Step two in DCCP adoption: The Libraries

Leandro Melo de Sales, Hyggo Oliveira, Angelo Perkusich
Embedded Systems and Pervasive Computing Lab
{leandro,hyggo,perkusic}@embedded.ufcg.edu.br

Arnaldo Carvalho de Melo
Red Hat, Inc.
acme@redhat.com

Abstract

Multimedia applications are very popular in the Internet. The use of UDP in most of them may result in network collapse due to lack of congestion control. DCCP [4] is a new protocol to deliver multimedia in congestion controlled unreliable datagrams.

This paper presents discussions and results in enabling DCCP in open source libraries, as part of our efforts in disseminating the DCCP protocol to developers.

At OLS'08 we presented experimental results [9] using the DCCP implementation in the Linux kernel, where it was shown that DCCP behaves better than UDP in congested environments, while being fair with respect to TCP. This is a work in progress and nowadays DCCP is supported in libraries such as GNU CommonCPP, CCRTP, GNU uCommon, in the GStreamer framework and on Farsight 2.

1 Introduction

The Datagram Congestion Control Protocol (DCCP) is a message-oriented Transport Layer protocol that implements reliable connection setup, teardown, ECN, congestion control, and feature negotiation [11]. It was published as RFC 4340 [4] in March of 2006 by Internet Engineering Task Force (IETF) with the main propose of be an Internet protocol for transport multimedia content. In the Linux kernel, the first DCCP implementation was released in version 2.6.14.

The firstly versions of DCCP in the Linux kernel, considering the application developers point of view, was implemented to be used by a very small set of applications, simplest ones based on DCCP socket and not for that advanced multimedia applications. For instance, it was possible to use the socket API to implement a DCCP application to send characters between two hosts. The developers was able to use the common socket functions such as *connect*, *bind* and *accept* in a very similar TCP fashion. By the end of 2007, the DCCP implementation in the Linux kernel became stable and developers began to required DCCP in real development libraries and frameworks.

In the OLS'08 we published a paper and gave a talk discussing about experimental results on the performance of DCCP against UDP and TCP over a wireless network [2]. In that year, we presented that DCCP data flows are fair with respect to others TCP flows, while UDP was very aggressive in terms of network congestion, where in some situations both TCP and DCCP could not transmit any data. This occurs because TCP and DCCP implements congestion control, while UDP does not.

Considering the multimedia application developers requests for providing DCCP in the user space, we have concentrated our efforts on enabling it in a set of selected well-known open source multimedia frameworks. In this paper we present the experiences on enabling DCCP in these frameworks with two goals:

- enable DCCP in the user space to provide the developers an alternative for UDP;
- provide feedback to DCCP developers to improve the DCCP implementation in the Linux kernel.

It is a work in progress and for the first phase we have selected the following libraries: GNU CommonCPP, CCRTP, GNU uCommon, GStreamer framework and Farsight 2. By providing DCCP on these libraries, we aim at disseminating DCCP and making it useful in any Internet applications, while effectively make use of DCCP implementation provided in the kernel – it does not make sense provide DCCP in the Linux kernel and nobody use it.

This paper is organized as follow: in Section 2 are presented overview and background as a base for the rest of the paper. In Section 3 are provided an overview about DCCP and its main features. In Section 4, it is discussed our efforts on enabling DCCP in a set of open source libraries. The current and future works about enabling DCCP in these libraries are described in the Section 7. In Section 8, the conclusions are presented.

2 Overview and Background

The motivation for DCCP is based on the growth of Multimedia applications over the Internet in the last few years. The multimedia applications have receive special attention due to the popularization of high-speed residential Internet access and wireless connections, considering also new standards such as IEEE 802.16 (WiMax). This enables network applications that transmit and receive multimedia contents through the Internet to become feasible once developers and industry invest money and software development efforts in this area.

Industry and the open source community have developed specialized multimedia applications based on technologies such as Voice over IP (e.g., Skype, GoogleTalk, Gizmo), Internet Radio (e.g., SHOUTcast, Rhapsody), online games (e.g., Half Life, World of Warcraft), video conferencing. These applications offer sophisticated solutions that can approximate a face-to-face dialog for people, although they can be physically separated by hundreds or thousands of miles in distance.

These applications have different requirements when compared with application such as HTTP and E-Mail (connection oriented applications). The multimedia applications are delay sensitive, while they make a intensive use of the network bandwidth and tolerate occasional packet loss. Based on the behaviour of the multimedia data flows, this may lead to changes on the design principles of the multimedia application development.

Non-functional requirements such as end-to-end delay (latency) and the variation of the delay (jitter) must be taken into account, regardless the network topology considered [7]. Usually, multimedia applications use TCP and UDP as their transport protocol, but they may present many drawbacks regarding these non-functional requirements, and hence decrease the quality of the multimedia content transmitted.

The developers of multimedia applications usually choose to use the UDP protocol for transport the multimedia data. The massive choice for UDP by the multimedia application developers are explained by the fact that UDP introduces less delay in the data transmission in comparison with TCP, for example.

TCP is a connection oriented protocol that provides flow control, congestion control and retransmission of lost packets, which make the protocol appropriate for applications that require reliability during that transmission. Together, these TCP features increase the end-to-end delay during data transmission. Depends on the level of the delay, use TCP is not a best choice.

On the other hand, UDP is a very simple protocol, it does not provide any kind of congestion control, connection hand-shake and packet retransmission in case a packet is lost during the transmission. Together, these features – or the absence of them – in UDP can lead to high levels of network congestion. As a consequence, the network can collapse. In this circumstance, even those TCP (reliable) transmission can become impracticable.

In order to deal with those types of requirements, IETF standardized the Datagram Congestion Control Protocol (DCCP) [4], which appears as an alternative to transport congestion controlled flows of multimedia data, mainly for those applications focusing on the Internet. DCCP provides a way to gain access to congestion control mechanisms without having to implement them at the Application Layer. It allows for flow-based semantics like in TCP, but does not provide reliable in-order delivery.

3 A Bit About DCCP

DCCP was first introduced by Kohler [4] in July, 2001, at the IETF transport group. It provides specific features designed to fulfil the gap between TCP and UDP protocols for multimedia application requirements. It provides a connection-oriented transport layer for congestion-controlled, but unreliable data transmission. DCCP provides a framework that enables addition of new congestion control mechanism, which may be used and specified during the connection handshake, or even negotiated in already established connections. In addition, DCCP provides a mechanism to get connection statistics, which contain useful information about

packet loss, a congestion control mechanism with Explicit Congestion Notification (ECN) support, and Path Maximum Transmission Unit (PMTU) discovery [4].

From TCP, DCCP provides the connection-oriented and congestion-controlled features, and from UDP, DCCP provides an unreliable data transmission. The main reasons to specify a connection-oriented protocol is to facilitate the implementation of congestion control algorithms and enable firewall traversal. This is a UDP limitation that motivated network researchers to specify the STUN [8] (Simple Traversal of UDP through NATs (Network Address Translation)). STUN is a mechanism that helps UDP applications to work over firewalled networks. An important feature of DCCP is the modular congestion control framework. The congestion control framework was designed to allow extending the congestion control mechanism, as well as to load and unload new congestion control algorithms based on the application requirements. Each congestion control algorithm has an identifier called Congestion Control Identifier (CCID). Nowadays there are two standardized congestion control algorithms: CCID 2 [5] and CCID 3 [6].

DCCP is useful for applications with timing constraints on the delivery of data that may become useless to the receiver if reliable in-order delivery combined with congestion avoidance is used. Such applications include streaming media, multiparty online games and Internet telephony. Primary feature of these applications is that old messages quickly become stale, so that getting new messages is preferred than resending lost messages. Currently, such applications have often either settled for TCP or used UDP and implemented their own congestion control mechanisms.

While being useful for these applications, DCCP can also be positioned as a general congestion control mechanism for UDP-based applications, by adding, as needed, a mechanism for reliable and/or in-order delivery on the top of UDP/DCCP. In this context, DCCP allows the use of different – but generally TCP-friendly – congestion control mechanisms [10].

4 Libraries

Libraries is a collection of programming functions that can be used to develop a software. A software invokes these functions and, as a result, they provide to the software a return value or take an action. One of the main characteristic of a library is that it provides generic functions that can be shared by a set of software, and each of them combines the library functions with its functions to take actions. By allowing sharing of source code, the use of libraries avoid source code duplication.

In the context of what it is discussed in this paper, the Twinkle [20] soft-phone and Telepathy are two examples of software that adopted the concept explained before. Both projects are free software, one for Voice over IP and the other a library for developing videoconference applications. In the case of Twinkle, it provides many features for communicating: peer-to-peer, conference calls, call redirection, voice mail and instant messaging, all provided by the SIP protocol. The last Twinkle version available provides support for both TCP and UDP, while using Real-Time Protocol (RTP) [3] for signaling audio and video contents. The Figure 1 illustrates two examples of the library sharing. On the top of the stack the Twinkle uses CCRTP, that uses CommonCPP2 and Telepathy, that uses Farsight2 and GStreamer. Both uses the common file socket.h, the standard socket library provided by the operating system. It is the main socket header, where it is found the prototype for the well-known socket functions such as *connect*, *bind*, *send*, *recv* and *accept*.

Telepathy is a framework that can be used to develop communication software, such as VoIP, instant messaging, chat or videoconferencing. It is an open source software, applications use it as a library to simplify the process of developing multimedia applications. Empathy [14], Ekiga [13] and Tapioca [19] are examples of applications that use Telepathy on some of its multimedia service.

The Figure 1 shows a hierarchy of libraries that are used by applications Twinkle and Telepathy. The stack is divided in groups comprising the libraries according to the functionality. Despite the Twinkle and Telepathy are different applications in purpose and use of different libraries, they are at the top of the stack, indicating that they are classified at the highest level for transmitting media streams. The second level of the stack presents the libraries responsible for effectively process the application data, passed though Twinkle or Telepathy, and wrapper them into specific packets based on the protocol used to transport the data.

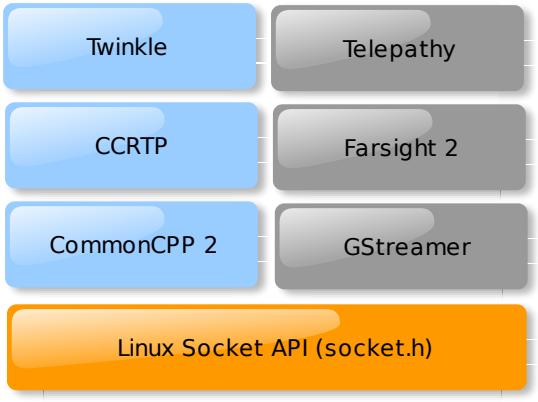

Figure 1: Library usage/sharing between Twinkle and Telepathy

5 Enabling DCCP on Libraries

After evaluating the performance of DCCP and presented the results in the OLS' 08, we have started the use of DCCP in the multimedia applications, where Twinkle and Telepathy being considered our starting point. Both of them have a set of libraries dependency and also we also had to change somehow these dependency libraries to accommodate DCCP.

We decided to start the changes in order to enabling DCCP in the selected applications considering the bottom-up approach. In this way, considering the pyramid illustrated in Figure 1, we started from the closest library from the pyramid base to the top of the pyramid. After finishing the work in a specific library, we started to implement DCCP support in the library immediately above and the process was repeated until there is no Twinkle or Telepathy dependencies without DCCP available.

For each libraries modified to make it support DCCP, we have implemented a corresponding example application to test and exploit the features of data transmission using DCCP between pairs. In addition to guide our implementation of DCCP for a given library, this application can be used as a documentation for enabling developer to understand the concepts of the library being used. The example developed in each step was a implementation of a ''hello world" application, where the sender application sends the ''hello world" message and the receiver application receives it.

During the process of changing a certain library, the example application to test the progress of the implementation was continuously executed. This characterized a kind of test-driven development. Once the necessary changes to the libraries were applied, we run the example application and use wireshark [21] to investigate the DCCP traffic transmitted in the network. The Figure 2 shows an example of the DCCP traffic while using DCCP with CommonCPP2. By verifying this, it was possible to certify the the application, through the library that we have provided DCCP support, was indeed transmitting DCCP flows.

5.1 Sockets Libraries—First Layer of the Stack

The libraries GStreamer and Commoncpp2, that are in the first layer considering the base stack shown in Figure 1, invokes operating system socket functions. They offer basic functionalities for transmitting data through the connected sockets between a client and a server. In this case, both libraries had to be changed in order to support DCCP, once these libraries only supported TCP and UDP sockets.

The strategy adopted was to add a structure for the DCCP client and server, so that define a connection-oriented sockets. Since GStreamer and CommonCPP uses TCP sockets, we started by coping the TCP implementation, since DCCP and TCP shares the same concept of connection-oriented sockets. Based on TCP implementation, we adapt the code to DCCP parameters passed to the socket functions, such as the *socket* function provided by the operating system. In the next section, we show by using some parts of the code added how we implemented DCCP support in GStreamer and in CommonCPP. Between lines 1-7 of Listing 1, it is shown a set of definitions used to provide DCCP support in the CommonCPP and GStreamer. Between lines 4-7 of Listing 1, it is the constants to read or write DCCP parameters defined by the DCCP implementation in the Linux Kernel. This values are read or written through the functions *getsockopt* and *setsockopt*. For example, the constant DCCP_SOCKOPT_AVAILABLE_CCIDS is passed to *getsockopt* to get the list of CCIDs available in the Linux Kernel. The constant DCCP_SOCKOPT_TX_CCID can be passed either to *getsockopt* or to *setsockopt* to get or set the current CCID, respectively.

In the line 9, it is illustrated how to create a new DCCP socket. Note, IPPROTO_DCCP assumes value 33 because it is the id defined by IANA [18] to DCCP. This

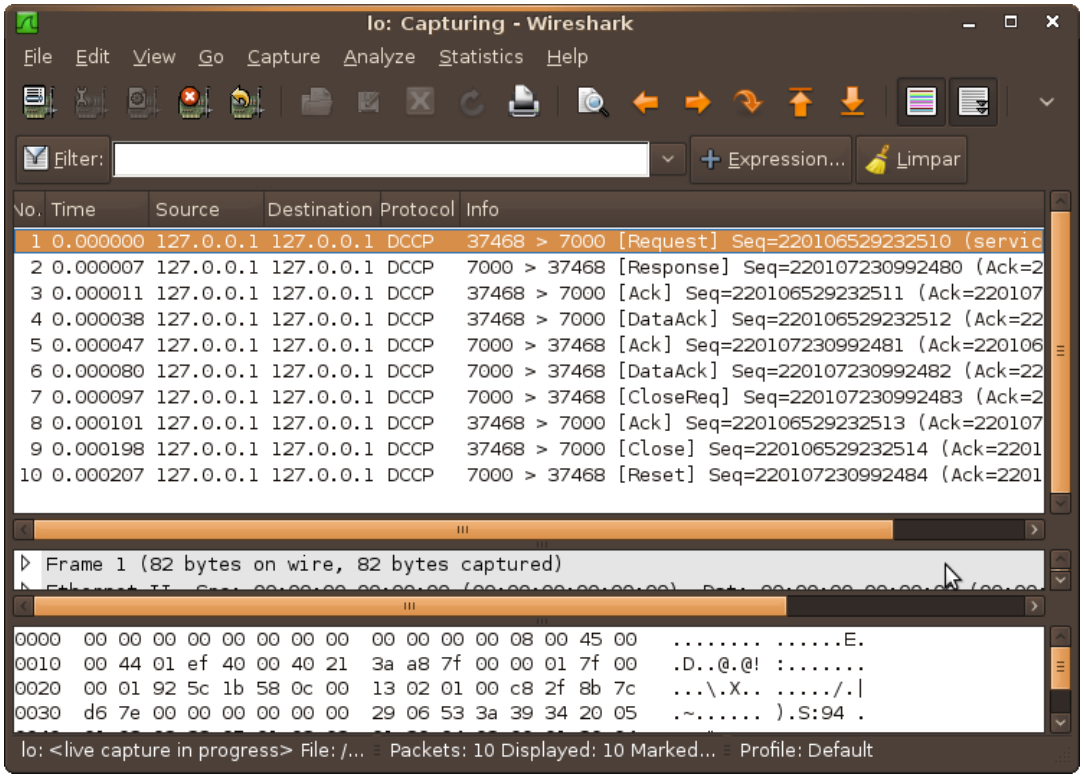

Figure 2: Wireshark filtering DCCP traffic and outputting DCCP packets details.

value is used in the IP packet header to specify which protocol is being used in the transport layer. The common values for this field are 1, 6 and 17 for ICMP, TCP and UDP, respectively. For a complete list of protocol identifier consult reference [17].

```
#define SOCK_DCCP                    6
#define IPPROTO_DCCP                 33
#define SOL_DCCP                     269
#define DCCP_SOCKOPT_AVAILABLE_CCIDS 12
#define DCCP_SOCKOPT_CCID            13
#define DCCP_SOCKOPT_TX_CCID         14
#define DCCP_SOCKOPT_RX_CCID         15

socket(AF_INET, SOCK_DCCP, IPPROTO_DCCP)
```

Listing 1: Definition for DCCP

Before discuss each library that was modified to support DCCP, consider a basic example of DCCP socket shown in the Listing 2. The example was implemented in Python programming language.

```
import socket

socket.SOCK_DCCP = 6
socket.IPPROTO_DCCP = 33
address = (socket.gethostname(),12345)
server = socket.socket(socket.AF_INET,
         socket.SOCK_DCCP,
         socket.IPPROTO_DCCP)
server.bind(address)
server.listen(1)
s,a = server.accept()
print s.recv(1024)
```

Listing 2: DCCP Server Socket in Python

```
import socket

socket.SOCK_DCCP = 6
socket.IPPROTO_DCCP = 33
address = (socket.gethostname(),12345)
server = socket.socket(socket.AF_INET,
         socket.SOCK_DCCP,
         socket.IPPROTO_DCCP)
server.bind(address)
server.listen(1)
s,a = server.accept()
print s.recv(1024)
```

Listing 3: DCCP Client Socket in Python

Listing 3 shows the corresponding DCCP client in Python. As it is possible to verify in both client and server examples written in Python, the DCCP socket programming is very simple as TCP socket programming. Basically the unique difference is *socket* func-

tion parameters, where it is necessary to specify IPPROTO=33 (DCCP), as explained before.

The example illustrated in Listing 2 implements a DCCP server that accept a DCCP client connection on port 12345. After connecting, the DCCP server reads 1024 bytes from the DCCP client and exit.

5.1.1 GNU CommonCPP 2

In order to provide DCCP support in CommonCPP, we started by implementing a TCP application to understand CommonCPP API. After making the test application and understand how the CommonCPP works, we investigated the code of the library and located the source codes responsible of handling the sockets by invoking the kernel socket functions. Once located, the TCP implementation code was copied, basically a class named TCPSocket, and modified to create the DCCPSocket class. Listing 4 shows fragments of the DCCPSocket class implemented in the file *src/socket.cpp* of CommonCPP 2 library. The complete code can be found in the CommonCPP repository referred in [15].

After implementing DCCP support for CommonCPP, we have modified the TCP application to make it a DCCP application. Next, we ran the test application and by using wireshark we have validated the implementation by filtering DCCP data packets sent by the test application using the DCCPSocket class.

```
\\ Socket class implementation
   (...)
\\ TCPSocket class implementation
   (...)
DCCPSocket::DCCPSocket(const IPV4Address
           &ia,
           tpport_t port,
           unsigned backlog) :
Socket(AF_INET, SOCK_DCCP, IPPROTO_DCCP) {
  struct sockaddr_in addr;

  memset(&addr, 0, sizeof(addr));
  addr.sin_family = AF_INET;
  addr.sin_addr = getaddress(ia);
  addr.sin_port = htons(port);
  family = IPV4;
  (...)
bool DCCPSocket::setCCID(int ccid) {
  (...)
  return (setsockopt(so, SOL_DCCP,
          DCCP_SOCKOPT_CCID,
          (char *)&ccid,
          sizeof (ccid)) >= 0);
}

int DCCPSocket::getTxCCID() {
  int ccid, ret;
  socklen_t ccidlen;

  ccidlen = sizeof(ccid);
  ret = getsockopt(so, SOL_DCCP,
        DCCP_SOCKOPT_TX_CCID,
        (char *)&ccid,
        &ccidlen);
  if (ret < 0) return -1;
  return ccid;
}

int DCCPSocket::getRxCCID() {
  int ccid, ret;
  socklen_t ccidlen;

  ccidlen = sizeof(ccid);
  ret = getsockopt(so, SOL_DCCP,
        DCCP_SOCKOPT_RX_CCID,
        (char *)&ccid,
        &ccidlen);
  if (ret < 0) return -1;
  return ccid;
}
(...)
```

Listing 4: Fragments of DCCPSocket class implemented in CommonCPP 2 (src/socket.cpp)

5.1.2 Gstreamer

GStreamer [16] is an open source multimedia framework that allows the programmer to write many types of streaming multimedia applications. Many well-know applications use GStreamer, such as Kaffeine, Amarok, Phonon, Rhythmbox, and Totem. The GStreamer framework facilitates the process of writing multimedia applications, ranging from audio and video playback to streaming multimedia content.

The work initiated by studying the mechanism of data transmission implemented in GStreamer and its concept of plugin-based framework. GStreamer is a plugin-based framework, where each plugin contains elements. Each of these elements provides a specific function – such as encoding, displaying, or rendering data – as well as the ability to read from or write to files. By combining and linking those elements, the programmer can build a pipeline for performing more complex functions. For example, it is possible to create a pipeline for reading

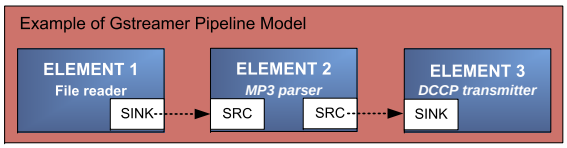

Figure 3: GStreamer Pipeline with three elements: a file reader, an MP3 encoder, and a DCCP transmitter.

from an MP3 file, decoding its contents, and playing the MP3.

Figure 3 represents a GStreamer pipeline composed by three elements. Data flows from Element 1 to Element 2 and finally to Element 3. Element 1 is the source element, which is responsible for providing data to the pipeline, whereas Element 3 is responsible for consuming data from the pipeline. Between the source element and the sink element, the pipeline is permitted to use other elements, such as Element 2 (shown in Figure 3). These intermediary elements are responsible for processing and modifying the content as the data passes along the pipeline.

Based on similar methodology adopted while implementing DCCP support for CommonCPP, we developed the DCCP plugin [9] for GStreamer to deal with data transmission using the DCCP protocol. This plugin has four elements: *dccpserversrc*, *dccpserversink*, *dccpclientsrc*, and *dccpclientsink*. The source elements (*dccpserversrc* and *dccpclientsrc*) are responsible for reading data from a DCCP socket and pushing it into the pipeline, and the sink elements (*dccpserversink* and *dccpclientsink*) are responsible for receiving data from the pipeline and writing it to a DCCP socket.

The *dccpserversrc* and the *dccpserversink* elements behave as the server, but only *dccpserversink* can transmit and only *dccpserversrc* can receive data. When the server element is initialized, it stays in a wait mode, which means the plugin is able to accept a new connection from a client element. The *dccpclientsink* element can connect to *dccpserversrc*, and *dccpclientsrc* can connect to *dccpserversink*.

If a developer wants to send data from the server to the client, you need to use *dccpclientsrc* and *dccpserversink* elements. To send data from the client to server, you need to use the *dccpclientsink* and *dccpserversrc* elements. GStreamer's *gst-launch* command supports the creation of pipelines, and it is also used to debug plugins.

```
gst-launch [!<element> <element params>]+
```

Listing 5: GStreamer gst-launch syntax

Listing 5 illustrates the basic syntax for *gst-launch*. The *gst-launch* command get a list of GStreamer elements with its parameters separated by a exclamation character. Note the ! character, it links the plugin elements, which is similar to the pipe character ("|") very used in the Linux shell prompt. This means that the output of an element is the input to the next specified plugin element.

As an example of the gst-launch command, consider two pipelines to transmit an MP3 stream over the network with DCCP: One works as a DCCP server that streams an MP3 audio file, and the second pipeline is associated with a DCCP client that connects to the remote DCCP server and reproduces the audio content transmitted by the server. To make the example work, you must install GStreamer. In this case, you need the GStreamer-Core, Gst-Base-Plugins, and Gst-Ugly-Plugins packages. Do not worry about the GStreamer installation; GStreamer is a widely used framework available in many Linux package systems for a variety of distributions, such as Debian, Gentoo, Mandriva, Red Hat, and Ubuntu. Once you perform the GStreamer installation, the last step is to compile and install the DCCP Plugin for GStreamer. The Listing 6 shows the command that you can run to install DCCP Plugin for GStreamer, after download it from [12].

```
./autogen --prefix=/usr
make
make install
```

Listing 6: Installing DCCP Plugin for GStreamer

Listing 7 shows a *gst-launch* example that runs a server accepting DCCP connections. Once a client connects, the server starts to stream the audio file named *yourmusic.mp3*. Note that you can specify the CCID with the ccid parameter. This pipeline initializes the server in DCCP port 9011. The server will be waiting for a client to connect to it. When the connection occurs, the server starts to transmit the MP3 stream using CCID-2. The *mp3parse* element is responsible for transmitting a stream. To see more information about mp3parse and the other parameters that are available, run *gst-inspect dccpserversink*.

```
gst-launch filesrc \
  location=yourmusic.mp3 ! \
```

```
  mp3parse ! dccpserversink port=9011 \
  ccid=2
```

Listing 7: Gst-Launch example starting a DCCP server to stream an mp3 file

Next, start the corresponding client as shown in Listing 8. This GStreamer pipeline initializes the client and connects to the host localhost in port 9011. Once connected, the client starts to receive the MP3 stream, decodes the stream using the *decodebin* element, and pipes the stream to the *alsasink* element, which reproduces the multimedia content in the default audio output device.

```
gst-launch -v dccpclientsrc host=localhost
   port=9011 ccid=2 ! decodebin ! alsasink
```

Listing 8: Gst-Launch example starting a DCCP client to receive an mp3 stream

After implementing the DCCP GStreamer plugin by using socket programming in a similar way done for CommonCPP and validate it using *gst-launch* and *wireshark*, we developed a set of example applications, where client and server applications can stream multimedia content reading from several data sources. For instance, we implemented an application that capture audio from the microphone or from a mp3 file and stream the content to a remote host using DCCP sockets.

The next example shows how to use the GStreamer API to embed DCCP plugin into applications. The application will do the same example explained using *gst-launch*, but this time through the C programming language and GObject, a programming library available for GStreamer application and plugin development. The application creates the same pipeline of the previous examples.

Start by initializing the GStreamer settings, as shown in Listing 9. Note that Listing 9 also defines *GstElements filesrc, mp3parse, and dccpserversink*.

```
#include <string.h>
#include <math.h>
#include <gst/gst.h>

int main(int argc, char **argv) {
  GMainLoop *loop;
  GstElement *pipeline, *filesrc;
  GstElement *mp3parse, *dccpserversink;
  GstBus *bus;

  gst_init(&argc, &argv);
  loop = g_main_loop_new (NULL, FALSE);

  if (argc != 3) {
    g_print("Usage: %s port mp3_location",
            argv[0]);
     return -1;
  }
  return 0;
}
```

Listing 9: Initializing GStreamer Pipeline

The next step is to instantiate a bus callback function to listen to GStreamer pipeline events. A bus is a system that takes care of forwarding messages from the pipeline to the application. The idea is to set up a message handler on the bus that leads the application to control the pipeline when necessary. Put the function shown in Listing 10 above the main function defined in Listing 9.

```
static gboolean bus_event_callback (
  GstBus *bus, GstMessage *msg,
  gpointer data) {

  GMainLoop *loop = (GMainLoop *) data;
  switch (GST_MESSAGE_TYPE (msg)) {
    case GST_MESSAGE_EOS:
       g_print("End-of-stream\n");
       g_main_loop_quit(loop);
       break;
    case GST_MESSAGE_ERROR:
       gchar *debug;
       GError *err;
       gst_message_parse_error(msg, &err,
                               &debug);
       g_free(debug);
       g_print("Error: %s\n",
               err->message);
       g_error_free(err);
       g_main_loop_quit(loop);
       break;
    default:
       break;
  }
  return TRUE;
}
```

Listing 10: Defining GStreamer Bus Event Callback

Every time an event occurs in the pipeline, GStreamer calls the gboolean bus_call function. For example, if you implement a GUI interface for your application, you can show a message announcing the end of the stream or deactivate the GUI stop button when the type of the GStreamer bus message is GST_MESSAGE_EOS. Now comes the most important part of this example—defining the elements and building the GStreamer

pipeline. Insert the code shown in Listing 11 into the main function, after checking the parameter count.

```
pipeline = gst_pipeline_new
           ("dccp-audio-sender");
filesrc = gst_element_factory_make
           ("filesrc", "file-source");
mp3parse = gst_element_factory_make
           ("mp3parse", "mp3parse");
dccpserversink = gst_element_factory_make
           ("dccpserversink",
            "server-sink");
```

Listing 11: Defining GStreamer Elements

Listing 11 first instantiates a new pipeline, dccp-audio-sender, which can be used for future references in the code. Then the code instantiates the *filesrc* element with the name file-source. This element will be used to read the specified MP3 file as an argument of the application. Use the same process to instantiate the elements *mp3parse* and *dccpserversink*. Once all the necessary elements are instantiated, certify that all are properly loaded. For this case, proceed as shown in Listing 12.

```
if (!pipeline || !filesrc ||
    !mp3parse || !dccpserversink) {
  g_print("Element(s) not instantiated");
  return -1;
}
```

Listing 12: Checking GStreamer Elements

The next step is to set the respective element parameters, as shown in Listing 13. For this application, we need to set two parameters: the *port*, where the server will listen and accept client connection from, and the audio file path represented by the parameter *location*.

```
g_object_set (G_OBJECT (dccpserversink),
         "port", atoi(argv[1]), NULL);
g_object_set (G_OBJECT (filesrc),
         "location", argv[2], NULL);
```

Listing 13: Setting Elements Parameters

Once all the elements are instantiated and the parameters are defined, it is time to attach the bus callback defined in Listing 10 to the bus of the pipeline. Also, it is need to add the elements to the pipeline and link them, as shown in Listing 14.

```
bus = gst_pipeline_get_bus
       (GST_PIPELINE(pipeline));
gst_bus_add_watch (bus,
       bus_event_callback, loop);
gst_object_unref(bus);
gst_bin_add_many(GST_BIN (pipeline),
       filesrc, mp3parse, dccpserversink,
       NULL);
gst_element_link_many(filesrc, mp3parse,
       dccpserversink, NULL);
```

Listing 14: Linking GStreamer Elements (Server)

Listing 15 shows how to execute the pipeline. Note that GStreamer runs in a main loop (line 5). This means that when this main loop finishes—for example, when the user types Ctrl+C—it is necessary to do some clean up (lines 6 and 10).

```
1  g_print("Setting to PLAYING\n");
2  gst_element_set_state
3     (pipeline, GST_STATE_PLAYING);
4  g_print("Running\n");
5  g_main_loop_run(loop);
6  g_print("Returned, stopping playback\n");
7  gst_element_set_state
8     (pipeline, GST_STATE_NULL);
9  g_print("Deleting pipeline\n");
10 gst_object_unref(GST_OBJECT (pipeline));
```

Listing 15: Executing the GStreamer Pipeline (Server)

The easiest part is to compile the server application—just run the command, which will link the GStreamer libs with the example application, that is in Listing 16. To run the DCCP GStreamer Server execute the command in the line 4 of the Listing 16.

```
1 $ gcc -Wall $(pkg-config --cflags \
2     --libs gstreamer-0.10) \
3     -o gst_dccp_server gst_dccp_server.c
4 $ ./gst_dccp_server 9011 yourmusic.mp3
5
6 $ gcc -Wall $(pkg-config --cflags
7    --libs gstreamer-0.10)
8    gst_dccp_client.c -o gst_dccp_client
9 $ ./gst_dccp_client localhost 9011
```

Listing 16: Compile and run server and client examples

Note that the example uses port 9011, which the server will use to open the DCCP socket and transmit the stream through the network to the remote DCCP client. Now it is time to build a corresponding client application that acts just like the gst-launch client command discussed previously. The DCCP client application is similar to the server application (Listing 17). Basically, you must initialize GStreamer, check command-line parameters, instantiate the necessary elements, and link them to build the GStreamer pipeline. Finally, to compile and

run the client application, execute the commands of the line 6 and 9 of the Listing 16.

```
1 #include <string.h>
2 #include <math.h>
3 #include <gst/gst.h>
4
5 static gboolean bus_event_callback
6   (GstBus *bus, GstMessage *msg,
7   gpointer data) {
8   GMainLoop *loop = (GMainLoop *) data;
9   switch (GST_MESSAGE_TYPE(msg)) {
10    case GST_MESSAGE_EOS:
11      g_print("End-of-stream\n");
12      g_main_loop_quit(loop);
13      break;
14    case GST_MESSAGE_ERROR:
15      gchar *debug;
16      GError *err;
17      gst_message_parse_error(msg, &err,
18        &debug);
19      g_free(debug);
20      g_print("Error: %s\n",
21        err->message);
22      g_error_free(err);
23      g_main_loop_quit(loop);
24      break;
25    default:
26      break;
27  }
28  return TRUE;
29 }
30
31 int main(int argc, char *argv) {
32   GMainLoop *loop;
33   GstElement *pipeline, *dccpclientsrc;
34   GstElement *decodebin, *alsasink;
35   GstBus *bus;
36
37   gst_init(&argc, &argv);
38   loop = g_main_loop_new(NULL, FALSE);
39   if (argc != 3) {
40     g_print("Usage: %s host Port\n",
41             argv[0]);
42     return -1;
43   }
44
45   pipeline = gst_pipeline_new(
46             "audio-sender");
47   dccpclientsrc = gst_element_factory_make
48            ("dccpclientsrc",
49             "client-source");
50   decodebin = gst_element_factory_make
51               ("decodebin", "decodebin");
52   alsasink = gst_element_factory_make
53               ("alsasink", "alsa-sink");
54   if (!pipeline  || !alsasink ||
55       !decodebin || !dccpclientsrc) {
56     g_print(
57      "Element(s) not instantiated\n");
58     return -1;
59   }
60
61   g_object_set(G_OBJECT(dccpclientsrc),
62             "host", argv[1], NULL);
63   g_object_set(G_OBJECT(dccpclientsrc),
64             "port", atoi(argv[2]), NULL);
65   gst_bin_add_many(GST_BIN(pipeline),
66             dccpclientsrc, decodebin,
67             alsasink, NULL);
68   gst_element_link_many(dccpclientsrc,
69             decodebin, alsasink, NULL);
70   bus = gst_pipeline_get_bus
71             (GST_PIPELINE(pipeline));
72   gst_bus_add_watch(bus,
73             bus_event_callback, loop);
74   gst_object_unref(bus);
75
76   g_print("Setting to PLAYING\n");
77   gst_element_set_state(pipeline,
78               GST_STATE_PLAYING);
79   g_print("Running\n");
80   g_main_loop_run(loop);
81   g_print(
82     "Returned, stopping playback\n");
83   gst_element_set_state(pipeline,
84               GST_STATE_NULL);
85   g_print("Deleting pipeline\n");
86   gst_object_unref(GST_OBJECT(pipeline));
87   return 0;
88 }
```

Listing 17: DCCP Client source code

6 Contributions with the Open Source

In addition to the implementation of DCCP support in the libraries mentioned before, we have also provided some additional contributions while developing the support for DCCP in that libraries. For example, during the development of the DCCP GStreamer plugin, we have noticed that DCCP implementation in the Linux kernel did not provide a mechanism for reading how much bytes is available in the receiving buffer in a given moment of the DCCP connection. We have reported this missing to the DCCP developers, while we contribute with them by implementing and testing this feature in the Linux Kernel. The summary of the patch and the patch itself is available from [1].

We have also contributed to the GStreamer and CommonCPP projects by providing DCCP support patches. Nowadays, both projects officially support DCCP protocol.

7 Current and Future works

Nowadays we are in a working in progress to provide DCCP support in Farsigh2 and CCRTP. We are developing a testing application for video conferencing between hosts. For both APIs we have started the process of adding the DCCP support for connection-oriented services. We are in constant contact with the Farsight2 and CCRTP developers. They are helping us to implement DCCP support on them.

For future works, we will provide DCCP support in MPlayer and finalize DCCP support in the Twinkle softphone.

8 Conclusion

We have presented the basic concepts of DCCP, the process of enabling DCCP in CommonCPP and in the GStreamer and how to build a DCCP-based application using the GStreamer DCCP plugin. The way how DCCP was implemented in the Linux kernel allowed us to rapidly implement DCCP support in the user-space API, like GStreamer and CommonCPP. The contributions that we have provided in this work will enable new DCCP applications, enabling alternatives for UDP protocol based applications.

Network analysis and testing applications, such as TTCP, tcpdump, and Wireshark already provide support for the DCCP protocol, and multimedia tools such as the open source VLC player accommodate DCCP streaming. As multimedia developers become aware of its benefits, it can expect to hear more about DCCP in the coming years.

References

[1] Arnaldo Carvalho de Melo, Leandro Melo de Sales, Ian McDonald, and David S. Miller. Implement siocinq/fionread. `http://git.kernel.org/?p=linux/kernel/git/davem/net-2.6.git;a=commitdif%f;h=6273172e1772bf5ce8697bcae145f0f2954fd159`. Last access on July 2009.

[2] Leandro Melo de Sales, Hyggo Oliveira de Almeida, Angelo Perkusich, and Arnaldo Carvalho de Melo. Measuring DCCP for Linux Against TCP and UDP With Wireless Mobile Devices. In *Ottawa Linux Symposium 2008*, volume 1, pages 163–177, 7 2008.

[3] J. Du, D. Putzolu, L. Cline, D. Newell, M. Clark, and D. Ryan. An Extensible Framework for RTP-based Multimedia Applications. In *Proceedings do 7th International Workshop on Network and Operating System Support for Digital Audio and Video*, volume 1, pages 53–60, 1997.

[4] Eddie Kohler, Mark Handley, and Sally Floyd. Datagram Congestion Control Protocol (DCCP), 3 2006. http://www.ietf.org/rfc/rfc4340.txt. Last access on July 2009.

[5] Eddie Kohler, Mark Handley, and Sally Floyd. Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 2: TCP-like Congestion Control, 3 2006. http://www.ietf.org/rfc/rfc4341.txt. Last access on July 2009.

[6] Eddie Kohler, Mark Handley, and Sally Floyd. Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 3: TCP-Friendly Rate Control (TFRC), 3 2006. http://www.ietf.org/rfc/rfc4342.txt. Last access on July 2009.

[7] James F. Kurose and Keith W. Ross. *Computer Networks and the Internet: A New Approach*. Addison Wesley, 2 edition, 9 2005.

[8] J. Rosenberg, J. Weinberger, C. Huitema, and R. Mahy. STUN - Simple Traversal of User Datagram Protocol (UDP) through Network Address Translators (NATs), 3 2003. http://www.ietf.org/rfc/rfc3489.txt. Last access on July 2009.

[9] Leandro Sales, Hyggo Almeida, and Angelo Perkusich. The DCCP Protocol in Three Steps. *Linux Magazine*, (92):56–62, 12 2008.

[10] Leandro M. Sales, Hyggo O. Almeida, Angelo Perkusich, and Marcello Sales Jr. An Experimental Evaluation of DCCP Transport Protocol: A Focus on the Fairness and Hand-off over 802.11g Networks. In *Consumer Communications and Networking Conference Proceedings*, pages 1149–1153, 1 2008.

[11] Leandro M. Sales, Hyggo O. Almeida, Angelo Perkusich, and Marcello Sales Jr. On the Performance of TCP, UDP and DCCP over 802.11g Networks. In *In Proceedings of the SAC 2008 23rd ACM Symposium on Applied Computing Fortaleza, CE*, pages 2074–2080, 1 2008.

[12] E-Phone Team. Dccp plugin for gstreamer. `https://garage.maemo.org/projects/ephone`. Last access on July 2009.

[13] Ekiga Team. Ekiga - open source voip and video conferencing application. `http://ekiga.org/`. Last access on July 2009.

[14] Empathy Team. Empathy - instant-messaging. `http://live.gnome.org/Empathy`. Last access on June, 2009.

[15] GNU CommonCPP Team. Commoncpp source code repository. `http://savannah.gnu.org/projects/commoncpp`. Last access on July 2009.

[16] GStreamer Team. Gstreamer - library for constructing graphs of media-handling components. `http://www.gstreamer.net/`. Last access on July 2009.

[17] IANA Team. Iana - assigned internet protocol numbers. `http://www.iana.org/assignments/protocol-numbers/`. Last access on July 2009.

[18] IANA Team. Iana - internet assigned numbers authority. `http://www.iana.org/`. Last access on July 2009.

[19] Tapioca Team. Tapioca - provides a set of convenience libraries to easily integrate voip and im. `http://tapioca-voip.sourceforge.net/wiki/index.php/Tapioca`. Last access on July 2009.

[20] Twinkle Team. Twinkle - softphone for your voice over ip and instant messaging communcations using the sip protocol. `http://www.twinklephone.com/`. Last access on July 2009.

[21] Wireshark Team. Wireshark - the worldś foremost network protocol analyzer. `http://www.wireshark.org/`. Last access on July 2009.

Proceedings of the Linux Symposium

July 13th–17th, 2009
Montreal, Quebec
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*

Programme Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
James Bottomley, *Novell*
Bdale Garbee, *HP*
Dave Jones, *Red Hat*
Dirk Hohndel, *Intel*
Gerrit Huizenga, *IBM*
Alasdair Kergon, *Red Hat*
Matthew Wilson, *rPath*

Proceedings Committee

Robyn Bergeron
Chris Dukes, *workfrog.com*
Jonas Fonseca
John 'Warthog9' Hawley

With thanks to
John W. Lockhart, *Red Hat*