of them, which are implemented in DCCP.

3.2. Congestion Control mechanisms

This section introduces two congestion control mechanisms implemented in the DCCP standard. Both are TCP-Friendly, which means that they were designed to compete fairly with each other and with TCP-Based applications in FIFO (First In Foist Out) queues.

The two TCP-Friendly Congestion Control mechanisms used by DCCP are the TCP-like AIMD (Additive-Increase, Multiplicative Decrease), and TFRC equation-based congestion. TFRC implements smooth changes in the sending rate, rather than opportunistically use of increases in available bandwidth [TFRC].

3.2.1. AIMD congestion control

AIMD uses additive increase and multiplicative decrease of the congestion window. This congestion control mechanism is described in [AIMD].

AIMD(a, b) congestion control uses an increase parameter a and a decrease parameter b . After a loss event, the congestion window is decreased from W to $(1-b)W$ packets. Otherwise, the congestion window is increased from W to $W + a$ packets each round-trip time.

AIMD is the mechanism used by TCP, with $a = 1$, and $b = \frac{1}{2}$: AIMD(1, $\frac{1}{2}$). TCP increases its window size by one packet when no loss occurs, and reduces it to half its size if a packet is lost. For an AIMD congestion control mechanism to reduce its sending rate more smoothly than TCP, the most obvious choice is thus to use a decrease parameter b less than $\frac{1}{2}$.

3.2.1.1. The deterministic AIMD response function

The deterministic AIMD model used in [AIMD] assumes that a packet is lost each

time the congestion window reaches W packets (i.e. each time the congestion window is full).

The AIMD(a,b) response function T is the AIMD flow's steady state sending rate T in packets per second in the deterministic model. It is a function of the increase and the decrease parameters a and b , the Round-Trip-Time R , and the packet-drop rate p .

The AIMD response function is :

$$T = \frac{pa(-2 + b) + \sqrt{p(-2 + b)a(pba - 8b - 2 pa)}}{4 pbR}$$

The approximate response function is :

$$T' = \frac{\sqrt{(2 - b)a}}{R\sqrt{2 bp}}$$

3.2.1.2. Choice of the increase and the decrease parameters

In this part I will give a and b values for which AIMD(a,b) is compatible with AIMD(1, 1/2), i.e. TCP-compatible, and would provide better smoothness to the bit-rate changes (b less than 1/2)

When we apply the approximate response function to AIMD(1, 1/2), we get the approximate TCP response function :

$$T' = \frac{\sqrt{1.5}}{R\sqrt{p}}$$

In order for AIMD(a,b) to be TCP compatible, i.e. to have the same long-term sending rate as AIMD(1, 1/2), we need to have :

$$\frac{\sqrt{(2-b)a}}{R\sqrt{2 bp}} = \frac{\sqrt{1.5}}{R\sqrt{p}}$$

From this equation, two solutions were found – AIMD(3/7,1/4) and AIMD(1/5,1/8) – which should compete reasonably fairly with AIMD(1, 1/2).

3.2.2. TFRC : an Equation-Based Congestion Control

TFRC is an equation-based congestion control mechanism for uni-cast traffic where the sender adjusts its sending rate as a function of the loss event rate. This mechanism has been designed to maintain a smoothly-changing sending rate, while still being responsive to network congestion over long time periods [TFRC].

3.2.2.1. The control equation : TCP response function

In equation-based congestion control, the basic decision is what control equation is going to be used. TFRC has been designed to compete fairly with TCP in First In First Out queues. For this to happen, the two types of traffic must have similar response functions. The correct choice for the control equation is thus the TCP response function.

Here is one formulation of this function from [TCP]:

$$T = \frac{s}{R \sqrt{\frac{2p}{3}} + t \left(3 \sqrt{\frac{3p}{8}} \right) p (1 + 32 p^2)}$$

with R the round-trip time, s the packet size, p the steady-state loss event rate, t the TCP retransmit time-out.

3.2.2.2. The TFRC protocol

The goals of TFRC are, in contrast with TCP :

- not to aggressively seek out all available bandwidth (i.e. increase the sending rate slowly)
- not to halve the sending rate in response to a single loss event

- the receiver should report feedback to the sender
- if the sender has not received any feedback, it should reduce its sending rate, then stop sending

In TFRC, the receiver estimates the loss event rate, the loss event being *one or more* packets lost within a round-trip time. A loss interval is defined as the number of packets transmitted between two loss events. The loss event rate is measured over the most recent loss intervals, using a method called the Average Loss Interval method. It computes a weighted average of the loss rate over the last n loss intervals, with equal weights on each of the most recent $n/2$. This method is the method that gives the smoothest rate changes.

3.2.3. Smoothness in a steady-state : Comparison of AIMD and TFRC

Equation-based congestion control (TFRC) provides a better smoothness in a steady-state than AIMD congestion control (TCP). The figures 1 to 3 from [AIMD] show how TCP(2/5,1/8) is less bursty than TCP, and how TFRC flows are considerably smoother than TCP flows.

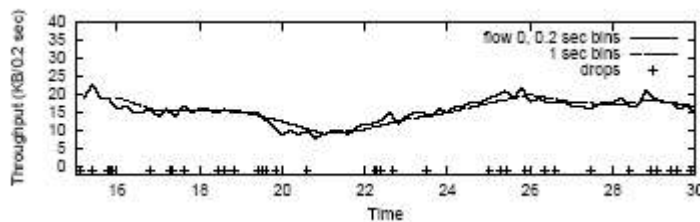


Figure 1 : TFRC Throughput

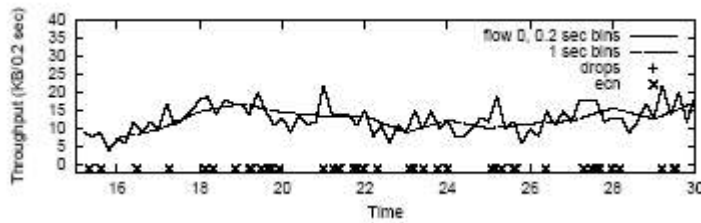


Figure 2 : TCP(2/5,1/8) Throughput

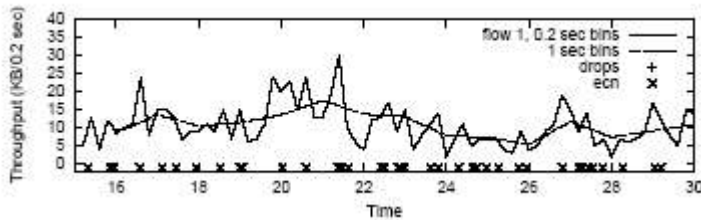


Figure 3 : TCP Throughput

Both TFRC and a TCP-like congestion control mechanism are implemented in DCCP.

3.3. DCCP main features

The Datagram Congestion Control Protocol implements a congestion-controlled, unreliable flow of datagrams. It is suitable for applications such as streaming media, that can “benefit from control over the tradeoffs between delay and reliable in-order delivery” [DCCP]. DCCP provides built-in congestion protocols and avoids the arbitrary delays associated with TCP.

This extract of [DCCP] gives the list of features that DCCP provides :

“

- ◆ Unreliable flows of datagrams, with acknowledgements.
- ◆ Reliable handshakes for connection setup and teardown.

- ◆ Reliable negotiation of options, including negotiation of a suitable congestion control mechanism.
- ◆ Mechanisms allowing servers to avoid holding state for unacknowledged connection attempts and already-finished connections.
- ◆ Congestion control incorporating Explicit Congestion Notification (ECN) and the ECN Nonce.
- ◆ Acknowledgement mechanisms communicating packet loss and ECN information. Acks are transmitted as reliably as the relevant congestion control mechanism requires, possibly completely reliably.
- ◆ Optional mechanisms that tell the sending application, with high reliability, which data packets reached the receiver, and whether those packets were ECN marked, corrupted, or dropped in the receive buffer.
- ◆ Path Maximum Transmission Unit (PMTU) discovery [RFC 1191].
- ◆ A choice of modular congestion control mechanisms. Two mechanisms are currently specified, TCP-like Congestion Control and TFRC (TCP-Friendly Rate Control) Congestion Control, but DCCP is easily extensible to further forms of unicast congestion control.”

DCCP was designed for applications that require the flow-based semantics of TCP, but have a preference for delivery of timely data over in-order delivery or reliability.

3.3.1. CCID2 usage

DCCP uses Congestion Control Identifiers, or CCIDs, to specify the congestion control mechanism in use on a half-connection. Two CCIDs have been defined : CCID2 and CCID3.

CCID2 is the TCP-like congestion control mechanism implemented in DCCP, described in [CCID2].

CCID2 should be used for applications that would like to take advantage of the available bandwidth in an environment with rapidly changing conditions, and that can adapt to abrupt changes of the congestion window.

CCID2 is suitable for flows that would like to receive as much bandwidth as possible over the long term, and that can tolerate the large sending rate variations, as the halving of the congestion window following a congestion event.

Here are some differences between CCID 2 and straight TCP congestion control :

- CCID2 applies congestion control to acknowledgements, which TCP does not do.
- Since DCCP is a datagram protocol, the units for the congestion window are in packets, whereas it is in bytes for TCP.
- DCCP never retransmit packets so the TCP congestion control mechanisms that distinguish new packets from retransmitted packets have been redesigned for the DCCP context.

3.3.2. CCID3 usage

The Congestion Control Identifier 3 is the TCP-Friendly Rate Control implemented in DCCP, described in [CCID3].

As described in 3.2, TFRC is a congestion control mechanism that provides a TCP-Friendly sending rate while minimizing the abrupt rate changes of TCP. The rate at which a sender is allowed to send data is determined according to the loss event rate reported by the receiver to the sender.

CCID3 is appropriate for applications that suffer from abrupt changes in the sending

Marie Camier – Development of a DCCP-Based Adaptive VoIP Client – Aug 2005

rate, like streaming media applications with small receiver buffering, like VoIP applications. CCID3 allows such applications to have a smoother throughput but responds slower to changes in available bandwidth. This is why applications that just need to transfer as much data as possible in as short time as possible should use CCID2 rather than CCID3.

For applications that change their sending rate by varying the packet size, a new CCID will be required.

Chapter 4 – Adaptive VoIP Client over DCCP

This chapter describes how an adaptive client can overcome the congestion problems caused specific to combining VoIP and wireless technologies by using Variable Bit-Rate codecs, and gives details on the implementation of such an adaptive client based on the DCCP protocol.

4.1. Variable Bit-Rate Applications

Voice traffic is composed of periods of talk and periods of silence. WLAN bandwidth can be used more efficiently if we take this feature into account.

DCF mode is not suitable for QoS implementation.

In PCF mode, stations are given a specific period of time in which they can transmit. If Constant Bit Rate traffic is sent on this PCF mode, we have fixed length packets sent at predetermined intervals of time. This is not really adapted to the bursty nature of voice traffic.

Using Variable Bit Rate traffic significantly improves end-to-end delay, as well as the capacity of the system, as simulations carried out [ADAP] and in [VBR] have showed.

Variable bit rate applications can vary the bit rate according to parameters, such as the amount of data to send (high energy data requires a higher bit-rate than low energy data), and the network conditions.

For example, in voice traffic, talk periods contain more information than silent periods of time. Silence can thus be encoded with smaller packets, and should be, in order to use bandwidth more intelligently. The packet size then varies according to the amount of information to be encoded.

The other parameter that can be considered when choosing the bit rate is the state of the network just before the datagram is sent : If the channel capacity is high enough (channel lightly loaded) to encode voice with a high quality (i.e. a high bit rate), the application should do so. If the channel load does not allow an application to send high bit rates without significantly increasing the end-to-end delay, then a compromise should be found between the quality of transcoding and keeping the delay acceptable. A lower bit rate could then be a better option.

To sum it up, VBR applications should allow to find the best compromise between the quality of the voice data sent over the network and the end-to-end delay, based on the nature of the data to encode and on the network conditions.

4.2. The Speex codec

Speex is the codec I have chosen for the implementation of my adaptive client, since it can support both low and high bit-rates, and has features targeted for voice such as a voice detector. It is also open-source and free from software patents.

4.2.1. Introduction to Speex

Speex is an open source and free from software patent speech codec, targeted for

Voice over IP. It is currently used for Voice over IP clients such as Linphone and Asterisk. It is a multi-rate codec, that can detect voice activity. It allows both low bit-rate and high voice quality, though not in the same time.

Speex is based on the CELP (Code-Excitation Linear Predictive) encoding technique. CELP can scale well to both low bit-rates (as in the DoD CELP codec, at 4.8kbps) and to high bit-rates (G.728 speech coder standard, with 16 kbps).

4.2.2. Speex main features

Here is a list of Speex main features, as described in [SPEEX]:

- **Sampling rate**

Speex is mainly designed for 3 different sampling rates : 8kHz (narrowband), 16kHz (wideband) and 32kHz(ultra-wideband)

- **Quality**

Speex encoding can be controlled by a quality parameter ranging from 0 to 10. This parameter is an integer in constant bit-rate (CBR) encoding, a float in variable bit-rate (VBR) encoding.

- **Variable complexity**

It is possible to vary the complexity allowed for the encoder. This allows to find the best compromise between CPU requirements and the noise level in each case.

- **Variable Bit-rate (VBR)**

VBR allows a codec to change its bit-rate dynamically to adapt to the amount of information contained in the audio being encoded : in Speex, vowels and high-energy transients require a higher bit-rate to achieved good quality, while low-energy sounds can be coded with less bits. VBR can thus achieve lower bit-rate for the same quality or a better quality for the same bit-rate. There are two drawbacks to Speech VBR : First, if only quality is specified, there is no guaranty about the average bit-rate. Second, in the case of Voice over IP, what counts is the maximum bit-rate, which must be low enough for the channel.

- **Average Bit-rate**

Average bit-rate solves the problem of VBR by dynamically adjusting VBR quality in order to meet a specified bit-rate. The resulting global quality will be slightly lower.

- **Voice Activity Detection (VAD)**

This feature is very useful in Voice over IP : VAD detects whether the audio being encoded is speech or silence/background noise. VAD is implicitly activated when encoding using VBR. Speex uses just enough bits to reproduce a “comfort” background noise when a non-speech period is detected. (This comfort noise is necessary for the user to know that the communication has not been cut).

- **Discontinuous Transmission (DTX)**

Discontinuous transmission allows to stop transmitting completely when the background noise is stationary.

- **Perceptual Enhancement**

Perceptual enhancement is used in the decoder. When it is turned on, it tries to reduce the perception of the noise produced by the coding / decoding process. It makes the sound objectively further from the original, but it sounds better, which is a subjective improvement for the user.

- **Algorithmic Delay**

Speech codecs introduce delay in the transmission. For Speex, it is equal to the frame size, plus delay required to process each frame. In narrowband operation, the delay is 30 ms, in wideband 34 ms, without taking into account the CPU time necessary to encode or decode the frames.

4.2.3. The Libspeex API

This section briefly describes the main operations that can be implemented in a client using Speex API.

Programming with Speex is very well documented in [SPEEX], which gives the code necessary for each of the following operations :

- **Encoding**
- **Decoding**
- **Codec Options** : the Speex encoder and decoder support many options and request that can be accessed in the encoding and decoding functions.
- **Mode queries** : Speex modes have a query system similar to the encoding and decoding functions. Speex API has a mode query function that allows to get information about a particular mode (the frame size or the bit-rate for the mode).
- **Packing and in-band signalling** : Packing is used when it is necessary to pack more than one frame by packet. It is also possible to send in-band messages. Most

of these messages are requests that are sent to the encoder or the decoder on the other end. They are ignored by default.

4.3. An adaptive VoIP client based on DCCP

The idea behind the conception on an adaptive VoIP client based on DCCP is to combine the benefits of congestion control mechanisms implemented in DCCP and the benefits of Variable bit-rate codecs, so as to improve the QoS of Internet telephony on DCCP.

The adaptivity will then combine the varying rate in packets/second as implemented in congestion control mechanisms as well as the variable size of the packets transcoded by the Variable Bit Rate (VBR) codec.

Unfortunately the benefits of TFRC mechanism can not be associated with the benefits of VBR encoding, as [CCID3] states that CCID3 is not suitable for applications varying their packet size. A new CCID will be needed for such applications, that should use CCID2 in the meantime.

The benefits of DCCP alone (less delay than TCP thanks to the unreliability) can though be combined with VBR encoding, but for now the choice of congestion control mechanism should be CCID2.

Transmitting voice traffic using TFRC is a hot topic in research at the moment and has just led to the publication of a new Internet-draft.

4.3.1. Accessing information contained in the DCCP module

The only access point to DCCP module from a client is the socket. There is no API, and no feedback retrieval has been implemented in any of the available DCCP prototype implementations. Retrieving feedback from DCCP thus implies either

adding a significant number of functionalities to DCCP code in order to implement an API that would allow a user-application to access information, or more simply trying to retrieve information from the socket, which is the solution chosen for my adaptive client.

There are issues with accessing information that is in the kernel memory space from a user application, as explained in chapter 6.

4.3.2. Feedback information to retrieve from the DCCP module

In order to adapt to the network conditions, a VBR application requires a certain number of metrics that describe the state of the network.

The round-trip time (RTT) is a very important parameter. It can be calculated at the application level, but it would be a good idea to implement the retrieval of this metric from DCCP in the DCCP module.

The sending rate is another important metric, as well as the congestion window size, in case the congestion control mechanism uses a congestion window (i.e. for CCID2).

The problem here is that we only have access to the socket information.

Here is the information that we can get from the socket [DCCP]:

Socket variables:

- S.SWL - sequence number window low
- S.SWH - sequence number window high
- S.AWL - acknowledgement number window low
- S.AWH - acknowledgement number window high
- S.ISS - initial sequence number sent
- S.ISR - initial sequence number received
- S.OSR - first OPEN sequence number received
- S.GSS - greatest sequence number sent
- S.GSR - greatest valid sequence number received
- S.GAR - greatest valid acknowledgement number received on a non-Sync; initialized to S.ISS

We can get the window-size W through the *S.SWL* and *S.SWH* fields :

$$W = S.SWH - S.SWL$$

It can also be interesting to have the acknowledgement congestion window, that can be obtained with the *S.AWH* and *S.AWL* variables.

Unfortunately, there is no information on the sending rate in the socket structure. The sending rate can be deduced from the window size and the RTT, that we can obtain at the application level as a start (it would be better to implement an API in DCCP to get the RTT from DCCP module, in future work).

4.4. Linphone : an extensible VoIP client

Linphone is a free Internet Telephony software developed under the GNU Public License (GPL) by the French developer Simon Morlat. The current version of Linphone is version 1.0.1.

4.4.1. Linphone

a) General Presentation

Linphone works with the Gnome Desktop under Linux, though it can also be compiled without Gnome in console mode, by using the program called *linphonec*.

Linphone transport protocol is UDP.

It implements SIP protocol, which makes it compatible with all phones using SIP.

b) Codecs implemented in Linphone

Linphone implements the Speex codec chosen to be the codec to use in my client.

Linphone actually allows the user to choose between these codecs :

- GSM : Codec widely used for mobile phones

- PCMU : This is the Mu-law codec ; it can only be used with high-speed connexions.
- PCMA : A-law codec. Only usable with high-speed networks.
- Speex
- 1015 : A low quality but very low-bit rate codec developed by the US Department of Defense.

4.4.2. Implementing DCCP in Linphone

Linphone code structure (see Appendix B) is simple. It is therefore easy to target specific Linphone modules where to make changes in order to implement DCCP instead of UDP as a transport protocol.

This is done by changing the socket type from SOCK_DGRAM (UDP socket) to SOCK_DCCP.

Chapter 5 – DCCP Implementations

Many developers are currently working on implementations of DCCP. Though a number of prototype implementations have been developed in the last few years, no production-quality implementation exists. This chapter presents some of the existing DCCP implementations that I have experimented, and which implementation I have chosen to work on, and why.

The prototype implementations introduced in this chapter can be downloaded from the DCCP web-page [WebDCCP], which also contains the current and old IETF Internet drafts related to DCCP.

It would also be useful for anyone carrying on this project to join the official IETF DCCP mailing-list dccp@ietf.org, and consult the archives of the mailing-list by subscribing on the DCCP Web-page.

5.1. DCCP prototype implementations

Here are the three most recent implementations officially released and available from the DCCP official web-page.

5.1.1. Kernel DCCP for Linux 2.6

This implementation of DCCP is downloadable as a kernel containing a DCCP Network Driver module. It is the most stable and most up-to-date implementation of DCCP, and is close to production-quality.

It has been mainly developed by Arnaldo Carvalho de Melo and Ian McDonald, who

Marie Camier – Development of a DCCP-Based Adaptive VoIP Client – Aug 2005

are still working on the project. They developed it from scratch but building on IP and TCP work in the kernel, and taking the Wakaito CCID3 implementation.

In this prototype DCCP is implemented as a Network driver module. It is located in /net/dccp in the Linux kernel 2.6.

The structure of the code can be found in Appendix A. It is very similar to the structure of the TCP module.

This implementation of DCCP seems not to be mature for laptops yet.

CCID 2 is missing, only CCID 3 has been implemented.

5.1.2. Kernel DCCP for Linux 2.4

This is a prototype implementation that has been developed and tested on User Mode Linux (UML), kernel 2.4.

This code has been developed by the WAND research group of the University of Waikato, based on code was written by Patrick McManus.

Ian McDonald has packaged the code for release and is now working on the implementation upgrade for Linux kernel 2.6.

User Mode Linux allows to run Linux inside of Linux : it allows to run an entire Linux as a program. The virtual machine may have more hardware and software resources than the physical computer. Whatever is done within the virtual machine cannot damage the physical computer. If the virtual machine crashes, the host kernel is still fine. It is ideal for safe kernel development.

Marie Camier – Development of a DCCP-Based Adaptive VoIP Client – Aug 2005

This implementation has a number of bugs and only runs under Linux 2.4. It has only been tested on User Mode Linux. This implies that it requires UML to be successfully installed.

5.1.3. Lulea University's FreeBSD implementation

This implementation of DCCP is downloadable in <http://www.freebsd.dccp.org/>.

It is a patch to apply to FreeBSD 5.0.

This implementation has been developed by Magnus Erixzon, Nils-Erik Mattsson and Joacim Häggmark of Lulea University in Sweden under the supervision of Sara Landström.

This implementation is now out of date. Better implementations (see above) based on this one have been released.

5.2. Choice of an implementation

When I started to work on the project, I chose Lulea's implementation that was the most recent release downloadable from DCCP Web-page.

Then I started to work on the Linux 2.4 prototype implementation, which was meant to be more stable, but still has severe stability issues on laptops.

The recent release of the Linux 2.6 implementation (in July 2005) made me switch to this implementation, which is considered by the DCCP community as the most mature and the closest to production-quality implementation to date.

Information on how to compile the DCCP-2.6 kernel can be found in Appendix C. There are severe stability issues when installing this implementation on laptops.

Chapter 6 – Implementing an adaptive client : Feedback retrieval from DCCP kernel module

This chapter gives an overview of the Linux Kernel architecture, Linux device drivers and Linux kernel modularity. It also describes how ioctl calls can be used to retrieve information from the kernel, and contains methods to implement an ioctl call for my adaptive client.

6.1. Linux Kernel architecture

This section introduces the Linux kernel architecture. The understanding of this architecture is essential for whoever needs to program for the kernel.

6.1.1. Introduction to Linux

The first Linux Kernel was written by Linus Torvalds in 1991. Linux has always been distributed as free software. This is one of the reasons why it gained popularity. The source code is available and users can freely change the kernel to suit their needs.

Since 1991, Linux has gone through several revisions. It is being developed jointly by a group of volunteers, with Linus Torvalds as the main kernel developer.

6.1.2. Structure of the Linux Operating System

The Linux operating system is composed of four major subsystems [KER] :

- User Applications : This subsystem contains user applications, such as a web-browser, or a text-editor, etc.

- O/S Services : This subsystem contains services that are typically considered part of the operating system, such as a windowing system, a command shell, etc. It also contains the programming interface to the kernel (compiler and libraries).
- Linux Kernel : controls and mediates the access to hardware.
- Hardware Controllers : This subsystem contains all the physical devices (CPU, memory hardware, network hardware, hard disks, etc.).

User Applications
O/S Services
Linux Kernel
Hardware Controllers

Figure 4 : The four main subsystems of Linux OS [KER]

6.1.3. The Linux kernel

As other operating systems, the Linux kernel controls and mediates the access to hardware. It starts programs, assigns memory and other resources for all the running programs and processes and makes sure that they <https://mail.dcu.ie/mail/mainall> get the appropriate share of the processor's cycles (multi-task support). It also provides an interface for programs to talk with hardware, responds to user's requests through system calls, receives packets from and send packets to the network, etc..

6.1.4. Splitting the Linux kernel

[DEVDR1] proposes a conceptual view of the Linux kernel where it can be split the into five main subsystems :

- Process management

The kernel creates and destroys processes. It also handles their connection to the outside world (input and output). Communication among processes is also handled by the kernel. The scheduler, which controls how processes share the CPU, is part of process management.

- Memory Management

The computer's memory is a major resource, and the policy used to deal with it is critical for system performance. The kernel builds up a virtual addressing space for every process on top of the limited available resources. The different parts of the kernel interact with the memory-management subsystem through a set of function calls, like the *malloc/free* pair.

- Filesystems

Unix is based on the filesystem concept : almost everything can be treated as a file. The kernel builds a structured filesystem on top of unstructured hardware, and the resulting file abstraction is used throughout the whole system. Linux supports multiple filesystem formats.

- Device Control

Almost every system operation maps to a physical device. Apart from the processor, the memory, and a very few other entities, every device control operations is

performed by code that is specific to the device being addressed : the corresponding device driver. The kernel must have a device driver for every peripheral on the system.

- Networking

Since incoming packets are asynchronous, most network operations are not specific to a process. Networking must thus be managed by the operating system. The packets are collected, identified, and dispatched before a process can take care of them. The system is in charge of delivering data packets across program and network interfaces. It controls the execution of programs depending on their network activity. All the routing and addressing issues are implemented within the kernel.

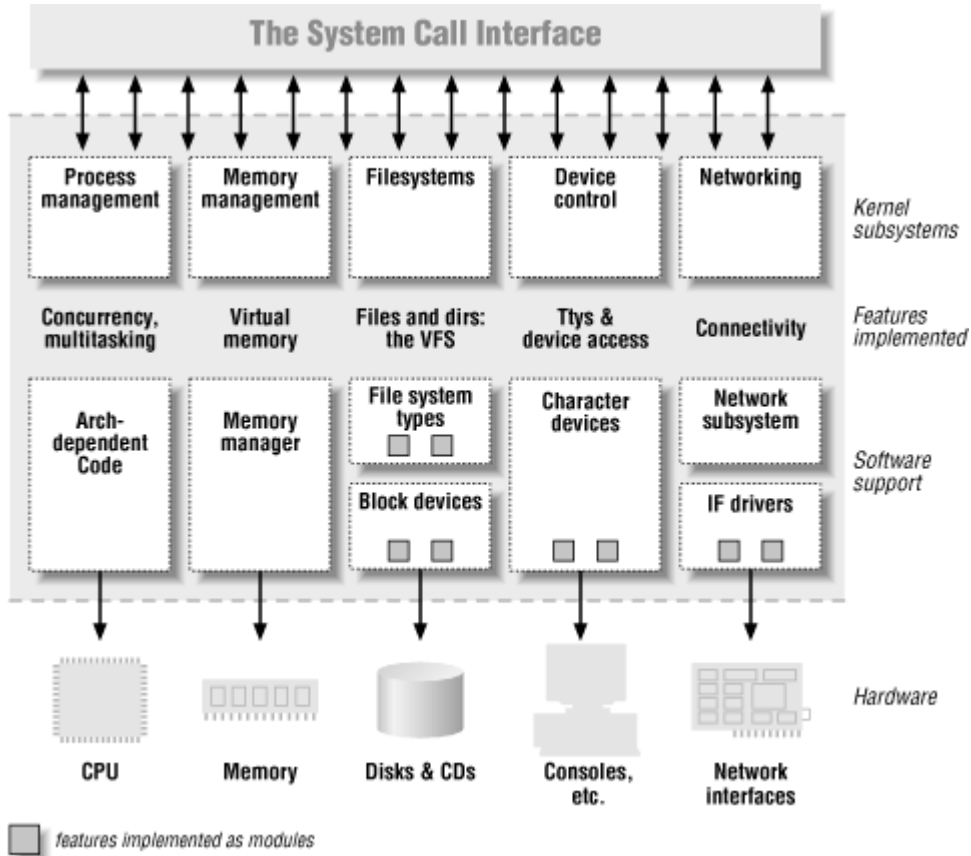


Figure 5 : A split view of the kernel [DEVDR1]

6.1.4. Concrete structure of the kernel

These “abstract” subsystems use the following “concrete” components of the module :

- “SCHED” : The Process Scheduler is responsible for controlling the access to the CPU. It ensures that processes will have fair access to the CPU and that necessary hardware actions are performed by the kernel on time.

- “MM” : The Memory Manager permits multiple processes to securely share the main memory system. It also supports virtual memory that allows Linux to support processes that use more memory than is available in the system : Unused memory is swapped out to persistent storage using the file system. It is then swapped back in when it is needed.
- “VFS” : The Virtual File System presents a common file interface to all devices. The variety of hardware is thus abstracted. The VFS supports multiple file system formats, some of which are compatible with other operating systems.
- “NET” : The Network Interface provides an access to several networking standards and to a variety of network hardware.
- “IPC” : The Inter-Process Communication subsystem supports several mechanisms for process-to-process communication on a Linux system.

6.2. Linux device drivers

A device driver is a software abstraction of a device. A device driver can be kernel-mode (in which case it has access to the kernel memory space) or user-mode (in which case it has access to the user memory space).

6.2.1. Classes of devices and modules

Linux distinguishes between three device types : character devices, block devices or network devices. Each module usually implements one of these types, can thus be classified as a char module, a block module or a network module.

- **Character devices**

A character device (char device) is a device that can be accessed by as a stream of bytes, like a file. A char device driver is in charge of implementing this behaviour. A typical character driver implements at least the *open*, *close*, *read*, and *write* system calls.

- **Block devices**

Block devices are also accessed by filesystem nodes in the “/dev” directory. A block device is a device that can host a filesystem, such as a disk. Linux allows applications to read and write a block device like a char device : it allows the transfer of any number of bytes at a time, whereas in most Unix systems a block device can only be accessed by multiples of a block (a block being one kilobyte of data, or another power of two). The only difference between a block device and a char device in Linux is thus the way that data is managed in the kernel.

- **Network interfaces**

Network transactions use devices that are able to send to and receive data from other hosts. Such interfaces usually are hardware devices, but they can also be a software devices (like the loop-back interface).

A network interface is in charge of sending and receiving data packets, driven by the network subsystem of the kernel (NET), without knowing how individual transactions map to the actual packets being transmitted. Devices do not see the individual streams, but only the data packets.

A unique name is assigned to network interfaces (such as “eth0”), so that they can

be accessed, but that name does not correspond to an entry in the filesystem. Communication between the kernel and a network device driver is done by the kernel calling functions related to packet transmission.

6.2.2. User mode versus kernel mode

In Linux systems, the kernel and the user do not have access to the same memory space. A user mode application is an application that has access to the user memory space, whereas a kernel mode process has access to the kernel memory space.

Sometimes a kernel process may need to read or write from the user memory space, or a user mode application from the kernel memory space.

In such cases, *System calls* or *ioctl calls* can be used for the user mode to communicate with the kernel mode.

6.3. Kernel modules

Kernel modules are device drivers that can be loaded and unloaded while the kernel is running.

6.3.1. Loading and unloading modules

They are not automatically loaded at boot time, and can be loaded using the “*modprobe [module_name]*” command line and unloaded using “*modprobe -r [module_name]*” . The list of modules loaded in the kernel can be seen by typing “*lsmod*”. All these operations require to be done as a super-user.

The kernel modularity is very useful when developing a device driver, as it allows the developer to only recompile his module and then load it without having to recompile the whole kernel.

When configuring the kernel, one can choose not to include a driver, or to include a driver as a module or as part of the kernel. If the driver has been included as a module, it is not automatically loaded at boot time and can be loaded and unloaded at any time (unloading a module in use can cause problems such as a kernel panic).

6.3.2. The DCCP module

I have used the Linux Debian 2.6 DCCP kernel described in Section [4.2]. This kernel contains a DCCP module in the “*net/*” directory.

The kernel tree is described in Appendix A.

I have included the DCCP module as a loadable module in the configuration of my workstation, so as to be able to change the DCCP module code and recompile it without having to recompile the whole kernel. This is also useful not to corrupt the whole kernel by mistake.

6.4. ioctl calls

If a program running in user-mode wants to call a kernel-mode function, it has to solve two problems. First, it must somehow jump across the barrier between user-mode and kernel-mode, and second, it must transfer data in and out. Usually system calls or existing functions such as (*open, read*) are used. But in some cases there is no system call that can achieve the functionality wanted. Then *ioctl* calls can be used. The *ioctl()* function can perform a variety of functions on STREAMS devices. It is a user-application function : It is called from the user space by an application and corresponds to a method implemented in the device driver concerned.

In [REF] Stevens describes the *ioctl()* function as “the catchall for I/O operations”. It

is used for anything that can not be done using the usual I/O functions.

ioctl is not a well documented part of Linux. It is relatively easy to find how to call an existing *ioctl()* function from the user space, but information about how to implement an *ioctl* driver method is scarce. There are still a few guidelines in [DEVDRIV] on how to implement an *ioctl* call, that are presented in this section.

6.4.1. *ioctl()* user function vs *ioctl* driver method

- The *ioctl* call function prototype in user space is showed below :

```
int ioctl(int fildev, int request, ... /* arg */);
```

Returns -1 on error, something else if OK

The *fildev* argument is an open file descriptor referring to a device, for example a socket.

The *request* argument selects the function to be performed. It depends on the device being addressed

The *arg* argument represents additional information needed by the device to perform the requested function. The type of *arg* depends on the request and can either be an integer or a pointer to a device-specific data structure. Sometimes there is no third argument in the function prototype.

- The *ioctl* driver method has the following prototype :

```
int (*ioctl) (struct inode *inode, struct file *filp,  
unsigned int cmd, unsigned long arg);
```

The *inode* and *filp* pointers are values corresponding to the file descriptor *fd* passed on by the application.

The *cmd* argument is passed from the user. It is the same as the *cmd* argument of the user *ioctl* function.

The optional argument *arg* is passed as an unsigned long.

- Most *ioctl* implementations consist of a *switch* statement that allows to choose between different behaviours according to the value of the *cmd* argument.

6.4.2. Choosing an ioctl command number

Commands are associated with a number that has to be unique across the system. When writing the code for *ioctl*, it is necessary to choose a number for each command implemented.

The *ioctl* command corresponding to a command must be chosen among the unassigned number.

6.5. Implementation of an ioctl call

In this section I describe how to implement a simple *ioctl* call to retrieve information from the DCCP module.

Appendix D contains an example *ioctl* driver method.

The *ioctl* call will be made on the DCCP socket (i.e. the *fdes* argument will be the open DCCP socket).

- A structure can be defined to contain the information needed :

```
struct dccp_info
{
    __u8  dccpi_swl;           // Sequence Window Low
    __u8  dccpi_swh;           // Sequence Window High
    __u8  dccpi_awl;           // Acknowledgement Window Low
    __u8  dccpi_awh;           // Acknowledgement Window High
};
```

The optional argument *arg* of the *ioctl()* function will be a pointer on this structure.

- Choice of a command name and number

The command name will be passed as the *cmd* argument in the *ioctl()* function. I have chosen to call this command `READ_INFO_DCCP`. This is not a standard name for an *ioctl command* but it does not matter here as I will be the only one using it. I have chosen the first free number from the list of reserved numbers in *include/linux/sockios.h* and defined the command in this header file :

```
#define READ_INFO_DCCP    0x89F1    /* to read info from dccp */
```

- Passing the structure from the kernel space to the user space

copy_to_user() is a function that can be used to pass a structure – or any block of data – from the kernel memory space to the user memory space.

Its prototype is :

```
unsigned long copy_to_user (void __user * to, const void * from,
unsigned long n);
```

The return value is the number of bytes that could not be copied. On success, this is zero.

Arguments :

- *to*: this argument is the destination address in user space

- *from* is the source address in kernel space

- *n* is the number of bytes to copy

- *ioctl* driver method skeleton in DCCP module

DCCP module already contains an *ioctl method* skeleton in *net/dccp/dccp_proto.c*, which only provides a debugging function (that prints debug messages in a kernel log):

```
int dccp_ioctl(struct sock *sk, int cmd, unsigned long arg)
{
    dccp_pr_debug("entry\n");
    return -ENOIOCTLCMD;
}
```

- Testing the debug *ioctl* call

To test the *ioctl* method, I have used the *Client.c* and *Server.c* files provides with the DCCP implementation for the kernel 2.4.

Unfortunately, I have issues with the memory mapping that I have not been able to solve yet.

Conclusion

This document presents the new issues raised by wireless Internet telephony and the interest of adapting to the network conditions to improve voice quality. This adaptivity can be implemented at different levels. This document presents investigations concerning the adaptation at the transport layer (through the use of DCCP and its congestion control mechanisms) and at the application layer, by varying the codec bit rate.

DCCP is a new transport-layer protocol that has not been standardized yet. It has been designed to meet the specific requirements of real-time streaming applications while implementing congestion control mechanisms. To date, there is no mature DCCP implementation. None of them has reached a production quality, nor provides feedback to user applications.

Speex codec implements variable bit rate encoding and decoding, and has been targeted for voice over IP : It provides specific features such as voice detection. It also has the considerable advantage of being free from software patents and it is thus the codec chosen to implement the adaptive VoIP client.

Technical details were also given on how to retrieve information from DCCP, that the client will use to adapt its codec rate.

Future work

The need for a feedback retrieval from DCCP has been acknowledged by Arnaldo de Melo, the developer of the Linux kernel 2.6 DCCP implementation, and added to the DCCP Project “TO DO” list.

References

- [DCCP] : E. Kohler, M.Handley, S. Floyd and J. Padhye. Datagram Congestion Control Protocol, IETF Internet Draft, work in progress, March 2005

- [CCID2] : Sally Floyd and Eddie Kohler. Profile for DCCP Congestion Control ID 2: TCP-like Congestion Control, IETF Internet Draft, work in progress, March 2005

- [CCID3] : Sally Floyd, Eddie Kohler, and Jitendra Padhye. Profile for DCCP Congestion Control ID 3: TFRC Congestion Control, IETF Internet Draft, work in progress, March 2005

- [WebDCCP] : Datagram Congestion Control Protocol web page
<http://www.icir.org/kohler/dccp/>

- [SPEEX] : Jean-Marc Valin. The Speex Codec Manual (version 1.0), 23rd March 2003

- [DEVDRIV] : Alessandro Rubini and Jonathan Corbet. Linux Device Drivers, 2nd Edition, June 2001, ISBN 0-59600-008-1

- [STEV] : W. Richard Stevens. Advanced Programming in the Unix Environment, February 2002, ISBN 0-201-56317-7

- [AIMD] : S. Floyd, M. Handley, J. Padhye. A Comparison of Equation-Based and AIMD Congestion Control, May 2000

- [TFRC] : S. Floyd, M. Handley, J. Padhye and J.Widmer. Equation-based

Marie Camier – Development of a DCCP-Based Adaptive VoIP Client – Aug 2005

Congestion Control for Unicast Applications, May 2000

- [TCP] : J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP Throughput : A Simple Model and its Empirical Validation, August 1998
- [ADAP] : A. Trad, Q. Ni and H. Afifi. Adaptive VoIP Transmission over Heterogeneous Wired/Wireless Networks, 2004
- [VBR] : D Chen, S. Garg, M. Kappes, K. Trivedi. Supporting VBR VoIP Traffic in IEEE 802.11 WLAN in PCF mode, July 2002
- [KER] : Ivan Bowman. Conceptual Architecture of the Linux Kernel, January 1998
- [802.11] : IEEE Standard 802.11 Part 11 Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications, 1999
- [802.11a] : IEEE Standard 802.11 Part 11 Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications : High-Speed Physical Layer in the 5 GHz Band, 1999
- [802.11b] : IEEE Standard 802.11 Part 11 Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications : Higher-Speed Physical Layer Extension in the 2.4 GHz Band, 1999
- [802.11g] : IEEE Standard 802.11 Part 11 Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications : Further Higher-Speed Physical Layer Extension in the 2.4 GHz Band, 2001
- [WLAN] : Deploying 802.11 Wireless LANs, 3Com Whitepaper

Appendix A : Code structure of the DCCP 2.6 module in the 2.6 DCCP Kernel

A.1. Source tree of the DCCP driver

Here is the source tree of the DCCP 2.6 driver :

```
kernel-2.6/  
  net/  
    dccp/  
      ccids/  
        ccid_skeleton.c  
        dccp_ccid3.c  
        Makefile  
      ccid.c  
      dccp_diag.c  
      dccp_input.c  
      dccp_ipv4.c  
      dccp_minisocks.c  
      dccp_options.c  
      dccp_output.c  
      dccp_probe.c  
      dccp_timer.c  
      Makefile
```

The DCCP-specific header files are in :

```
include/
```

net/

ccid.h

dccp.h

The *ccids/* subdirectory contains the parts of the code relative to the congestion control mechanisms. Only CCID3 has been implemented at this stage.

A.2. Source tree of the 2.6 DCCP kernel

dccp-2.6/

trunk/

arch/

crypto/

Documentation/

drivers/

fs/

include/

init/

ipc/

kernel/

lib/

Makefile

mm/

net/

security/

scripts/

sound/

Marie Camier – Development of a DCCP-Based Adaptive VoIP Client – Aug 2005

usr/

tags/

branches/

The interesting directories when programming for this project are *trunk/net/*, which contains the DCCP driver, and *trunk/include/* that contains header files.

Appendix B : Linphone

This appendix contains information on the linphone code structure, on the linphone installation, and a linphone user-manual with screenshots.

B.1. The linphone code structure

The main modules and libraries in the Linphone source tree are :

- console/ : is the code for the console version of linphone.
 - linphonec.c is the main file for the console version of linphone.
 - sipomatic.c / sipomatic.h contains the code for sipomatic, the test program that auto-answers linphone calls.

- coreapi/ : is the central point of linphone, which handles the relationship between SIP signalisation and media streaming. It contains an API to create a SIP phone.

- developper-docs/ : contains documentation.

- exosip/ : is user-agent library based on the osip SIP stack (<http://osip.atosc.org>).

- gnome/ : is the directory that contains the core files (GUI) of linphone. It uses all libraries described above.
 - interface.c is the file generated by glade to construct the graphical interface.
 - callbacks.c contains the Gtk+ callbacks and callbacks of the SIP library (these functions are called when we are invited, for example)
 - support.c is auto-generated by Glade.

- `propertybox.c` : code to handle the property box
 - `presence.c` : code to handle the presence box
 - `addressbook.c` : code to handle the address book
 - `linphone.c` : main code of gnome interface
 - `applet.c` : gnome applet.
 - `main.c` : contains the `main()` function of linphone's gnome interface, and the `gtk_main()` call.
-
- `gsm/` and `lpc10-1.5/` are libraries that implement the GSM and LPC-10 voice codecs. These libraries are used by the `mediastreamer`.
 - `mediastreamer` : is the library that handles all media operations such as RTP streaming from a file, from a soundcard, with codec transcoding. It is a framework library for audio and video processing. It contains codec objects to compress audio and video streams.
 - `media_api/` is meant to become an easy-to-use library to handle audio and video sessions. At the moment this library is still in early stages of development and is unused by the core application.
 - `oRTP` : is a LGPL licensed C library implementing the RTP protocol (RFC1889).It is used by the `mediastreamer` to send to and receive streams from the network.
 - `share/` : contains translations, documentation, ring tones and hello sound files.

B.2. Compiling, installing and running Linphone

a) Installation requirements

Compiling Linphone requires that the following programs and libraries are installed :

- `g++-3.4` - The GNU C++ compiler
- `libxml-parser-perl` : Perl module for parsing XML files
- `libosip0-dev` : development files for the SIP library
- `libspeex-dev` : The Speex Speech Codec, development files
- `libspeex1` - The Speex Speech Codec
- `libgnome-dev` - The GNOME libraries -- development package
- `libgnomeui-dev` - The GNOME 2 libraries (User Interface) - development files
- A soundcard and a soundcard driver are also required

These programs and libraries can be installed on a Debian Operating System using the `apt-get install` command, that must be executed under the superuser mode :

```
sudo apt-get install [program]
```

b) Installation commands

The installation of Linphone is composed of three steps, corresponding to three command lines to enter from the console, in the `linphone` directory :

- `./configure`
- `make`
- `make install` – requires that you are a superuser

c) Running Linphone

Linphone can be run using the Gnome interface mode, or in a console mode.

- To use the gnome user-friendly interface mode, simply type linphone in the linphone directory
- To use the console mode, type linphonec in the linphone directory.

d) Compilation issues

Depending on the version of gcc installed on the computer, a few compilation problems can occur. They can be solved by changing a few lines in gnome/main.c, gnome/linphone.c and coreapi/linphonecore.c :

- gnome/main.c
- gnome/linphone.c
- coreapi/linphonecore.c

Changing the order of the “#include” lines in the files above have solved the compilation problems I had.

B.3. Implementing DCCP support in Linphone

This is done by changing the socket type from SOCK_DGRAM (UDP socket) to SOCK_DCCP in :

- oRTP/rtpsession.c
- mediastreamer/audiostream.c
- exosip/eXosip.c
- exosip/eXutils.c
- coreapi/linphonecore.c

The following definition line must also be added at the top of each of these files :

```
#define SOCK_DCCP 6
```

Appendix C : How to compile a Debian Kernel

C.1. How to compile a Debian kernel

This appendix is based on a very simple tutorial by Falko Timme, available in :

http://www.projektfarm.com/en/support/howto/debian_kernel_compile.html

1) First you should login to your machine as root.

2) Then install the prerequisites needed to compile the kernel :

```
apt-get install kernel-package ncurses-dev fakeroot wget bzip2
```

3) Then, in */usr/src*, get the Linux kernel source you want to compile :

```
wget http://*****.tar.bz2
```

To install the DCCP 2.6 Debian kernel, the URL is ******TODO**

4) To unpack the kernel sources, type

```
tar xjf dccp-2.6.tar.bz2
```

```
cd dccp-2.6/
```

5) To take the configuration of your existing kernel as a starting point for the configuration of your new kernel, you can load the current configuration which is usually in a file under */boot*.

Type : *make menuconfig*

Select *Load an Alternate Configuration File* and enter the location of your current configuration file.

The configuration of your current kernel will be loaded, and if needed you can change the configuration through the menu. Then, save your new kernel configuration.

6) Then run the following commands:

```
make dep
```

```
make-kpkg clean
```

```
fakeroot make-kpkg --revision=custom.1.0 kernel_image
```

7) If there is an error during the compilation, type

```
make clean
```

and go back to 5) and change the kernel configuration where the errors occur.

8) If no error occurs you will find the new kernel as a Debian package called

```
kernel-image-2.4.23_custom.1.0_i386.deb under /usr/src.
```

9) Type *cd ../* and install the new kernel by typing :

```
dpkg -i kernel-image-2.4.23_custom.1.0_i386.deb
```

10) Reboot the machine.

C.2. Issues specific to compiling the 2.6 DCCP Debian kernel

The 2.6 DCCP Kernel does not have a classic structure but the following one :

```
dccp-2.6/
```

```
    trunk/
```

tags/

branches/

The *tags/* and *branches/* directories are empty, and the kernel source code is in the *trunk/* directory. In a classic tree structure, the contents of *trunk/* would be directly under *dccp-2.6/*. To compile the kernel following the method given above, it is thus necessary to create a link called *linux* that will point to what is inside *trunk/* using the Linux “*ln*” command.

Appendix D : Example of an ioctl driver method

This *scull* implementation of *ioctl* is taken from [DEVDR1]. It transfers the configurable parameters of the device.

```
switch(cmd) {

#ifdef SCULL_DEBUG
    case SCULL_IOCHARDRESET:
        /*
         * reset the counter to 1, to allow unloading in case
         * of problems. Use 1, not 0, because the invoking
         * process has the device open.
         */
        /*/rapport-1.1
        while (MOD_IN_USE)
            MOD_DEC_USE_COUNT;
        MOD_INC_USE_COUNT;
        /* don't break: fall through and reset things */
#endif /* SCULL_DEBUG */

    case SCULL_IOCRESET:
        scull_quantum = SCULL_QUANTUM;
        scull_qset = SCULL_QSET;
        break;

    case SCULL_IOCSQUANTUM: /* Set: arg points to the value
*/
        if (!capable(CAP_SYS_ADMIN))
            return -EPERM;
        ret = __get_user(scull_quantum, (int *)arg);
        break;

    case SCULL_I OCTQUANTUM: /* Tell: arg is the value */
        if (!capable(CAP_SYS_ADMIN))
            return -EPERM;
        scull_quantum = arg;
        break;

    case SCULL_I OCGQUANTUM: /* Get: arg is pointer to result
*/
        ret = __put_user(scull_quantum, (int *)arg);
        break;

    case SCULL_I OCQQUANTUM: /* Query: return it (it's
```

```
positive) */
    return scull_quantum;

    case SCULL_IOCXQUANTUM: /* eXchange: use arg as pointer
*/
    if (! capable (CAP_SYS_ADMIN))
        return -EPERM;
    tmp = scull_quantum;
    ret = __get_user(scull_quantum, (int *)arg);
    if (ret == 0)
        ret = __put_user(tmp, (int *)arg);
    break;

    case SCULL_IOCHQUANTUM: /* sHift: like Tell + Query */
    if (! capable (CAP_SYS_ADMIN))
        return -EPERM;
    tmp = scull_quantum;
    scull_quantum = arg;
    return tmp;

    default: /* redundant, as cmd was checked against MAXNR
*/
    return -ENOTTY;
}
return ret;
```