國立暨南國際大學通訊工程研究所

碩士論文

於不可靠傳輸協定上以擁塞控制進行網路電話封包傳送之機制研究

The Study of VoIP on Unreliable Transport Protocols with Congestion Control

指導教授：吳坤熹博士

研究生：王嘉裕

中華民國九十八年 一月

致謝

這篇論文能順利完成，首先最要感謝的是我的指導教授 吳坤熹老師。在研究所的這段求學期間，經由吳老師的教導，讓我在學業上獲得不少收穫，並且從老師身上學習到研究執著的精神與敬業的態度，讓我出社會後，面對各種挑戰都能更加的有自信。

此外，要感謝和我一同在實驗室共同為各自的論文努力熬夜、彼此加油打氣的柏州學長，並在我研究遇到瓶頸時，適時的給我意見與幫助。另外也要感謝，在英文上給予我很大幫助的正妹筱婷同學以及散打高手霓雅學妹；還有，總是很照顧我的菖育學長，幽默風趣的鐘逸學長，女排好手的兢真學姐，神龍見首不見尾的文萍學姐，佛心來的文仁同學，很大方的政霖同學，補尾刀的汎嘉同學，高手級的瑋勵學弟，總是用一堆高科技產品的韋霖學弟，攝影達人韋勳學弟，188 的韋立學弟，覺得冷手可以借放口袋的信富學弟，謝謝各位學長姐、同學和學弟妹們的照顧以及指導。我永遠都會記得我在暨南的這一段時光。

論文名稱：於不可靠傳輸協定上以擁塞控制進行網路電話封包傳送之機制研究

校院系：國立暨南國際大學通訊工程研究所　　頁數：66

畢業時間：98 年 1 月　　學位別：碩士

研究生：王嘉裕　　指導教授：吳坤熹博士

中文摘要

隨著數位時代的來臨，網路頻寬的增加以及無線網路的普及，造就越來越多新穎的網路應用服務被開發出來，其中Voice over Internet Protocol (VoIP) 就是近來在網際網路上最熱門的應用之一。傳統上VoIP的聲音訊號是以User Datagram Protocol (UDP) 的協定在網路上傳送，本文中我們提出如何使用一個由Internet Engineering Task Force (IETF) 所新制定的Datagram Congestion Control Protocol (DCCP) 協定，承載經由Real-time Transport Protocol (RTP) 所即時傳送的語音串流；並以Linux上的Linphone網路電話軟體為例，說明如何將DCCP實作應用於一個使用Session Initiation Protocol (SIP) 的開放原始碼網路電話，達到在網路上進行雙方通話的功能。

關鍵詞：資料封包擁塞控制協定(DCCP)、即時傳輸協定(RTP)、會議初始協定(SIP)、網路電話(VoIP)

Title of Thesis: The Study of VoIP on Unreliable Transport Protocols with Congestion Control

Name of Institute: Graduate Institute of Communication Engineering, National Chi Nan University Pages: 66

Graduation Time: 01/2009 Degree Conferred: Master

Student Name: Jia-Yu Wang Advisor Name: Quincy Wu

Abstract

In the digital era, the increase of network bandwidth and the ubiquitous wireless access facilitates the creation of more and more innovative network services. Among these services, Voice over Internet Protocol (VoIP) is surely one of the most popular and successful real-time multimedia services on the Internet. For decades, User Datagram Protocol (UDP) has been adopted to transport the voice streams of VoIP applications on the Internet. In this thesis, we presented how the Datagram Congestion Control Protocol (DCCP), which was recently developed by Internet Engineering Task Force (IETF), can be utilized by the Real-time Transport Protocol (RTP) to transport real-time audio streams. Unlike UDP which is a connectionless protocol, DCCP is a connection-oriented protocol which requires both ends to establish a connection before they can begin sending and receiving packets. We proposed the design and architecture of a VoIP application running on DCCP, and take Linphone, an open-source Internet VoIP phone on Linux, as an example to illustrate how to apply DCCP to establish a bidirectional Session Initiation Protocol (SIP) communication.

Keyword: DCCP, RTP, SIP, VoIP

Table of Contents

Figure Index

Table Index

1. Motivation

As the recent advancement of Internet technologies, the network applications have been evolved from text messages delivery like emails, to multimedia applications such as audio and video messages. Among them, Voice/Video over Internet Protocol (VoIP) is a technique to deliver audio and video streams through the Internet or any network using Internet Protocol (IP) technology. Because of its low cost in communication, flexibility for extension, and brand-new coding/decoding technologies to enhance the voice quality, VoIP has demonstrated itself as a technology which can potentially compete with traditional telecommunication services.

There are generally two different modes in delivering multimedia data on the Internet. That is, the *download mode* and the *streaming mode*. For the download mode, it usually takes a long time (a few minutes or even a few hours) to completely download the multimedia information, especially when the network only has limited bandwidth. Therefore, this approach always implies a significant delay before the multimedia session starts. On the contrary, the streaming mode need not wait for the whole files to be completely downloaded. The starting delay only takes a few seconds and then the users can begin listening to or watching the online multimedia audio/video file. Apparently, for users to listen to a live broadcast program or to participate a video conference through the network, it is crucial to send and receive streaming media in real time.

In contrast to the real-time multimedia applications which are very sensitive on the transmission delay, most traditional Internet services we use today, such as the web browsing and E-mail delivery, utilize Transmission Control Protocol (TCP) as the transport

protocol. TCP adopts mechanisms like *flow control* and *re-transmission*. Before the source node receives the acknowledgment from the destination node, the flow control mechanism will postpone delivering successive data. Moreover, when there is any packet loss, TCP will re-transmit the lost packet. All these extra mechanisms will certainly cause longer delay, which makes TCP inappropriate for real-time multimedia applications. Table 1 shows the comparison between four protocols in the transport layer. Note that both TCP and the Stream Control Transmission Protocol (SCTP) [11] are reliable transport protocols.

Table 1. Comparison of Transport Protocol

	TCP	UDP	SCTP	DCCP
Reliable	O	X	O	X
Connect-oriented	O	X	O	O
Congestion Control	O	X	O	O
Sequence Number	O	X	O	O
Message-oriented	X	O	O	O

Currently most Internet real-time applications transport multimedia data with Real-time Transport Protocol (RTP)[1][2] over User Datagram Protocol (UDP). Compared with the reliable TCP protocol, UDP is an unreliable transport protocol which can reduce the communication overhead of connection establishment and re-transmission. Considering the VoIP application, which delivers an audio packet in every 10-40 ms, a lightweight protocol like UDP is a better choice. Although UDP does not guarantee that all audio packets will arrive at the destination, VoIP users can generally tolerate the short audio interruption caused by the packet loss and ask the other party to repeat the sentence again. By doing so, users can get the lost information to be “re-transmitted” by themselves, even though there is no re-transmission mechanism in UDP. Therefore, for the past decades, multimedia applications such as audio streaming and video conferences

generally use UDP as the transport protocol, instead of a reliable protocol like TCP.

One problem of UDP is that, because it does not employ the congestion control mechanism. When there is congestion in the network, UDP will not detect that and will keep on transmitting packets, so it will cause more serious congestion, that finally leads to congestion collapse [12] in the network. In recent years, the UDP traffic on the Internet increases dramatically as multimedia applications getting more popular. Take the application of Internet Telephony as an example. Each audio packet is very small (e.g. 160 bytes), and a packet will be transmitted in every 20 ms, so there will be 50 UDP packets transmitted in each second. For 1000 users to make phone calls via the network, in a second there will be 50,000 packets transmitted. Therefore, it can be seen that as multimedia traffic transported by UDP increases, it would easily lead to congestion in the network! In view of this, a new transport protocol would be to required which has the congestion control mechanism that UDP does not have, but this protocol need not be reliable like TCP because many applications like VoIP do not require their audio streams to be re-transmitted in case of packet loss.

The Datagram Congestion Control Protocol (DCCP)[4] was recently developed by Internet Engineering Task Force (IETF) for this purpose. It is not a reliable protocol, but it supports congestion control, which make it a good choice to replace UDP as the transport protocol for multimedia traffic. However, it is not straightforward to revise an application running on UDP to be transported by DCCP. Because UDP is connectionless, while DCCP is connection-oriented, the programming style and communication model is totally different. In this thesis, we shall study the model in converting a connectionless communication model to a connection-oriented model, and utilize a VoIP application as an example to illustrate how this can be done.

The remaining of this article is organized as follows. Section 2 will briefly introduce

some protocols, including the DCCP connection establishment and termination and data delivery, and have an overview of DCCP packet types. Moreover, the basic principles of the Session Initiation Protocol (SIP)[3], Session Description Protocol (SDP)[17][18] and Real-time Transport Protocol (RTP) will also be described. Section 3 will describe the Application Programming Interface (API) of DCCP communication.

Section 4 will illustrate the software architecture and design which we proposed to transport VoIP applications on DCCP and introduce an open-source Internet phone Linphone, which we took as an example to illustrate the process for porting a VoIP application which was running on UDP to utilize DCCP as the transport protocol. Furthermore, we will illustrate how an open-source network protocol analyzer software Wireshark[16] can be modified to analyze RTP packets over DCCP. Section 5 will evaluate the DCCP performance, focusing on how DCCP flows affect TCP flows and how UDP flows affect TCP flows. Finally, Section 6 concludes our work and addresses some possible future works.

2. Background & Related work

2.1 Datagram Congestion Control Protocol (DCCP)

In this section we will introduce the connection initiation and termination in DCCP and the data delivery, and explain the packet types used with the connection flow.

Datagram Congestion Control Protocol (DCCP) is a new Layer 4 protocol proposed by Internet Engineering Task Force (IETF) and its main purpose is to replace UDP in transporting multimedia streaming data. Similar to UDP, DCCP is an unreliable transmission protocol with no in-order delivery or re-transmission mechanisms. Therefore, it is appropriate for real-time multimedia applications. DCCP transmission model is connection-oriented. The connection can be initiated or terminated by both ends. Each DCCP packet contains a Sequence Number so that any packet loss can be detected. The major difference between DCCP and UDP is that, it can detect the congestion in the network and activate the congestion control mechanism to prevent the hosts from sending more packets which worsen the network situation. In addition, to support the divergent need of different applications, DCCP can work with different congestion control algorithms. Currently DCCP provides two kinds of congestion control algorithms, which are identified as CCID2 (TCP-Like Congestion Control) [5] and CCID3 (TCP-Friendly Rate Control) [6]. According to its characteristic, DCCP is appropriate for multimedia applications like video streaming, audio streaming and Internet telephony.

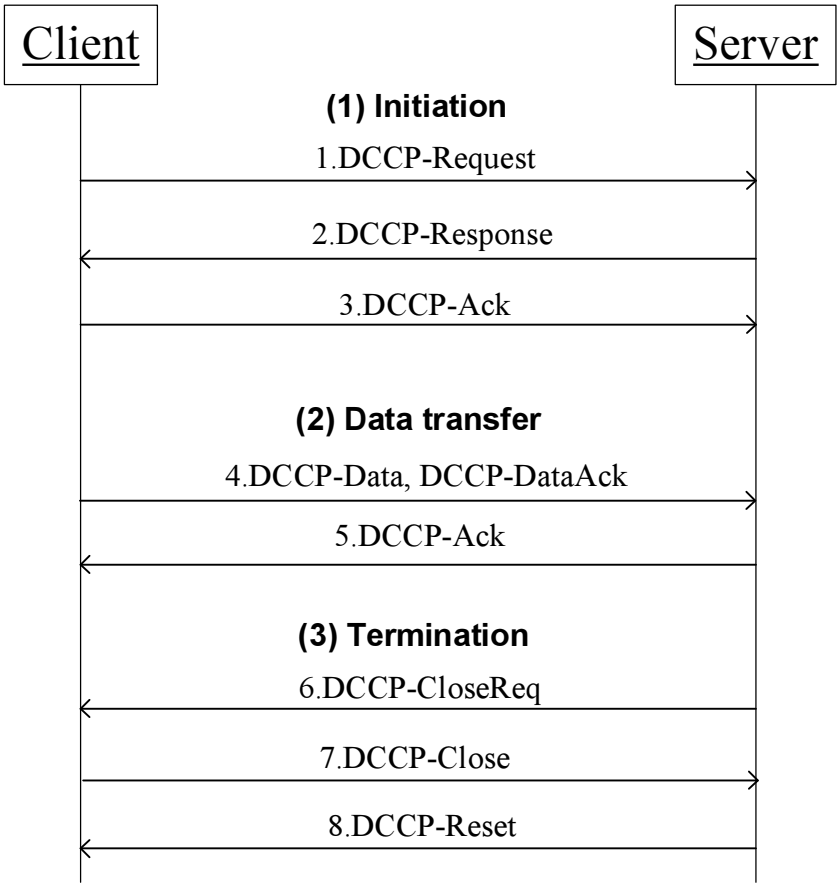


Figure 1. DCCP Packet Exchange Flowchart

The DCCP connection flow is similar to TCP. It is divided into three states: Initiation, Data transfer, and Termination. Suppose a host (Client) wants to establish a DCCP connection to another host (Server). Figure 1 shows the flow of connection initiation and termination in DCCP and data delivery. It contains the following steps:

1. The Client sends a DCCP-Request packet to the Server, indicating it would like to initiate a connection. This request will make the system to enter an "Initiation" state and activate a process of a three-way handshake.
2. After the Server receives the packet, it will send a DCCP-Response message back to the Client in response.
3. After the Client receives the DCCP-Response it will send a DCCP-Ack message to

acknowledge the connection establishment. Now the connection between the Client and the Server is successfully established. The system transits to the "Data transfer" state and data delivery can start.

4. During the data transfer, the Client sends DCCP-Data packets or DCCP-DataAck packets to the Server.
5. After the Server receives the packet, it will send a DCCP-Ack message to the Client.
6. When the Server wants to terminate the connection, it will send a DCCP-CloseReq packet to the Client, which initiated the connection. If it is the Client that wants to terminate the connection, Step 6 is ignored and the action in Step 7 is executed directly.
7. After the Client receives the DCCP-CloseReq request, it will send a DCCP-Close packet to the Server. (If it is the Client which wants to terminate this connection, it will send the DCCP-Close packet directly.)
8. After the Server receives the DCCP-Close packet, it will reply a DCCP-Reset packet and close the DCCP connection.

Table 2 lists the ten DCCP packet types mentioned above and their behaviors.

Table 2. DCCP Packet Types

Packet Types	**Explanation**
DCCP-Request	Sent by the client to initiate a connection.
DCCP-Response	Sent by the server in response to a DCCP-Request.
DCCP-Data	Used to transmit application data.
DCCP-Ack	Used to transmit pure acknowledgements.
DCCP-DataAck	Used to transmit application data with piggybacked acknowledgement information.
DCCP-CloseReq	Sent by the server to request the client to close the connection.
DCCP-Close	Used by the client to close the connection.
DCCP-Reset	Used to terminate the connection, either normally or abnormally.
DCCP-Sync	Used to resynchronize sequence numbers after large bursts of loss.
DCCP-SyncAck	Response of DCCP-Sync to resynchronize sequence numbers.

The DCCP header contains the Source Port, the Destination Port, the Sequence Number, etc. The DCCP generic header takes different forms depending on the value of the Extended Sequence Numbers bit (denoted as X). If X is equal to one, the Sequence Number field is 48 bits long, and the generic header takes 16 bytes, as shown in Figure 2:

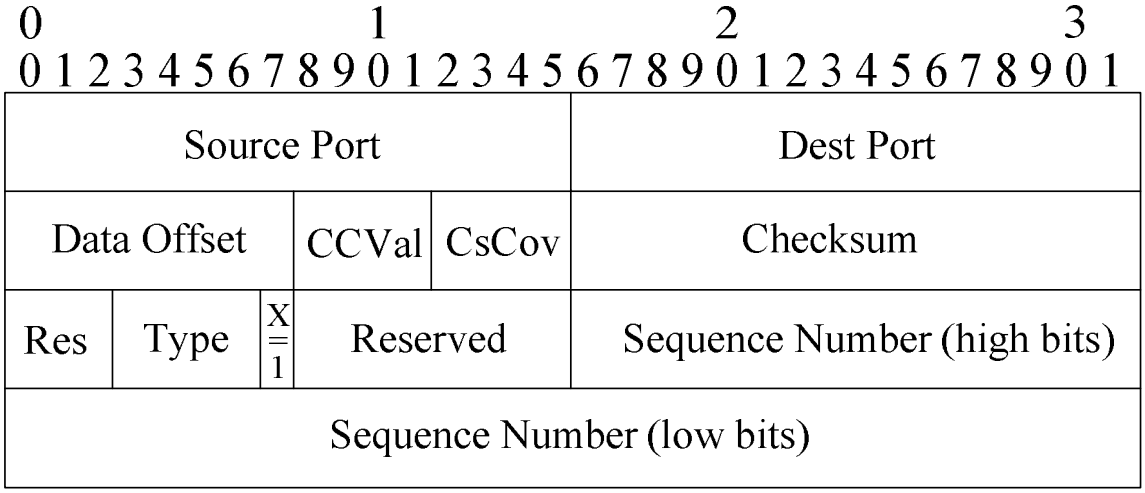


Figure 2. DCCP Header

2.2 Session Initial Protocol (SIP)

Session Initial Protocol (SIP) is an application-layer control protocol proposed by Internet Engineering Task Force (IETF). SIP is used to establish, modify, and terminate multimedia sessions such as Internet telephone calls. In SIP, users are identified by Uniform Resource Identifiers (URIs), whose formats are similar to email addresses. In each session, the URI is resolved to an IP address by the SIP proxy server. After SIP establishes a session, some other protocols (e.g., the Real-time Transport Protocol (RTP)) will be required to transport the multimedia data. SIP also requires other protocol, such as the Session Description Protocol (SDP), to describe the capabilities of session participants. A SIP server is responsible for interpreting incoming SIP packets and setting up sessions between callers and callees.

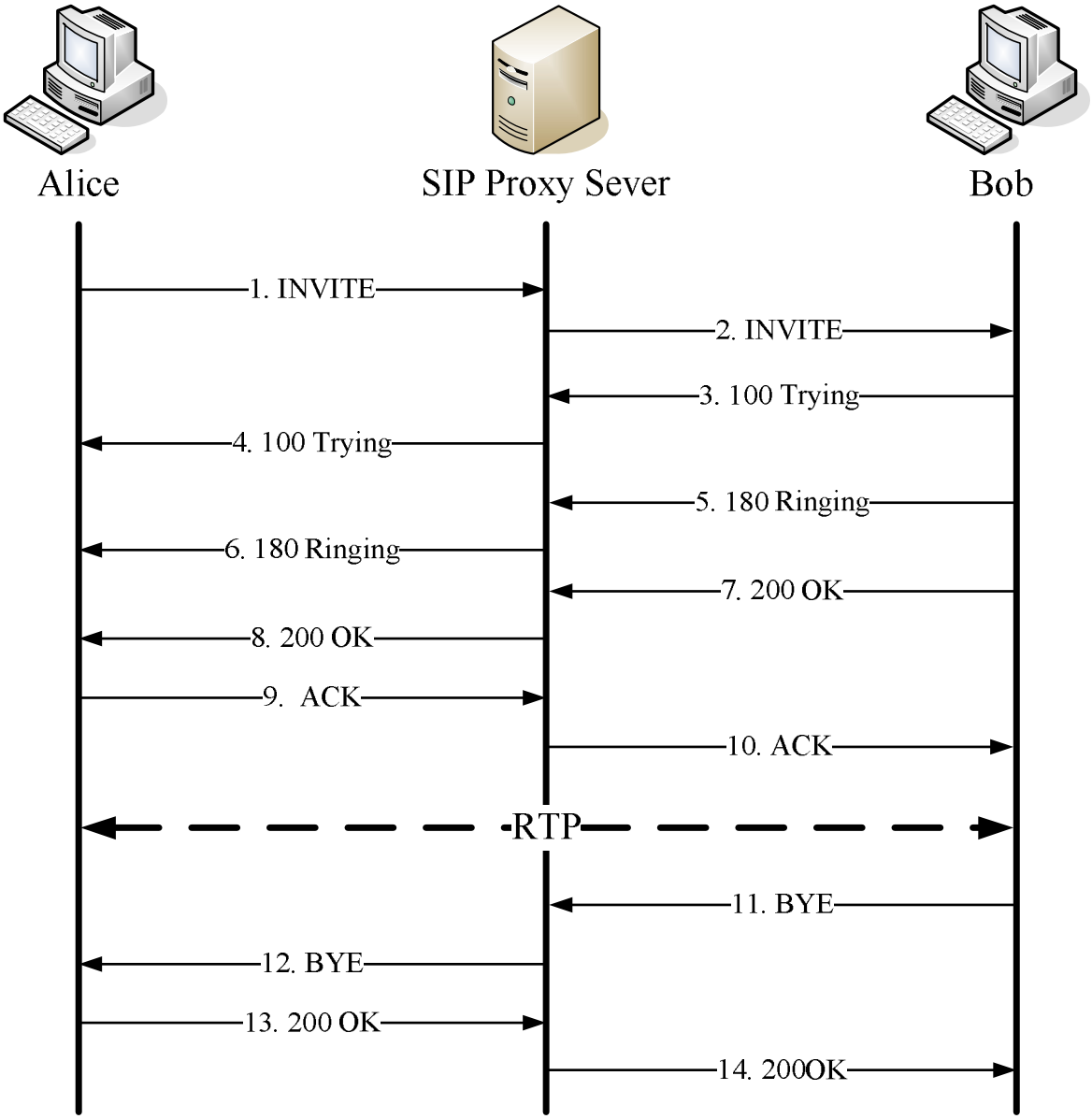


Figure 3. SIP Session Flowchart

Figure 3 illustrates a typical SIP session. When Alice and Bob want to communicate, the call will be established and terminated as follows:

1. First, Alice sends an INVITE message to the proxy server.
2. The proxy server forwards the INVITE message to Bob.
3. When Bob receives the INVITE message, it replies a 100 Trying message.
4. The proxy server forwards the 100 Trying message to Alice.
5. Bob also sends a 180 Ringing message to Alice to indicate that its IP phone begins ringing.

6. The proxy server forwards the 180 Ringing message to Alice.
7. When Bob picks up the phone and answers the call, it replies a 200 OK message.
8. The proxy server forwards the 200 OK message to Alice.
9. When Alice receives a 200 OK message, it sends an ACK message.
10. After Bob receives the ACK message, they use the RTP protocol to send their voice to each other.
11. Assume Bob wants to terminate this call; it sends a BYE message to Alice.
12. The proxy server forwards the BYE message to Alice.
13. After Alice receives the BYE message, it sends a 200 OK message to Bob.
14. After Bob receives a 200 OK message, the call is terminated.

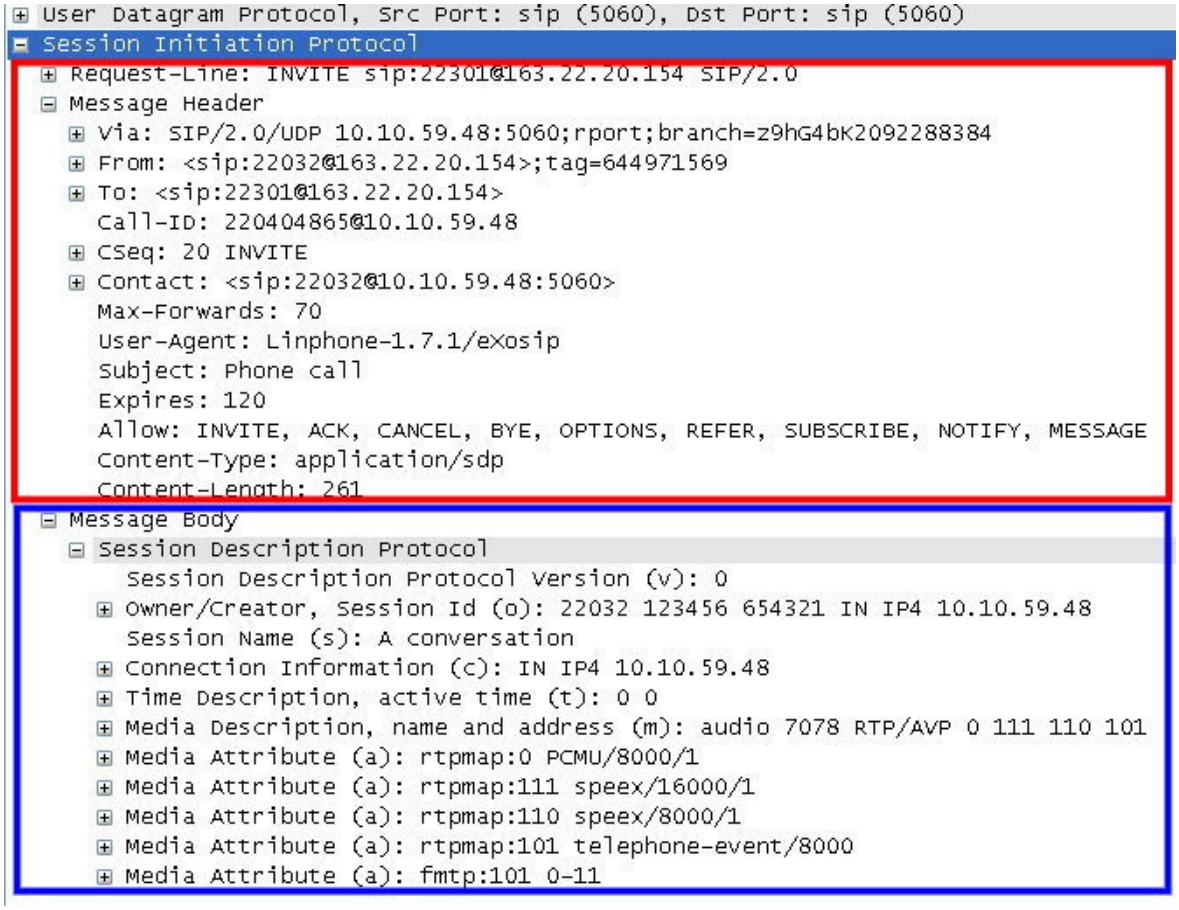


Figure 4. A SIP Message

As shown in Figure 4, a SIP message consists of three parts: a status line (INVITE sip:22301@163.22.20.154 SIP/2.0), the message header (as shown in the upper red

rectangle), and the message body (as shown in the lower blue rectangle). In the message header, there are lots of fields about the route handling (where the packet comes from and where the packet is sent to). The message body in SIP uses Session Description Protocol (SDP) which we will discuss in the next section.

2.3 Session Description Protocol (SDP)

The Session Description Protocol (SDP) is an ASCII text based protocol for describing multimedia sessions and their related scheduling information. The purpose of SDP is to convey information about media streams in multimedia sessions to allow the recipients of a session description to participate in the session. The designing philosophy of SDP is to make it as general as possible so that it can describe conferences in most network environments. It provides information for two parties to negotiate the media type and encoding format for further communication. We show some important SDP fields in the following table:

Table 3. SDP Fields

v	protocol version
o	originator and session identifier
s	session name
t	time when the session is active
c	connection information
m	media descriptions
a	media attribute lines

Among these fields, two of them contain important information to establish multimedia sessions. The c field contains the connection information including network

type, address type and connection address. The m field contains supporting codec, media type and media port.

2.4 Real-time Transport Protocol (RTP)

Real-time Transport Protocol (RTP) is a protocol designed to support real-time delivery of multimedia data in IP networks. It is extensively used for transporting real-time multimedia data in currently Internet. RTP was defined in RFC 1889[1], and further updated in RFC 3550[2] with enhancements on transmission rules and algorithms so that it could be capable of transporting a great deal of multimedia streams concurrently and handling the issues introduced by Network Address Translation (NAT). Audio-on-Demand, Video-on-Demand, Internet Telephony, and videoconferencing are some popular multimedia applications which utilize RTP.

The size of the RTP header is 12 bytes. The RTP header contains information such as the version of RTP, the payload type, the timestamp, etc, as shown in Figure 5. The initial value of the sequence number is random, and it increments by 1 after a RTP packet is sent. With the sequence number, the receiver can check whether any packet is lost. The value of synchronization source (SSRC) identifies the source of an RTP stream.

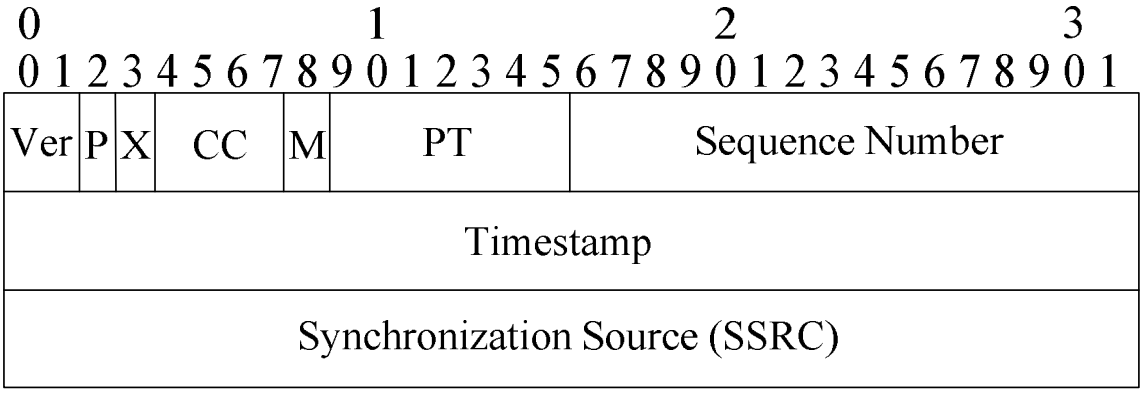


Figure 5. RTP Header

Payload type (PT) specifies the format of RTP payload. RTP supports lots of payload types, such as PCM (A-law and μ-law), G.729, GSM. Figure 6 lists a few codecs and the corresponding payload types. For example, the GSM codec has payload type 3, and G.729 codec has payload type 18. The audio data are carried as payload following the 12-byte RTP header. For the destination to successfully decode the audio data using the correct codec, the receiver must be notified of the payload type in advance, as described in the SDP m field mentioned in the previous section.

PT	Codec
0	PCMU
2	G.721
3	GSM
8	PCMA
18	G.729

Figure 6. RTP Payload Types with Corresponding Codecs

3. Application Programming Interface

3.1 DCCP stack

Many operating systems provide a software object "socket" that connects an application to a network protocol. Take UNIX for example, a program can send and receive TCP/IP messages by opening a socket to read or write data through networks. This simplifies program development because programmers need only worry about manipulating the socket, and they can rely on the operating system to handle the details about transporting messages across the network.

There is a GPL version of DCCP stack [13] included in Linux kernel 2.6.14 and later versions. The implementation of DCCP on Linux is based on the TCP implementation. A server application normally listens to a specific port waiting for connection requests from a client. When a connection request arrives, the client and the server establish a dedicated connection over the port on which they can communicate. During the connection process, the client assigns a local port number, and binds a socket to this communication. The client talks to the server by writing to the socket and gets information from the server by reading from it. Similarly, the server allocates a new local port number (it needs to create a new port number so that it can continue listening to connection requests on the original port). The server also binds a socket to its local port and communicates with the client by reading from and writing to the created socket.

Certainly, the client and the server must agree on the same protocol; in other words, they must agree on the language that transfers the information back and forth through the socket.

A simple DCCP program would define some parameters and utilize some common functions as mentioned below:

- #define SOCK_DCCP 6: The socket type to support DCCP communication.
- #define IPPROTO_DCCP 33: The Protocol field in IP header.
- #define SOL_DCCP 269: Defines the socket option level in DCCP communication.
- `socket()`: Creates a new socket of a certain socket type.
- `bind()`: It is typically used on the server side, and associates a socket with a socket address structure.
- `listen()`: It is used on the server side, and causes a bound DCCP socket to enter the listening state.
- `accept()`: Creates a new connected socket, and returns a new file descriptor referring to that socket.
- `connect()`: It is used on the client side, and assigns a free local port number to a socket.
- `send()`: It is used for sending data to a remote socket.
- `recv()`: It is used for receiving data from a remote socket.

3.2 oRTP

The oRTP [15] library, written in C language, is an implementation of the Real-time Transport Protocol (RTP). It can run on Linux, FreeBSD, and Windows operating system; oRTP also supports partial RTP telephony events (RFC2833).

To develop a simple oRTP program, we use the following functions:

- `ortp_init()`: Initializes the oRTP library.
- `ortp_exit()`: Gracefully uninitialize the oRTP library, including shutdowning the scheduler if it was started.
- `rtp_session_new()`: Creates an RTP session
- `rtp_session_set_payload_type()`: Sets the RTP payload type
- `rtp_session_set_remote_addr()`: Sets the remote IP address and port number
- `rtp_session_recv_with_ts()`: Tries to read the bytes of the incoming RTP stream related to a timestamp.
- `rtp_session_send_with_ts()`: Sends an RTP datagram to the destination containing the data specified by a timestamp.

3.3 mediastreamer2

The mediastreamer2 [14] library contains procedures to handle audio/video streams. It will invoke oRTP to deliver the RTP packets. It also contains audio codecs including G.711, Speex, and GSM, and video codecs including H263-1998, MPEG4. It also supports echo cancellation and reading/writing WAV files during the audio conversation.

To implement a simple mediastreamer2 program, we use the following functions:

- `audio_stream_new()`: Creates an audio stream
- `create_duplex_rtpsession()`: Creates an RTP session
- `audio_stream_start_full()`: Starts an audio stream and sets RTP session parameters
- `ms_filter_call_method()`: Sets audio device and multimedia codec, and prepares for an RTP session

- `ms_filter_link()`: Sets the link to the pre-set data
- `ms_ticker_new()`: Creates a ticker
- `ms_ticker_attach()`: Attaches filter to the ticker

4. Implementation

In this section, we shall take Linphone as an example to explain how to utilize DCCP in delivering audio packets.

4.1 Linphone

Linphone [8] is an open-source IP phone running on Linux. It can send and receive audio, video, and text messages. It uses Session Initiation Protocol (SIP) [3] to establish a conversation session. In addition, it also supports some special features, such as Dual Tone Multi-Frequency (DTMF), Internet Protocol Version 6 (IPv6), and Simple Traversal of UDP through NATs (STUN). Linphone is an ideal platform for developing VoIP applications over DCCP.

As shown in Figure 7, Linphone consists of several modules. It uses *eXosip* to establish SIP sessions, while the *mediastreamer2* module provides procedures to handle audio and video streams. The *oRTP* module sends and receives audio packets via RTP.

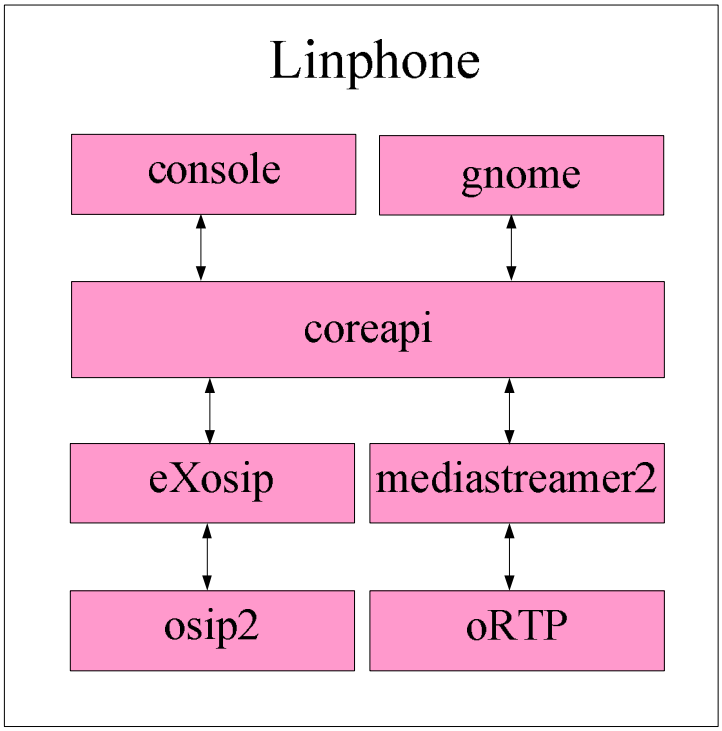


Figure 7. System Components of Linphone

4.2 Call Flow of Linphone

In this subsection, we shall introduce the original program structure of Linphone and the related function interfaces. Based on Figure 8, function calls of Linphone are divided into three portions. Later in Section 4.4 we will explain how to implement DCCP by modifying these three portions.

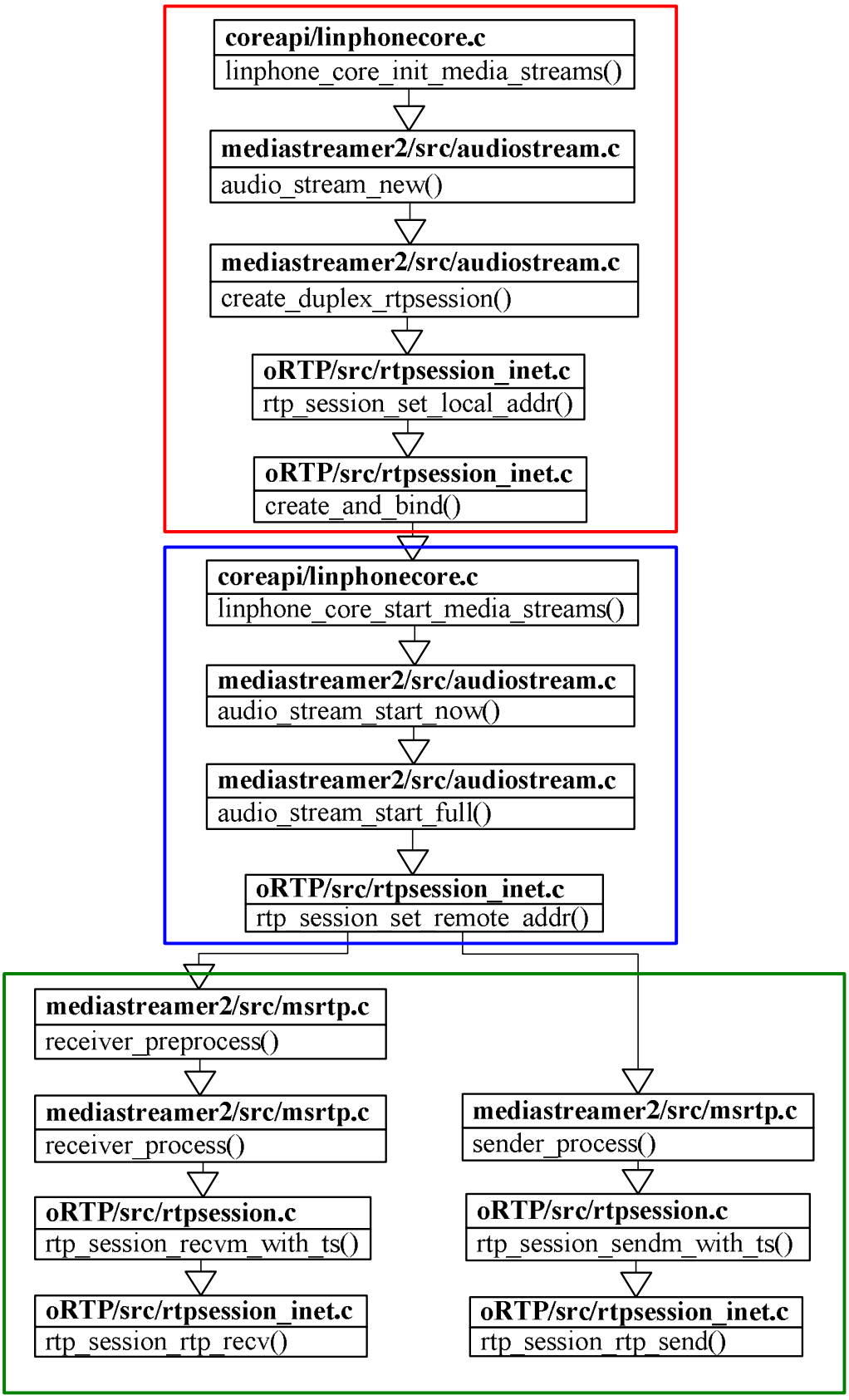


Figure 8. Linphone Original Function Call

As shown in Figure 8, when a SIP connection is created, the function `linphone_core_init_media_streams(…)` in coreapi/linphonecore.c initializes a media stream, and then calls `audio_stream_new(…)` in mediastreamer2/src/audiostream.c to create an audio stream. After that, `create_duplex_rtpsession(…)` creates an RTP session, and calls `rtp_session_set_local_addr(…)` in oRTP/src/rtpsession_inet.c to set the local address, then checks whether it receives a valid port number. If it does, `create_and_bind(…)` is used to create a socket on the specified port; if not, the function `create_and_bind_random(…)` will be invoked to randomly allocate a port and create a socket bound to it.

After the socket is created, in the second portion, `linphone_core_start_media_streams(…)` will start a media stream, and call `audio_stream_start_now(…)` to start an audio stream transmission. After that, it calls `audio_stream_start_full(…)` to configure the parameters of an RTP session, and calls `rtp_session_set_remote_addr(…)` to configure the remote IP address.

Following that, the program will create two threads named Receiver and Sender, respectively, in mediastreamer2/src/msrtp.c. The Receiver thread calls `receiver_preprocess(…)` to configure the payload type, and uses `receiver_process(…)` to start the procedure to receive RTP packets.

To read RTP packets at a constant time interval, `rtp_session_recvm_with_ts(…)` is invoked to set timestamp, and then `rtp_session_rtp_recv(…)` is used to receive RTP packets. On the other side, the Sender calls `sender_process(…)` to start the procedure to send RTP packets, and calls `rtp_session_sendm_with_ts(…)` to set timestamps. Finally `rtp_session_rtp_send(…)` is called to transmit RTP packets.

4.3 Porting Linphone to DCCP

In order to port Linphone from UDP to DCCP, three major modifications must be made in the call flow of original Linphone.

1. Because DCCP transmission is uni-directional, the two nodes participating in a connection are either a Client or a Server. Therefore, when porting Linphone to transmit RTP streams over DCCP, we need to establish two RTP sessions.
2. Moreover, DCCP is connection-oriented, so it is quite different from the original transport protocol of Linphone which utilizes UDP in a connectionless fashion. In DCCP, the server must invoke `accept(…)` to wait for a client to call the function `connect(…)` to establish a connection with the server. Notably, when the server creates a socket, the socket must be set as the *Blocking* mode so that the server will enter the *Accept State* and wait for clients to establish a DCCP connection. If the server did not choose the Blocking mode, the process would directly move to the next step without making any connection. In that case, no DCCP connection is established, so no further packets will be accepted. This connection type of communication is quite different from UDP.
3. When a server establishes a socket, it should specify DCCP as its transport protocol, and replace the related parameters from UDP to DCCP as well.

4.4 Modified Call Flow of Linphone

As described in Section 4.2, there are three portions of Linphone function calls to establish an audio conversation. In the following, we will explain what should be modified to make RTP packets transmit over DCCP.

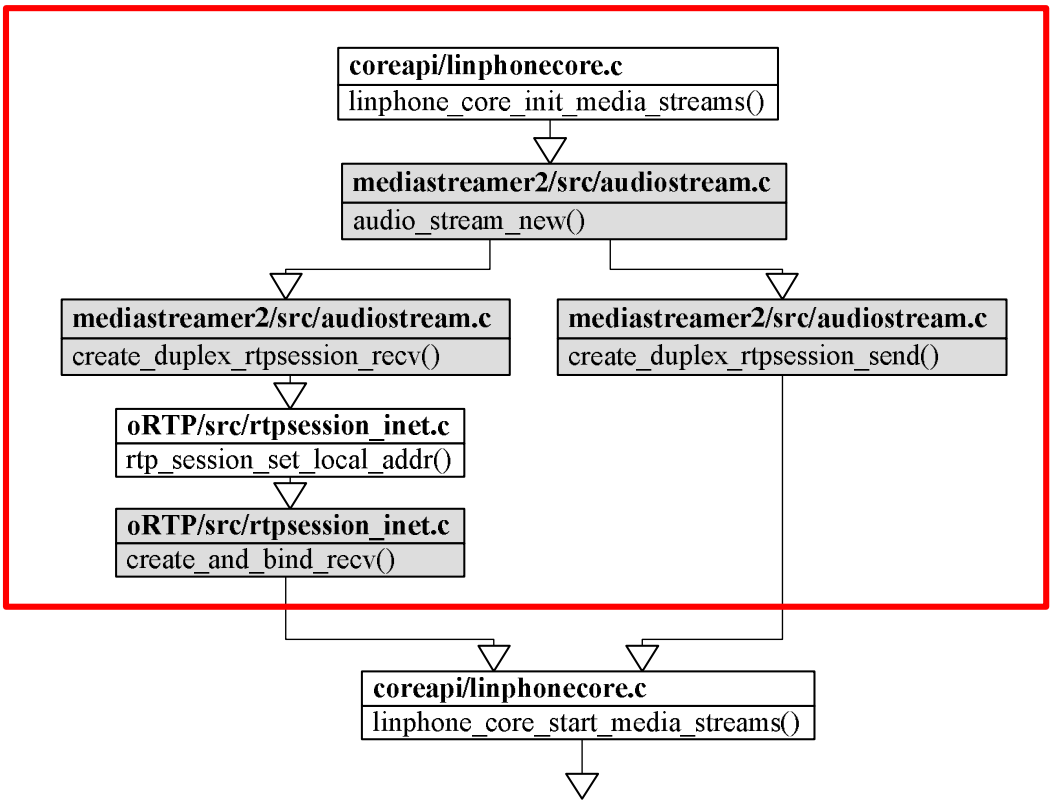


Figure 9. Implementing DCCP Function Call (1)

The modification of the first portion is shown in Figure 9, where the shadowed rectangles indicate the modules that must be modified to support DCCP. We rewrote the function `audio_stream_new(…)` so that after an audio stream is generated , it will create two uni-directional RTP sessions: `create_duplex_rtpsession_recv(…)` is invoked to create an RTP session for receiving audio streams, and `create_duplex_rtpsession_send(…)` is invoked to create another session for sending audio streams. Being a Receiver (left part in Figure 9), it will create an RTP session and call `rtp_session_set_local_addr(…)` to specify its local address, and then call `create_and_bind_recv(…)` to create a socket. When creating a socket, the transport protocol must be specified as DCCP, and related parameters must also be specified as described in Section 3.1. Moreover, the Blocking mode should be specified for the socket. At the Sender part (right part in Figure 9), in order to keep the original structure

of oRTP and Linphone, we do not create sockets at this moment, but at the next portion. This also provides the flexibility to allow users to choose UDP or DCCP as the transport protocol at run time.

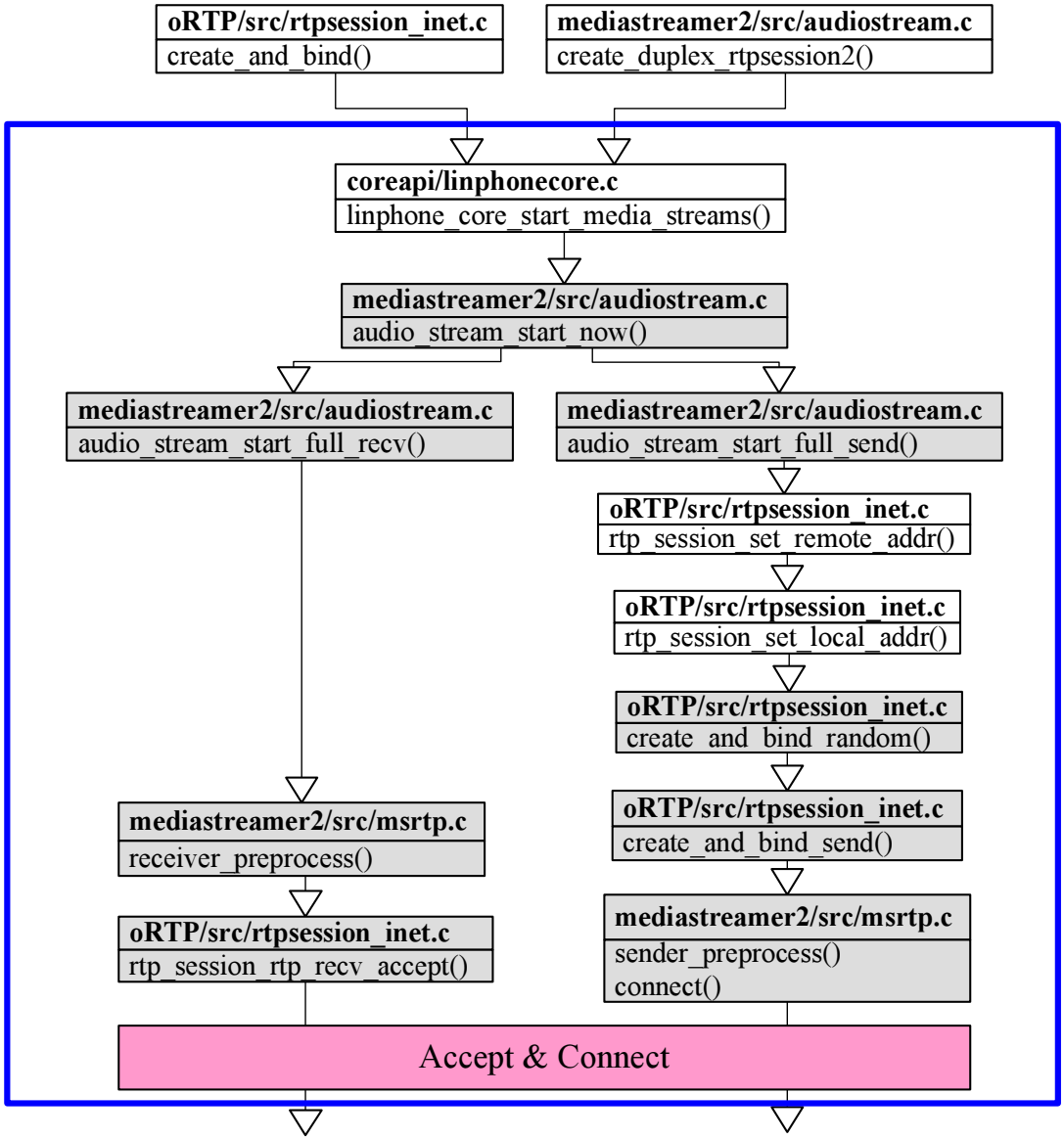


Figure 10. Implementing DCCP Function Call (2)

The modification of the second portion is shown in Figure 10. To deliver media streams over DCCP, we need to modify the function `audio_stream_start_now(…)` to transmit audio streams between the Receiver and the Sender. As shown in the left part of Figure 10, the Receiver invokes `audio_stream_start_full_recv(…)` to specify

the parameters for the RTP session of the Receiver, and creates a thread for the Receiver to receive RTP packets. Then it calls `receiver_preprocess(…)` to set the Receiver payload type, and uses the function `rtp_session_rtp_recv_accept(…)` to enter the accept state to wait for the Sender creating a connection. On the other side, as shown in the right part, the function `audio_stream_start_full_send(…)` is invoked by the Sender to specify the parameters of the RTP session, and creates a thread for the Sender to transmit RTP packets. In the function `rtp_session_set_remote_addr(…)`, it sets the remote IP address of the end host, and checks whether a socket was created. If not, it calls the function `rtp_session_set_local_addr(…)` to specify its local address, and randomly selects a valid UDP port by the function `create_and_bind_random(…)`, and then uses `create_and_bind_send(…)` to create the Sender socket. After that, it calls `sender_preprocess(…)` to connect with the Receiver.

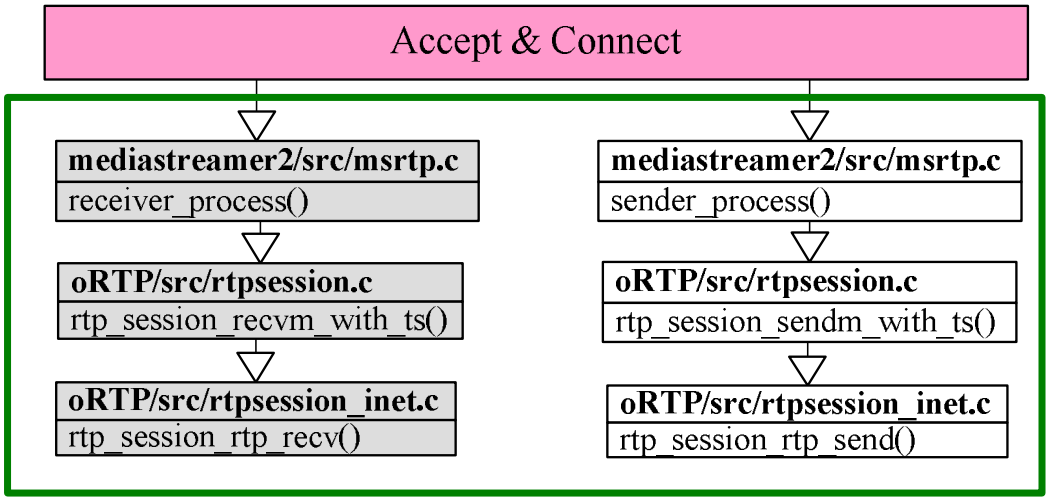


Figure 11. Implementing DCCP Function Call (3)

As shown in Figure 11, in the `Accept & Connect` state, the Receiver calls `receiver_process(…)` and becomes ready to receive RTP packets. This function call will further invoke the function `rtp_session_recvm_with_ts(…)` to set timestamps, and then call `rtp_session_rtp_recv(…)` to receive RTP packets. We

modified these three functions by adding a parameter named `connfd` that returned from function `accept(…)`. Meanwhile, as the right part of Figure 11 shows, the processes of the Sender session just follows the procedure in the original Linphone program so that these corresponding functions need not be changed.

4.5 Wireshark Enhancement

We implemented the DCCP protocol stack on Linphone, and ran the program on two hosts running Linux Fedora 7. The SIP and DCCP packets during the communication were captured by Wireshark. Because current Wireshark cannot analyze RTP packets in DCCP, the captured packets will only be shown as “DCCP data” as in Figure 12. To make it capable of parsing the RTP packets transported by DCCP, some modifications must be made, as we shall explain in Section 4.5.2.

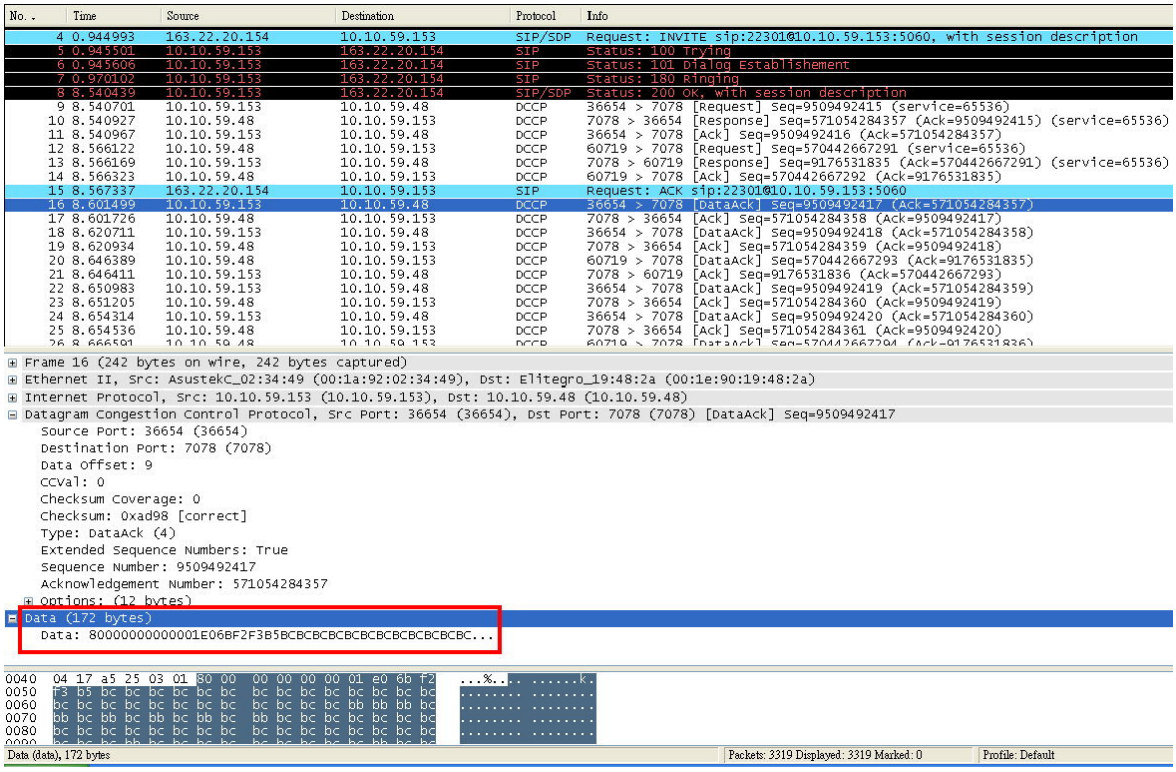


Figure 12. DCCP Packets Captured by Wireshark

4.5.1 Wireshark

Wireshark (known as Ethereal until a trademark dispute in 2006 summer) is a fantastic open-source network protocol analyzer for Unix and Windows. It allows network engineers to examine data from a live network or from a capture file on disk. You can interactively browse the captured data and view summary and detail information for each packet. Wireshark has several powerful features, including a rich display filter language and the ability to view the reconstructed stream of a TCP session. It also supports hundreds of protocols and media types. Wireshark is software that understands the structure of different network protocols. Thus it is capable of parsing single fields in protocol encapsulation and interpreting their meaning. Wireshark uses a library Pcap to capture packets, so it can perform packet capturing on every network adapter supported by Pcap.

4.5.2 SDP Modification for DCCP

To make Wireshark capable of automatically distinguishing the received RTP packets transmitted over UDP or DCCP, we refer to the Internet Draft "RTP and the Datagram Congestion Control Protocol (DCCP)"[19]. It re-defined the Media Description field in SDP, to specify the transport protocol. In the Media Description field in current SDP, **RTP/AVP** specified that UDP is used as the transport protocol, as shown in Figure 13.

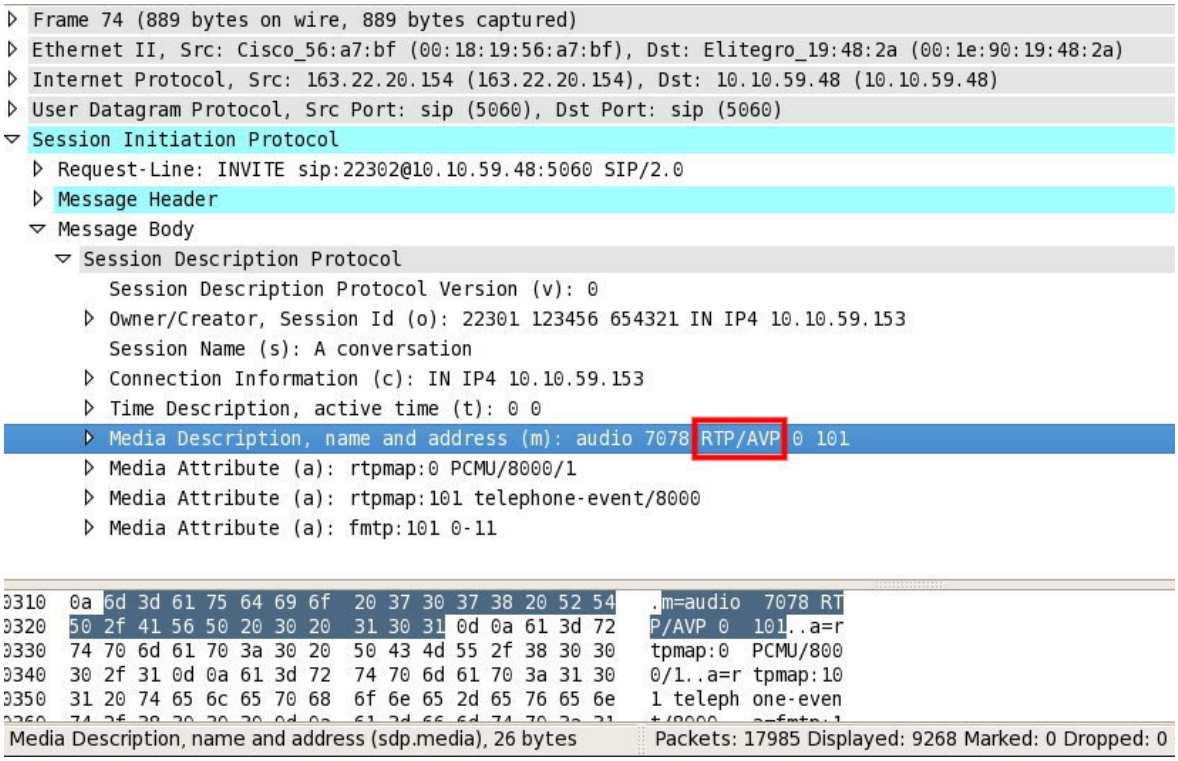


Figure 13. SDP Field for UDP

To use DCCP in delivering RTP packets, it should specify **DCCP/RTP/AVP** at the Media Description field in SDP. So, in Linphone we need to modify the function `sdp_context_add_payload (…)` in coreapi/sdphandler.c by replacing the value from **RTP/AVP** to **DCCP/RTP/AVP**. A sample code in C language is shown below.

```
Void sdp_context_add_payload (sdp_context_t *ctx,
 sdp_payload_t *payload, char *media)
{
        …
        payload->proto = "DCCP/RTP/AVP";
        …
}
```

Figure 14 shows the Media Description field in SDP captured by Wireshark.

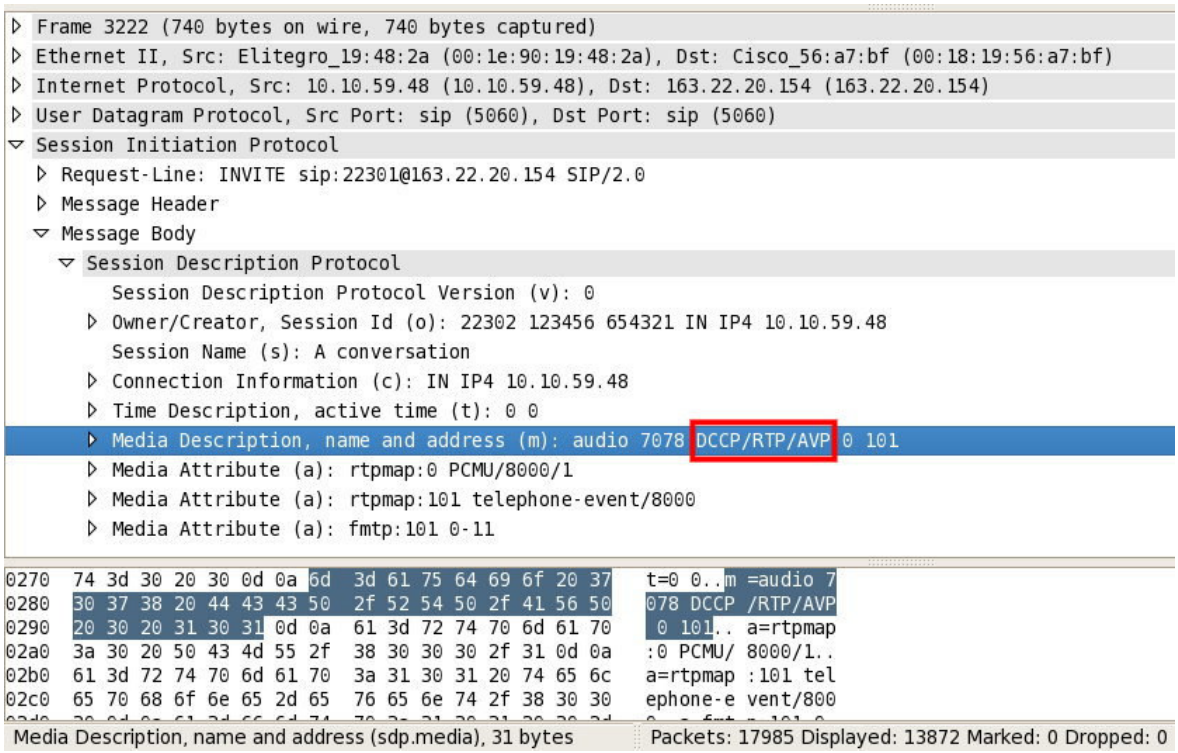


Figure 14. SDP Field for DCCP

In the following section, we will describe how to modify Wireshark to let it capable of analyzing RTP packets when they are transported over DCCP.

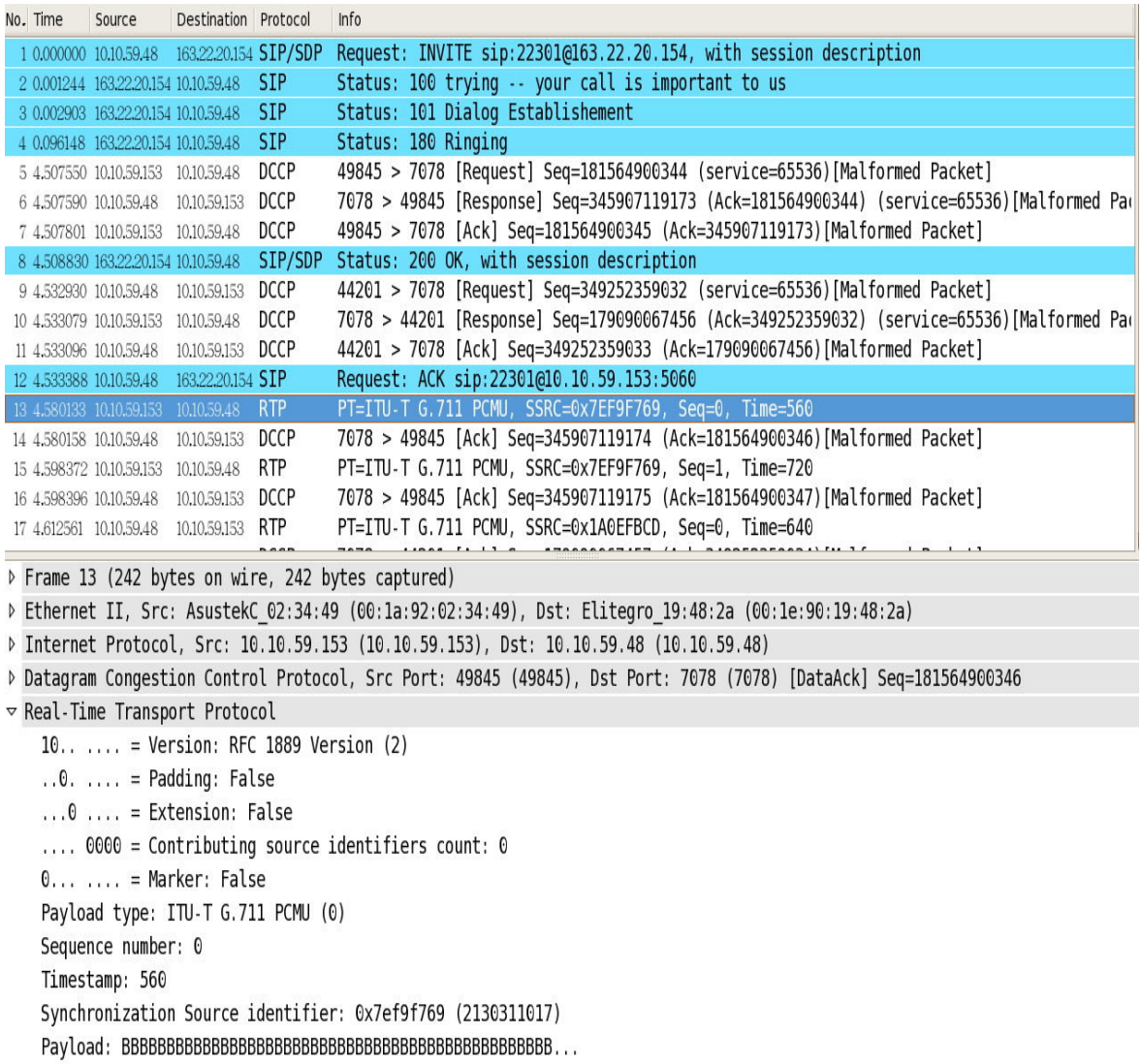
4.5.3 Wireshark Modification

To let Wireshark analyze RTP packets in DCCP datagrams, we need to modify the function `srtp_add_address(…)` in epan/dissectors/packet-rtps.c by replacing the parameter from **PT_UDP** to **PT_DCCP**. See the following two modified functions:

find_conversation(setup_frame_number, addr, &null_addr, **PT_DCCP**, port, other_port, NO_ADDR_B | (!other_port ? NO_PORT_B : 0));

conversation_new(setup_frame_number, addr, &null_addr, **PT_DCCP**,(guint32)port, (guint32)other_port, NO_ADDR2 | (!other_port ? NO_PORT2 : 0));

Figure 15 shows some DCCP packets analyzed by the modified Wireshark.



No.	Time	Source	Destination	Protocol	Info
1	0.000000	10.10.59.48	163.22.20.154	SIP/SDP	Request: INVITE sip:22301@163.22.20.154, with session description
2	0.001244	163.22.20.154	10.10.59.48	SIP	Status: 100 trying -- your call is important to us
3	0.002903	163.22.20.154	10.10.59.48	SIP	Status: 101 Dialog Establishement
4	0.096148	163.22.20.154	10.10.59.48	SIP	Status: 180 Ringing
5	4.507550	10.10.59.153	10.10.59.48	DCCP	49845 > 7078 [Request] Seq=181564900344 (service=65536)[Malformed Packet]
6	4.507590	10.10.59.48	10.10.59.153	DCCP	7078 > 49845 [Response] Seq=345907119173 (Ack=181564900344) (service=65536)[Malformed Pa
7	4.507801	10.10.59.153	10.10.59.48	DCCP	49845 > 7078 [Ack] Seq=181564900345 (Ack=345907119173)[Malformed Packet]
8	4.508830	163.22.20.154	10.10.59.48	SIP/SDP	Status: 200 OK, with session description
9	4.532930	10.10.59.48	10.10.59.153	DCCP	44201 > 7078 [Request] Seq=349252359032 (service=65536)[Malformed Packet]
10	4.533079	10.10.59.153	10.10.59.48	DCCP	7078 > 44201 [Response] Seq=179090067456 (Ack=349252359032) (service=65536)[Malformed Pa
11	4.533096	10.10.59.48	10.10.59.153	DCCP	44201 > 7078 [Ack] Seq=349252359033 (Ack=179090067456)[Malformed Packet]
12	4.533388	10.10.59.48	163.22.20.154	SIP	Request: ACK sip:22301@10.10.59.153:5060
13	4.580133	10.10.59.153	10.10.59.48	RTP	PT=ITU-T G.711 PCMU, SSRC=0x7EF9F769, Seq=0, Time=560
14	4.580158	10.10.59.48	10.10.59.153	DCCP	7078 > 49845 [Ack] Seq=345907119174 (Ack=181564900346)[Malformed Packet]
15	4.598372	10.10.59.153	10.10.59.48	RTP	PT=ITU-T G.711 PCMU, SSRC=0x7EF9F769, Seq=1, Time=720
16	4.598396	10.10.59.48	10.10.59.153	DCCP	7078 > 49845 [Ack] Seq=345907119175 (Ack=181564900347)[Malformed Packet]
17	4.612561	10.10.59.48	10.10.59.153	RTP	PT=ITU-T G.711 PCMU, SSRC=0x1A0EFBCD, Seq=0, Time=640

Figure 15. DCCP Packets Captured by the Modified Wireshark

5. Performance Evaluation

In this section, we describe the testing environment and tools we used in testing the performance of DCCP when it is used to transport VoIP packets.

5.1 The Testing Tool

- Iperf

Iperf is a tool to measure the throughput and the available bandwidth of a network, which allows user to specify the various parameters for testing a network. Iperf will report bandwidth, delay variation, and datagram loss. Iperf was originally developed by NLANR/DAST, and now it is maintained and developed as a SourceForge project.

5.2 The Testing Environment

Our testing environment consists of the following items:

- Hardware:
    1. CPU: Intel Pentium 4 CPU 3.4GHz
    2. Memory: 1GB
- Software:
    1. Operating System: Linux Fedora 7
    2. perl-XML-Parser: 2.34-6.1.2.2.1
    3. oRTP: 0.13.0
    4. speex-devel: 1.2-0.2.beta1

5. readline-devel: 5.2-4.fc7
6. libosip2 :2.2.2
7. mediastreamer2: 2.0.2

5.3 The Measured Results

We used Iperf to generate the TCP flow and similarly used oRTP to generate the flows of DCCP CCID2 and UDP. The network topology is shown in Figure 16, where the 10Mbit/s hub is the bottleneck between two end hosts. We tested the throughput of both the TCP flow and the DCCP CCID2 flow.

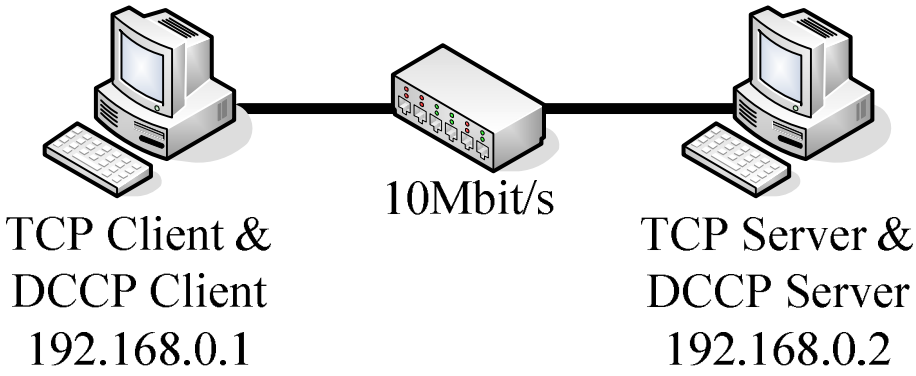


Figure 16. DCCP+TCP Network Topology

In each iteration, we simultaneously generated one TCP flow and one DCCP CCID2 flow with a duration of 30 seconds. The experiment is repeated 100 times. As shown in Table 4, the average transmission rate of TCP and DCCP is 7.74 Mbit/s and 2.02 Mbit/s, respectively.

Table 4. Average Transmission Rate of DCCP+TCP

TCP	7.74 Mbit/s
DCCP	2.02 Mbit/s

Next, with similar network topology shown in Figure 17, we tested the throughput of both the TCP flow and the UDP flow.

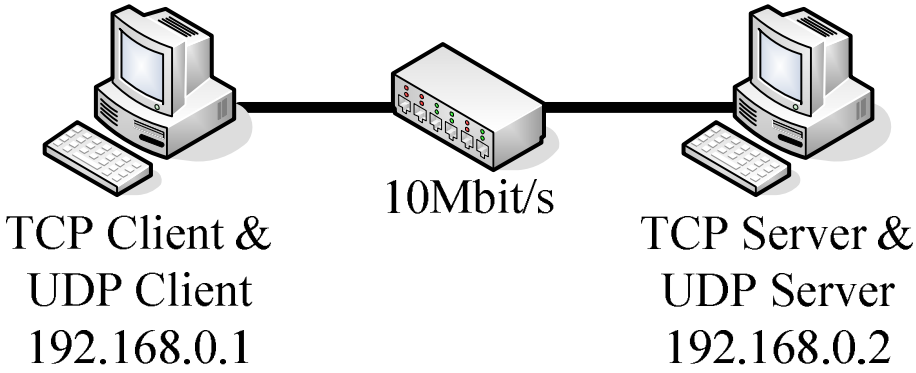


Figure 17. UDP+TCP Network Topology

Similarly, we simultaneously generated one TCP flow and one UDP flow with a period of 30 seconds. As shown in Table 5, the average transmission rate of TCP and UDP is 0.1 Mbit/s and 9.66 Mbit/s, respectively.

Table 5. Average Transmission Rate of UDP+TCP

TCP	0.10Mbit/s
UDP	9.66Mbit/s

As shown in Table 5, the TCP flow has low throughput compared with the UDP flow. This is due to the lack of congestion control mechanism in UDP. From this observation, it can be seen clearly that as the volume of UDP traffic increases, it would be harmful to TCP traffic. On the contrary, DCCP is very friendly to TCP.

Like the experiment result above, DCCP is more friendly than UDP. However in a network with low bandwidth, the VoIP communication may fail because of the low bandwidth. A possible solution is to re-negotiate the codec with the session partner by sending a re-INVITE SIP request with an updated “b=“ line to indicate the new required bandwidth [19].

6. Conclusions and Future work

In this thesis, we presented the system architecture that allows the real-time RTP audio streams to transmit over DCCP. Because the DCCP connections are uni-directional, we must create two RTP sessions for transmitting RTP packets. To verify the architecture that we proposed, we selected a well-known VoIP application Linphone on Linux, and further modified its transport protocol from UDP to DCCP. After our revision, users can use Linphone to call others, and the audio streams are successfully transported over DCCP during the conversation. We also modified Wireshark so that it can parse the RTP packets transported by DCCP. Performance testing shows that DCCP is friendly to TCP, while UDP will easily degrade the performance of TCP applications.

Currently, the modified Linphone that we rewrote can only support DCCP as the transport protocol. In the future, we plan to include both the support of UDP and DCCP on a single Linphone. We are also considering the transition mechanism of DCCP, such as the translator of UDP and DCCP. In addition, presently most Network Address Translation (NAT) implementations can only handle packets which are transported over TCP or UDP. How to efficiently handle DCCP packets through NAT is an interesting problem which deserves further study.

As congestion control mechanisms of DCCP, in addition to aforementioned CCID2 and CCID3, lots of researchers are investigating a new mechanism named CCID4 to handle small packets [draft-ietf-dccp-ccid4-02.txt]. In VoIP applications, generally small packets (160 bytes in G.711, and 20 bytes G.729) are transmitted. Therefore, if this

congestion control algorithm can be applied on Internet telephony, it is expected that the performance can be further improved. Besides, the study on the impact of congestion on Mean Opinion Score (MOS) for VoIP applications running over UDP or DCCP is also an interesting topic in the future.

References

[1] H. Schulzrinne, S. Casner, R. Frederick, V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications," IETF RFC 1889, January 1996.

[2] H. Schulzrinne, S. Casner, R. Frederick, V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications," IETF RFC 3550, July 2003.

[3] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, E. Schooler, "SIP: Session Initiation Protocol, " IETF RFC 3261, June 2002.

[4] E. Kohler, M. Handley, S. Floyd, "Datagram Congestion Control Protocol (DCCP)," IETF RFC 4340, March 2006.

[5] S. Floyd, E. Kohler, "Datagram Congestion Control Protocol (DCCP) Congestion Control ID 2: TCP-Like Congestion Control," IETF RFC 4341, March 2006.

[6] S. Floyd, E. Kohler, J. Padhye, "Datagram Congestion Control Protocol (DCCP) Congestion Control ID 3: TCP-Friendly Rate Control (TFRC)," IETF RFC 4342, March 2006.

[7] C. Perkins, L. Gharai, "RTP and the Datagram Congestion Control Protocol," IEEE International Conference on Multimedia and Expo (ICME) 2006, July 2006.

[8] Linphone [http://www.linphone.org/].

[9] J. Lai, E. Kohler, "Efficiency and late data choice in a user-kernel interface for congestion-controlled datagrams," 12th Annual SPIE Conference on Multimedia Computing and Networking (MMCN '05), January 2005.

[10] X.F. Guo, T.M. Feng, J.Y. Zhou, G.H. Chen, "DCCP Research and Analysis of its Performance," [http://scholar.ilib.cn/Abstract.aspx?A=jsjkx200310033], Computer Science, Vol. 30, No. 10, October 2003.

[11] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M.

Kalla, L. Zhang, V. Paxson, “Stream Control Transmission Protocol,” IETF RFC 2960, October 2000.

[12] C. Albuquerque, B.J. Vickers, T. Suda, “Network border patrol: preventing congestion collapse and promoting fairness in the Internet,” IEEE/ACM Transactions on Networking, Vol. 12, No. 1, February 2004.

[13] DCCP Socket [http://www.linuxfoundation.org/en/Net:DCCP].

[14] mediastreamer2 [http://webscripts.softpedia.com/script/Multimedia/mediastreamer2-24442.html].

[15] oRTP [http://freshmeat.net/projects/ortp/].

[16] Wireshark [http://www.wireshark.org/].

[17] M. Handley, V. Jacobson, “SDP: Session Description Protocol,” IETF RFC 2327, April 1998.

[18] J. Rosenberg, H. Schulzrinne, “An Offer/Answer Model with Session Description Protocol (SDP),” IETF RFC 3264, June 2002.

[19] C. Perkins, “RTP and the Datagram Congestion Control Protocol (DCCP),” IETF Internet-Draft [draft-ietf-dccp-rtp-07.txt], Work in Progress, June 2007.

Appendix

Appendix A . Codes of oRTP

- LinphoneD\oRTP\src\rtpsession_inet.c

…

```
static ortp_socket_t create_and_bind(const char *addr, int port, int
*sock_family){
    int err;
    ortp_socket_t sock=-1;
#ifdef ORTP_INET6
    char num[8];
    struct addrinfo hints, *res0, *res;
#else
    struct sockaddr_in saddr;
#endif
/*Define DCCP parameters*/
    int pkt_size=256;
    int SOCK_DCCP=6;
    int SOL_DCCP=269;
    int DCCP_SOCKOPT_PACKET_SIZE=1;
    int DCCP_SOCKOPT_SERVICE=2;
    int IPPROTO_DCCP=33;
#ifdef ORTP_INET6
    memset(&hints, 0, sizeof(hints));
    hints.ai_family = PF_UNSPEC;
    snprintf(num, sizeof(num), "%d",port);
    err = getaddrinfo(addr,num, &hints, &res0);
    if (err!=0) {
        ortp_warning ("Error in getaddrinfo on (addr=%s port=%i): %s",
                        addr, port, gai_strerror(err));
        return -1;
```

```
    }
    for (res = res0; res; res = res->ai_next) {
        //Use the predefined DCCP parameters to create a socket
        sock = socket(res->ai_family, SOCK_DCCP, IPPROTO_DCCP);
        if (sock < 0)
          continue;
        // Set DCCP as the parameters of the socket options
        err = setsockopt(sock,SOL_DCCP,DCCP_SOCKOPT_PACKET_SIZE,
                (char*)&pkt_size,sizeof(pkt_size));
        if (err < 0)
        {
            ortp_warning ("Fail to set DCCP address reusable: %s.",
                          getSocketError());
        }
        err = setsockopt(sock,SOL_DCCP,DCCP_SOCKOPT_SERVICE,
                    (char*)&pkt_size,sizeof(pkt_size));
        if (err < 0)
        {
            ortp_warning ("Fail to set DCCP2 address reusable: %s.",
                          getSocketError());
        }
        *sock_family=res->ai_family;
        err = bind (sock, res->ai_addr, res->ai_addrlen);
        if (err != 0)
          {
            ortp_warning ("Fail to bind rtp socket to (addr=%s port=%i) :
%s.", addr,port, getSocketError());
            close_socket (sock);
             sock=-1;
            continue;
          }
        // Listen for socket connections
        err=listen(sock,10);
#ifndef __hpux
        switch (res->ai_family)
          {
            case AF_INET:
```

```
 if (IN_MULTICAST(ntohl(((struct sockaddr_in *)
                res->ai_addr)->sin_addr.s_addr)))
{
  printf("bind1-__hpux_AF_INET_in_multicast\n");
     struct ip_mreq mreq;
  mreq.imr_multiaddr.s_addr = ((struct sockaddr_in *)
                               res->ai_addr)->
                               sin_addr.s_addr;
  mreq.imr_interface.s_addr = INADDR_ANY;
  err = setsockopt(sock, IPPROTO_IP, IP_ADD_MEMBERSHIP,
                  (SOCKET_OPTION_VALUE) &mreq,
                   sizeof(mreq));
  if (err < 0)
    {
      ortp_warning ("Fail to join address group: %s.",
                     getSocketError());
      close_socket (sock);
         sock=-1;
      continue;
    }
}
 break;
case AF_INET6:
  printf("bind1-__hpux_AF_INET6\n");
 if (IN6_IS_ADDR_MULTICAST(&(((struct sockaddr_in6 *)
                            res->ai_addr)->sin6_addr)))
{
  struct ipv6_mreq mreq;
  mreq.ipv6mr_multiaddr = ((struct sockaddr_in6 *)
                            res->ai_addr)->sin6_addr;
  mreq.ipv6mr_interface = 0;
  err = setsockopt(sock, IPPROTO_IPV6, IPV6_JOIN_GROUP,
                  (SOCKET_OPTION_VALUE)&mreq,
                   sizeof(mreq));
  if (err < 0)
    {
      ortp_warning ("Fail to join address group: %s.",
```

```
                                getSocketError());
                    close_socket (sock);
                       sock=-1;
                    continue;
                   }
               }
                break;
            }
#endif
        break;
    }
    freeaddrinfo(res0);
#else
    printf("bind1--create_ipv4_socket\n");
    saddr.sin_family = AF_INET;
    *sock_family=AF_INET;
    err = inet_aton (addr, &saddr.sin_addr);
    if (err < 0)
    {
        ortp_warning ("Error in socket address:%s.", getSocketError());
        return err;
    }
    saddr.sin_port = htons (port);
    sock = socket (PF_INET, SOCK_DCCP, IPPROTO_DCCP);
    if (sock<0) return -1;
    err = setsockopt (sock, SOL_SOCKET, SO_REUSEADDR,
              (SOCKET_OPTION_VALUE)&optval, sizeof (optval));
    if (err < 0)
    {
        ortp_warning ("Fail to set rtp address reusable:
                       %s.",getSocketError());
    }
    err = bind (sock,
            (struct sockaddr *) &saddr,
            sizeof (saddr));
    if (err != 0)
    {
```

```
        ortp_warning ("Fail to bind rtp socket to port %i: %s.", port,
                      getSocketError());
        close_socket (sock);
        return -1;
    }
#endif
    if (sock>=0)
    {
#ifdef WIN32
        /* increase RTP buffer on windows */
        int bufsize = 32768;
        err = setsockopt(sock, SOL_SOCKET, SO_SNDBUF, (void *)&bufsize,
                         sizeof(bufsize));
        if (err == -1) {
            ortp_warning ("Fail to increase buffer size for socket
                          (port %i): %s.", port, getSocketError());
        }
        bufsize = 32768;
        err = setsockopt(sock, SOL_SOCKET, SO_RCVBUF, (void *)&bufsize,
                         sizeof(bufsize));
        if (err == -1) {
            ortp_warning ("Fail to increase buffer size for socket
                          (port %i): %s.", port, getSocketError());
        }
#endif      /* comment the following line to set the socket under the
blocking mode
        set_non_blocking_socket (sock);
        */
    }
    return sock;
}

…

//Wait for a new connection
int rtp_session_rtp_recv_accept (RtpSession * session)
{
```

```
    int connfd;
    ortp_socket_t sockfd=session->rtp.socket;
    struct sockaddr_in remaddr;
    socklen_t addrlen = sizeof (remaddr);
    if((connfd=accept(sockfd,(struct sockaddr *)
&remaddr,&addrlen))<0){
                        ortp_warning("accept_error");
       }
    else {
    printf("accept_ok\n");
    }
    return connfd;
}

…

//Receive RTP packets and add a parameter named connfd that returned from
function accept(…)to function rtp_session_rtp_recv(…)
int rtp_session_rtp_recv (RtpSession * session, uint32_t user_ts,int
connfd)
{
    int error ;
    ortp_socket_t sockfd=session->rtp.socket;
#ifdef ORTP_INET6
    struct sockaddr_in remaddr;
#else
    struct sockaddr remaddr;
#endif
    socklen_t addrlen = sizeof (remaddr);
    mblk_t *mp;
    if ((sockfd<0) && !rtp_session_using_transport(session, rtp))
        return -1;  /*session has no sockets for the moment*/
    while (1)
    {
        int bufsz;
        bool_t sock_connected=!!(session->flags &
                                 RTP_SOCKET_CONNECTED);
```

```
            if (session->rtp.cached_mp==NULL){
                session->rtp.cached_mp = allocb (session->recv_buf_size,
0);
            }
            mp=session->rtp.cached_mp;
            bufsz=(int) (mp->b_datap->db_lim - mp->b_datap->db_base);
        /*  if (sock_connected){
                error=recv(connfd,mp->b_wptr,bufsz,0);
            }else if (rtp_session_using_transport(session, rtp)){
                error = (session->rtp.tr->t_recvfrom)(session->rtp.tr,
                          mp->b_wptr, bufsz, 0,
                         (struct sockaddr *) &remaddr,&addrlen);
            }
            else{
                error = recvfrom(connfd, mp->b_wptr,bufsz, 0,
                                 (struct sockaddr *) &remaddr,&addrlen);
            }
        */
        //Receive RTP packets
        error = recv(connfd, mp->b_wptr,bufsz, 0);
        if(error>0) {
                session->totalpacketrecv++;
            //printf("recvpct=%d ",session->totalpacketrecv);
              }
        if (error > 0){
            if (session->symmetric_rtp && !sock_connected){
                // store the sender rtp address to do symmetric RTP
                memcpy(&session->rtp.rem_addr,&remaddr,addrlen);
                session->rtp.rem_addrlen=addrlen;
                /*
                if (session->use_connect){
                    if (try_connect(connfd,(struct sockaddr*)&remaddr,
                                    addrlen))
                    {
                */
                        session->flags|=RTP_SOCKET_CONNECTED;
                /*
```

```
            }
        }
        */
    }
    /* then parse the message and put on queue */
    mp->b_wptr+=error;
    rtp_session_rtp_parse (session, mp,
                           user_ts + session->rtp.hwrcv_diff_ts,
                           (struct sockaddr*)&remaddr,
                            addrlen);
    session->rtp.cached_mp=NULL;
    /*for bandwidth measurements:*/
    update_recv_bytes(session,error);
    return 0;
}
else
{
    int errnum=getSocketErrorCode();
    if (error == 0)
    {
        ortp_warning("rtp_recv: strange... recv() returned
                      zero.");
    }
    else if (!is_would_block_error(errnum))
    {
        if (session->on_network_error.count>0){
            rtp_signal_table_emit3(&session->on_network_error,
                                   (long)"Error receiving RTP
                                    packet",INT_TO_POINTER(get
                                    SocketErrorCode()));
            printf("network_error\n");
        }else{
             ortp_warning("Error receiving RTP packet:
                          %s.",getSocketError());
        }
    }
    /* don't free the cached_mp, it will be reused next time */
```

```
            return -1;  /* avoids an infinite loop ! */
        }
    }
    return error;
}
```

Appendix B . Codes of mediastreamer2

- LinphoneD\ mediastreamer2\src\ audiostream.c

…

```
RtpSession * create_duplex_rtpsession_recv( int locport, bool_t ipv6){
    RtpSession *rtpr;
    ortp_init();
    ortp_scheduler_init();
    rtpr=rtp_session_new(RTP_SESSION_SENDRECV);
    //Set the socket under the blocking mode
    rtp_session_set_scheduling_mode(rtpr,1);
    rtp_session_set_blocking_mode(rtpr,1);
    rtp_session_set_recv_buf_size(rtpr,MAX_RTP_SIZE);
    rtp_session_enable_adaptive_jitter_compensation(rtpr,TRUE);
    rtp_session_set_symmetric_rtp(rtpr,TRUE);
    rtp_session_set_local_addr(rtpr,ipv6 ? "::" : "0.0.0.0",locport);
    return rtpr;
}

RtpSession * create_duplex_rtpsession_send( int locport, bool_t ipv6){
    RtpSession *rtpr;
    ortp_init();
    ortp_scheduler_init();
    rtpr=rtp_session_new(RTP_SESSION_SENDRECV);
    rtp_session_set_recv_buf_size(rtpr,MAX_RTP_SIZE);
    //Set the sender under the connected mode
    rtp_session_set_connected_mode(rtpr,TRUE);
    rtp_session_enable_adaptive_jitter_compensation(rtpr,TRUE);
    rtp_session_set_symmetric_rtp(rtpr,TRUE);
    return rtpr;
}
```

…

```
int audio_stream_start_full_recv(AudioStream *stream, RtpProfile
*profile, const char *remip,int remport, int payload,int jitt_comp, const
char *infile, const char *outfile, MSSndCard *playcard, MSSndCard
*captcard, bool_t use_ec)
{
    RtpSession *rtps_recv=stream->session;
    PayloadType *pt;
    int tmp;
    rtp_session_set_payload_type(rtps_recv,payload);
    rtp_session_set_jitter_compensation(rtps_recv,jitt_comp);
    stream->rtprecv=ms_filter_new(MS_RTP_RECV_ID);
    ms_filter_call_method(stream->rtprecv,MS_RTP_RECV_SET_SESSION,
                          rtps_recv);
    stream->session=rtps_recv;
    if (playcard!=NULL)
stream->soundwrite=ms_snd_card_create_writer(playcard);
    else {
        stream->soundwrite=ms_filter_new(MS_FILE_REC_ID);
        if (outfile!=NULL) audio_stream_record(stream,outfile);
    }
    /* creates the couple of encoder/decoder */
    pt=rtp_profile_get_payload(profile,payload);
    if (pt==NULL){
        ms_error("audiostream.c: undefined payload type.");
        return -1;
    }
    stream->decoder=ms_filter_create_decoder(pt->mime_type);
    if ((stream->encoder==NULL) || (stream->decoder==NULL)){
        /* big problem: we have not a registered codec for this payload...*/
        ms_error("mediastream.c: No decoder available for payload
                  %i.",payload);
        return -1;
    }
    if (use_ec) {
        stream->ec=ms_filter_new(MS_SPEEX_EC_ID);
```

```
        ms_filter_call_method(stream->ec,MS_FILTER_SET_SAMPLE_RATE,
                              &pt->clock_rate);
        printf("use_ec---\n");
    }
    /* give the sound filters some properties */
    ms_filter_call_method(stream->soundwrite,
                          MS_FILTER_SET_SAMPLE_RATE,&pt->clock_rate);
    tmp=1;
    ms_filter_call_method(stream->soundwrite,MS_FILTER_SET_NCHANNELS,
                          &tmp);
    ms_filter_call_method(stream->decoder,MS_FILTER_SET_SAMPLE_RATE,
                          &pt->clock_rate);
    ms_filter_call_method(stream->decoder,MS_FILTER_SET_BITRATE,
                          &pt->normal_bitrate);
    if (pt->recv_fmtp!=NULL)
ms_filter_call_method(stream->decoder,MS_FILTER_SET_FMTP,
                      (void*)pt->recv_fmtp);
    /* and then connect all */
    /* tip: draw yourself the picture if you don't understand */
    if (stream->ec){
        ms_filter_link(stream->ec,0,stream->soundwrite,0);
    }else{
        ms_filter_link(stream->dtmfgen,0,stream->soundwrite,0);
    }
    ms_filter_link(stream->rtprecv,0,stream->decoder,0);
    ms_filter_link(stream->decoder,0,stream->dtmfgen,0);
    /* create ticker */
    stream->ticker=ms_ticker_new();
    ms_ticker_attach(stream->ticker,stream->rtprecv);
    return 0;
}

int audio_stream_start_full_send(AudioStream *stream, RtpProfile
*profile, const char *remip,int remport, int payload,int jitt_comp, const
char *infile, const char *outfile, MSSndCard *playcard, MSSndCard
*captcard, bool_t use_ec)
{
```

```
    //Declare the RTP session of sender
    RtpSession *rtps=stream->session2;
    PayloadType *pt;
    rtp_session_set_profile(rtps,profile);
    if (remport>0) {
        rtp_session_set_remote_addr(rtps,remip,remport);
    }
    rtp_session_set_payload_type(rtps,payload);
    rtp_session_set_jitter_compensation(rtps,jitt_comp);
    stream->rtpsend=ms_filter_new(MS_RTP_SEND_ID);
    if (remport>0)
        ms_filter_call_method(stream->rtpsend,MS_RTP_SEND_SET_SESSION,
                              rtps);
    //Assign values to RTP session of the sender
    stream->session2=rtps;
    stream->dtmfgen=ms_filter_new(MS_DTMF_GEN_ID);
    rtp_session_signal_connect(rtps,"telephone-event",
                               (RtpCallback)on_dtmf_received,
                               (unsigned long)stream->dtmfgen);
    rtp_session_signal_connect(rtps,"payload_type_changed",
                               (RtpCallback)payload_type_changed,
                               (unsigned long)stream);
    // creates the local part /
    if (captcard!=NULL)
stream->soundread=ms_snd_card_create_reader(captcard);
    else {
        stream->soundread=ms_filter_new(MS_FILE_PLAYER_ID);
        if (infile!=NULL) audio_stream_play(stream,infile);
    }
    // creates the couple of encoder/decoder /
    pt=rtp_profile_get_payload(profile,payload);
    if (pt==NULL){
        ms_error("audiostream.c: undefined payload type.");
        return -1;
    }
    stream->encoder=ms_filter_create_encoder(pt->mime_type);
    if (use_ec) {
```

```
        stream->ec=ms_filter_new(MS_SPEEX_EC_ID);
        ms_filter_call_method(stream->ec,MS_FILTER_SET_SAMPLE_RATE,
                              &pt->clock_rate);
    }
    // give the sound filters some properties
    ms_filter_call_method(stream->soundread,MS_FILTER_SET_SAMPLE_RATE
                          ,&pt->clock_rate);
    // give the encoder/decoder some parameters
   ms_filter_call_method(stream->encoder,MS_FILTER_SET_SAMPLE_RATE,
                          &pt->clock_rate);
    if (pt->normal_bitrate>0){
         ms_message("Setting audio encoder network bitrate to %i",
                    pt->normal_bitrate);
         ms_filter_call_method(stream->encoder,MS_FILTER_SET_BITRATE,
                              &pt->normal_bitrate);
    }
    if (pt->send_fmtp!=NULL)
    ms_filter_call_method(stream->encoder,MS_FILTER_SET_FMTP,
                           (void*)pt->send_fmtp);
    // and then connect all
    // tip: draw yourself the picture if you don't understand
    if (stream->ec){
        ms_filter_link(stream->soundread,0,stream->ec,1);
        ms_filter_link(stream->ec,1,stream->encoder,0);
        ms_filter_link(stream->dtmfgen,0,stream->ec,0);
    }else{
        ms_filter_link(stream->soundread,0,stream->encoder,0);
    }
    ms_filter_link(stream->encoder,0,stream->rtpsend,0);
    printf("encoder,0-rtpsend,0\n");
    // create ticker
    stream->ticker=ms_ticker_new();
    ms_ticker_attach(stream->ticker,stream->soundread);
    return 0;
}
```

- LinphoneD\ mediastreamer2\src\ msrtp.c

```
//Include ortp/ortp.h and ortp/rtpsession.h
#include "ortp/ortp.h"
#include "ortp/rtpsession.h"

…

static void sender_preprocess(MSFilter * f){
    SenderData *d = (SenderData *) f->data;
    //Connect with the Receiver
    if(connect(d->session->rtp.socket,
               (struct sockaddr*)&d->session->rtp.rem_addr,
               d->session->rtp.rem_addrlen)<0)
    {
        ortp_warning("Could not connect() socket: %s",getSocketError());
    }else{
        printf("sender-connect()--ok!!!\n");
    }
}

…

static void receiver_preprocess(MSFilter * f){
    ReceiverData *d = (ReceiverData *) f->data;
    if (d->session){
        PayloadType *pt=rtp_profile_get_payload(
            rtp_session_get_profile(d->session),
            rtp_session_get_recv_payload_type(d->session));
        if (pt){
            if (pt->type!=PAYLOAD_VIDEO){
                rtp_session_flush_sockets(d->session);
            }
        }
    }
    //Call rtp_session_rtp_recv_accept() to wait for the sender creating
a connection
    d->connfd=rtp_session_rtp_recv_accept(d->session);
}
```

…

```
static void receiver_process(MSFilter * f)
{
    ReceiverData *d = (ReceiverData *) f->data;
    mblk_t *m;
    uint32_t timestamp;
    if (d->session == NULL)
        return;
    timestamp = (f->ticker->time * d->rate) / ((uint64_t)1000);
    // using "if" statement to replace a "while" loop
    if((m = rtp_session_recvm_with_ts(d->session,
        timestamp,d->connfd)) != NULL) {
        mblk_t *payload = m->b_cont;
        mblk_set_timestamp_info(payload, rtp_get_timestamp(m));
        mblk_set_marker_info(payload, rtp_get_markbit(m));
        mblk_set_payload_type(payload, rtp_get_payload_type(m));
        freeb(m);
        ms_queue_put(f->outputs[0], payload);
    }
    /* check received STUN request */
    if (d->ortp_event!=NULL)
    {
        OrtpEvent *evt = ortp_ev_queue_get(d->ortp_event);
        while (evt != NULL) {
            if (ortp_event_get_type(evt) ==
                ORTP_EVENT_STUN_PACKET_RECEIVED) {
                ice_process_stun_message(d->session, d->cpair, evt);
            }
            if (ortp_event_get_type(evt) ==
                ORTP_EVENT_TELEPHONE_EVENT) {
            }
            ortp_event_destroy(evt);
            evt = ortp_ev_queue_get(d->ortp_event);
        }
    }
```

```
}
```

Appendix B . Codes of Wireshark

- wireshark-1.0.2\epan\dissectors\packet-sdp.c

…

```
static void dissect_sdp(tvbuff_t *tvb, packet_info *pinfo, proto_tree
*tree) {
```

…

```
    if(global_sdp_establish_conversation){
    /* Check if media protocol is RTP */
        is_rtp = ( (strcmp(transport_info.media_proto[n],"RTP/AVP")==0)
    || (strcmp(transport_info.media_proto[n],"DCCP/RTP/AVP")==0) );
          /* Check if media protocol is SRTP */
        is_srtp =
    (strcmp(transport_info.media_proto[n],"RTP/SAVP")==0);
          /* Check if media protocol is T38 */
        is_t38 = ( (strcmp(transport_info.media_proto[n],"UDPTL")==0) ||
        (strcmp(transport_info.media_proto[n],"udptl")==0) );
/* Check if media protocol is MSRP/TCP */
        is_msrp =
        (strcmp(transport_info.media_proto[n],"msrp/tcp")==0);
}
```

…

```
        if (is_srtp) {
        struct srtp_info *dummy_srtp_info = se_alloc0(sizeof(struct
    srtp_info));
        srtp_add_address(pinfo, &src_addr, port, 0, "SDP",
    pinfo->fd->num,
        transport_info.media[n].rtp_dyn_payload, dummy_srtp_info);
```

```
        } else {
        // Distinguish the received RTP packets which are transmitted over
    UDP or DCCP from SDP
        if(strcmp(transport_info.media_proto[n],"RTP/AVP")==0){
            pinfo->ptype=3;//UDP
        }else
        if(strcmp(transport_info.media_proto[n],"DCCP/RTP/AVP")==0){
            pinfo->ptype=4;//DCCP
        }
        rtp_add_address(pinfo, &src_addr, port, 0, "SDP",
                        pinfo->fd->num,
        transport_info.media[n].rtp_dyn_payload);
        }
```

- wireshark-1.0.2\epan\dissectors\packet-rtp.c

…

```
void srtp_add_address(packet_info *pinfo, address *addr, int port, int
                      other_port, const gchar *setup_method, guint32
                      setup_frame_number, GHashTable *rtp_dyn_payload,
                      struct srtp_info *srtp_info)
{
   …

   // Analyze the received RTP packets transmitted over UDP or DCCP
   if(pinfo->ptype==3){
   p_conv = find_conversation( setup_frame_number, addr, &null_addr,
                               PT_UDP, port, other_port, NO_ADDR_B |
                               (!other_port ? NO_PORT_B : 0));
   }else if(pinfo->ptype==4){
   p_conv = find_conversation( setup_frame_number, addr, &null_addr,
                               PT_DCCP, port, other_port, NO_ADDR_B |
                               (!other_port ? NO_PORT_B : 0));
```

```
    }
    /*If not, create a new conversation.*/
    if ( !p_conv || p_conv->setup_frame != setup_frame_number) {
          if(pinfo->ptype==3){
              p_conv = conversation_new( setup_frame_number, addr,
                                         &null_addr, PT_UDP, (guint32)port,
                                         (guint32)other_port, NO_ADDR2 |
                                         (!other_port ? NO_PORT2 : 0));
          }else if(pinfo->ptype==4){
              p_conv = conversation_new( setup_frame_number, addr,
                                        &null_addr, PT_DCCP, (guint32)port,
                                        (guint32)other_port, NO_ADDR2 |
                                        (!other_port ? NO_PORT2 : 0));
          }

      …

}
```