*Live Video(H.264 encoded data) and Audio(AAC encoded data ) streaming using RTP/UDP over wireless network using GStreamers*

**Table of Contents**

[Abstract 4](#_Toc293691064)

[1. Introduction 4](#_Toc293691065)

[2. GStreamer Overview 4](#_Toc293691067)

[3. GStreamer Package 9](#_Toc293691090)

**4.**  **……………………………………………………12**

**5.** **………………………………………………………………………….41**

[References](#_Toc293691277)………………………………………………………………………………..41

**Table of Figures**

[Figure 1:Gstreamer Overview 3](#_Toc300322813)

[Figure 2:Source Element 4](#_Toc300322814)

[Figure 3:A Filter element with one output pad 4](#_Toc300322815)

[Figure 4: A Filter element with more than one output pad 5](#_Toc300322816)

[Figure 5: Sink Element 5](#_Toc300322817)

[Figure 6: Gstreamer Pipeline 6](#_Toc300322818)

[Figure 7: GStreamer pipeline with different communication flows 7](#_Toc300322819)

[Figure 8: Check available packages or plugins 9](#_Toc300322820)

[Figure 9:Check available specific element information 10](#_Toc300322821)

[Figure 10:Output of gst-launch command 11](#_Toc300322822)

[Figure 11:Pipeline for Video+Audio Player of mp4 file 12](#_Toc300322823)

[Figure 12:Output of Video and Audio(media file) playing together 1](#_Toc300322824)7

[Figure 13:Output of Video and Audio(media file) Playing stopped and End of Stream reached 1](#_Toc300322825)7

Figure 14: Compilation of server code…………………………………………………………. 27

Figure 15:Output of streaming Video+Audio Server application………………………………28

Figure 16: Compilation of client code…………………………………………………………...39

Figure 17: Output of streaming Video+Audio Client application……………………………...40

**Abstract**

My particular research will involve the **study and development of adaptive RTP streaming client for H.264 video over wireless network using GStreamer framework to write the client application**.

1. **Introduction**

**GStreamer:**  GStreamer is an open source multimedia framework. GStreamer is a framework for creating streaming media aplications. The GStreamer framework is designed to make it easy to write applications that handle audio or video or both.

1. **GStreamer Overview**

****

Figure :Gstreamer Overview

**2.1GStreamer Foundation**

**Elements:** An element is the most important class of objects in GStreamer. An element has one specific function, which can be the reading of data from a file, decoding of this data or outputting this data to the sound card (or display device). By chaining together several such elements, we can create a pipeline that can do a specific task, for example media playback or capture. GStreamer elements can be classified as sources, sinks and filters (or codecs). GStreamer ships with a large collection of elements by default, making the development of a large variety of media applications possible. If needed, we can also write new elements.

**Source Elements:** Source elements generate data for use by a pipeline, for example reading from disk or from a sound card. Source elements do not accept data, they only generate data. It has only a source pad (on the right).

****

Figure :Source Element

**Filters Elements:** Filters and filter-like elements have both input and outputs pads. They operate on data that they receive on their input (sink) pads, and will provide data on their output (source) pads. Examples of such elements convertors, demuxers, muxers and encoder and decoder(codecs). Filter elements can have any number of source or sink pads. A video demuxer, for example, would have one sink pad and several source pads, one for each elementary stream contained in the container format.

****

Figure :A Filter element with one output pad

****

Figure : A Filter element with more than one output pad

**Sink Elements:** Sink elements are end points in a media pipeline. They accept data but do not produce anything. Disk writing, soundcard playback, and video output would all be implemented by sink elements.

****

Figure : Sink Element

**Pads** : Pads are element’s input and output, where you can connect other elements. They are used to negotiate links and data flow between elements in GStreamer. A pad can be viewed as a “plug” or “port” on an element where links may be made with other elements, and through which data can flow to or from those elements. Pads have specific data handling capabilities: a pad can restrict the type of data that flows through it. Links are only allowed between two pads when the allowed data types of the two pads are compatible. Data types are negotiated between pads using a process called caps negotiation. Pads are element’s input and output, where you can connect other elements. They are used to negotiate links and data flow between elements in GStreamer. A source pad produces data buffers, while a sink pad consumes data buffers. A pad is similar to a plug or jack on a physical device.

**Bin:** A bin is a container for a collection of elements. A pipeline is a special subtype of a bin that allows execution of all of its contained child elements. Elements can be linked together by connecting their pads. A pipeline is a top-level bin. As you set it to PAUSED or PLAYING state, data flow will start and media processing will take place. Once started, pipelines will run in a separate thread until you stop them or the end of the data stream is reached.

**Element States:** After an element has been created it will not actually perform any actions yet. We need to change elements state to make it do something.

Four States of the Elements:

GST\_STATE\_NULL: This is the default state. No resources are allocated in this state.

GST\_STATE\_READY: In this state, an element has allocated all of its global resources. We can think about opening devices, allocating buffer.

GST\_STATE\_PAUSED: In this state, an element has opened the stream, but is not actively processing it.

GST\_STATE\_PLAYING: In the PLAYING state, accept and process events and buffers with data. In this state the clock now runs.

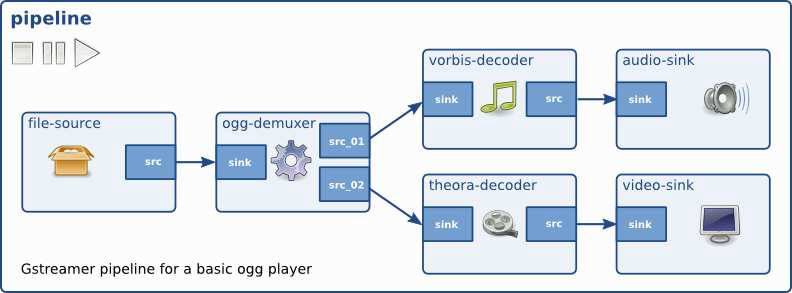
****

Figure : Gstreamer Pipeline

**Communication:** GStreamer provides several mechanisms for communication and data exchange between the application and the pipeline.

**Buffers** are objects for passing streaming data between elements in the pipeline. Buffers always travel from sources to sinks (downstream).

**Events** are objects sent between elements or from the application to elements. Events can travel

upstream and downstream.

**Bus** is a simple system that takes care of forwarding messages from the pipeline threads to an

application in its own thread context. The advantage of a bus is that an application does not need to be thread-aware in order to use GStreamer, even though GStreamer itself is heavily threaded.

Every pipeline contains a bus by default, so applications do not need to create a bus or anything. The only thing applications should do is set a message handler on a bus, which is similar to a signal handler to an object. When the mainloop is running, the bus will periodically be checked for new messages, and the callback will be called when any message is available.

**Messages** are objects posted by elements on the pipeline’s message bus, where they will be held for collection by the application. Messages can be intercepted synchronously from the streaming thread context of the element posting the message, but are usually handled asynchronously by the application from the application’s main thread. Messages are used to transmit information such as errors, tags, state changes, buffering state, redirects etc. from elements to the application in a thread-safe way.

**Queries** allow applications to request information such as duration or current playback position from the pipeline. Queries are always answered synchronously. Elements can also use queries to request information from their peer elements (such as the file size or duration). They can be used both ways within a pipeline, but upstream queries are more common.

****

Figure : GStreamer pipeline with different communication flows

**2.2 Design principles**

**Clean and powerful**: The programmer can use an extensive set of powerful tools to create media pipelines without writing a single line of code. Performing complex media manipulations becomes very easy. Plugins programmers are provided a clean and simple API to create self-contained plugins. GStreamer also comes with an extensive set of real-life plugins that serve as examples too.

**Object oriented** : The GStreamer framework is based on Glib, a C library that allows object-oriented code to be developed. GStreamer adheres to GObject, the GLib 2.0 object model. GStreamer uses the mechanism of signals and object properties. All objects can be queried at runtime for their various properties and capabilities.

**Extensible** : All GStreamer Objects can be extended using the GObject inheritance methods.

**High performance** : High performance is obtained by:

Using GLib’s GSlice allocator

Extremely light-weight links between plugins.

Data can travel the pipeline with minimal overhead.

Reference counting and copy on write minimize usage of memcpy. Sub-buffers efficiently split buffers into manageable pieces.

**2.3 GStreamer Plugins**

The framework is based on plug-ins that will provide the various codec and other functionality.

GStreamer plug-ins could be classified into:

sources: for audio and video

formats: muxers, demuxers

codecs: encoders and decoders

sinks: for audio and video

1. **GStreamer Package**

**Gstreamer is packaged into**

**gstreamer: the core package**

**gst-plugins-base: an essential exemplary set of elements**

**gst-plugins-good: a set of good-quality plug-ins under LGPL**

**gst-plugins-ugly: a set of good-quality plug-ins that might pose distribution problems**

**gst-plugins-bad: a set of plug-ins that need more quality**

**Few others packages.**

**3.1 Install GStreamer and Setup in Linux(Ubuntu) machine:**

**Debian :**Debian is free operating system is the set of basic programs and utilities that make your computer run. Debian uses the Linux kernel, but most of the basic OS tools come from the GNU Project, hence the name GNU/Linux.).

* **Install and Setup GStreamer in Debian / Ubuntu:**

-Tools for use with GStreamer

* student@ubuntu:~$ sudo apt-get install gstreamer-tools

- GNU libtool is a generic library support script. Libtool hides the complexity of using shared libraries behind a consistent, portable interface.

* student@ubuntu:~$sudo apt-get install libtool

-This package contains Gstreamer core development files for the library and elements.

* student@ubuntu:~$ sudo apt-get install libgstreamer0.10-dev
* **To Check available packages or plugins:**
* student@ubuntu:$ apt-cache search -n gstreamer0.10-plugins

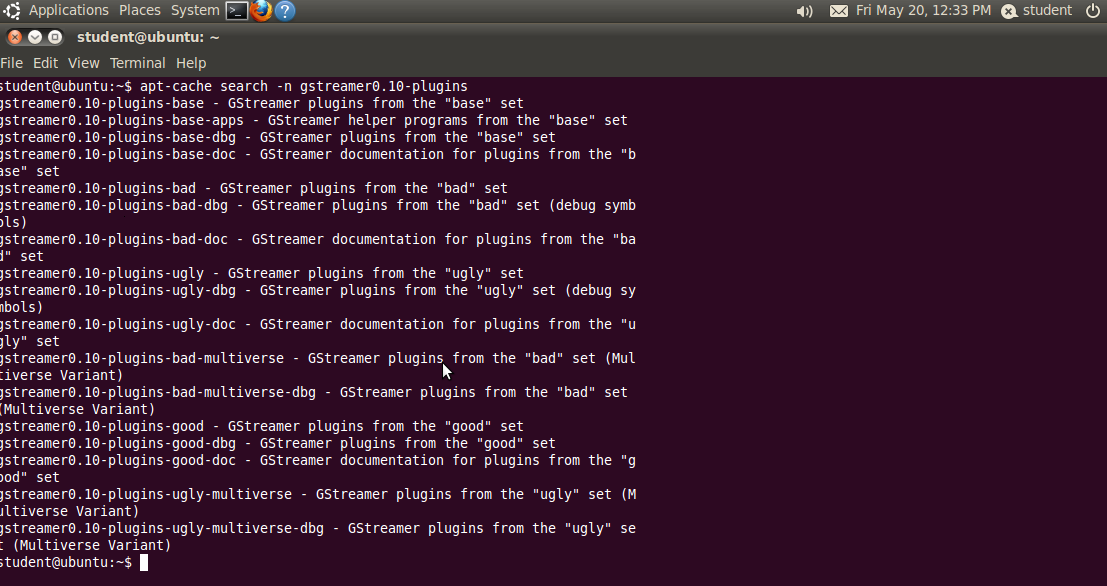
****

Figure : Check available packages or plugins

* **To Check available specific element information :**
* student@ubuntu:~$ gst-inspect filesrc

This should print out a bunch of information about this particular element ‘filesrc’.If the element is not found it will print that element is not found.

Then need to install the specific plugins with the latest version or available version for that element. For plugins and elements information refer this link:

<http://gstreamer.freedesktop.org/documentation/plugins.html>

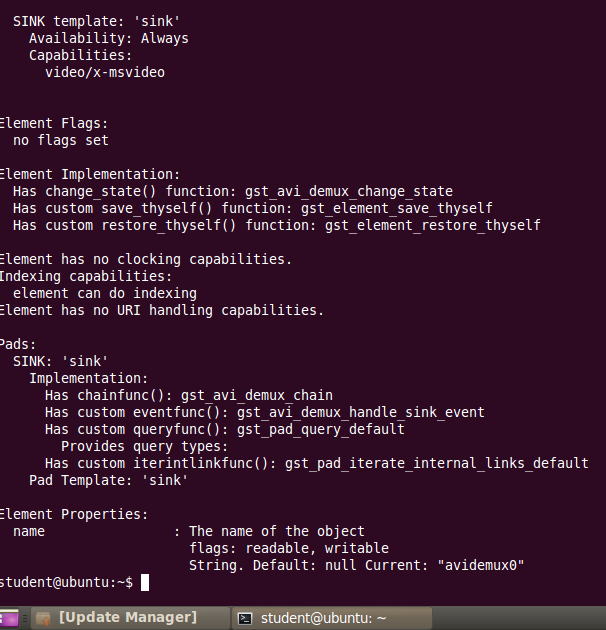
****

Figure :Check available specific element information

* **gst-launch is a simple script-like commandline application that can be used to test pipelines:**
* student@ubuntu:~/GStreamer/practice/demo$ gst-launch -v playbin uri=file:/home/student/GStreamer/practice/demo/foreman\_qcif.avi

This pipeline will play a media file which contains video+audio or only video or only audio stream.

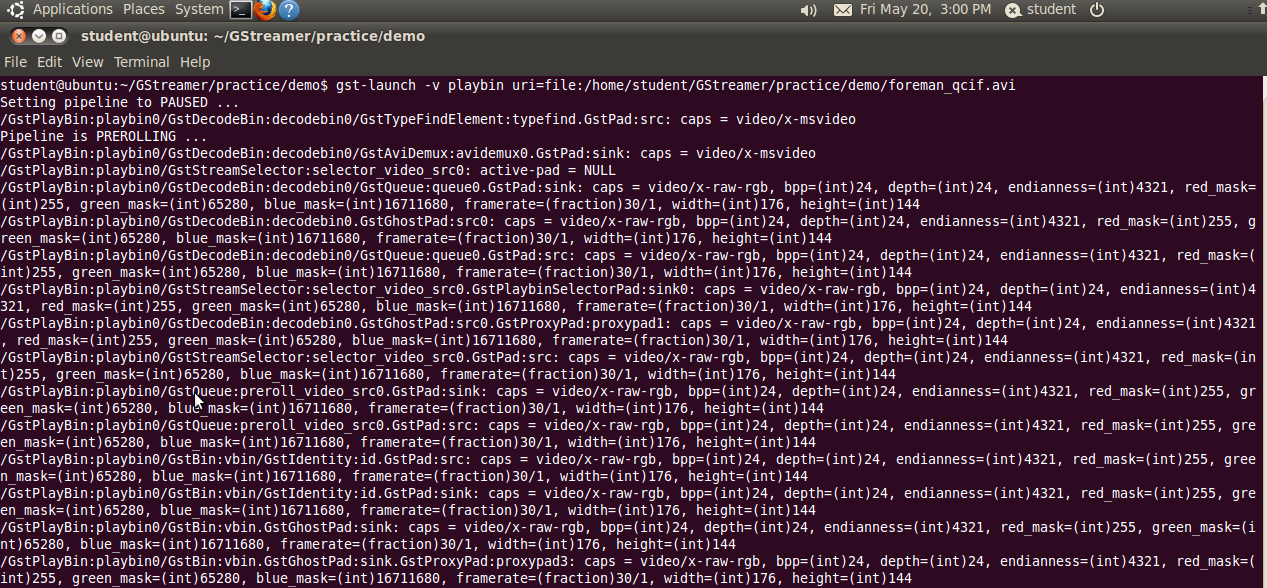
****

Figure :Output of gst-launch command

1. **Applications using GStreamer**

**I have written the following media player applications using c language in Ubuntu machine where I installed and set up the Gstreamer.**

* **Video+Audio Player Application**

queue

faad

audioconvert

autoaudiosink

qtdemux

filesrc

autovideosink

ffmpegcolorspace

queue

ffdec\_h264

Figure :Pipeline for Video+Audio Player of mp4 file

Here we need to add queue elements after the demuxer .The problem is that the pipeline will deadlock without queues: When the demuxer element is sending data to the first sink, the flow will block there while going from READY to PAUSED (this is called "prerolling"),if there are two streams video+audio available then this problem arise. Since the flow is blocked inside the first sink, the second sink will never receive any data which means that it can never reach the PAUSED state. A queue element decouples the flow by sending on data in another thread.

**Source Code of videoaudiotest.c:**

#include<gst/gst.h>

#include<glib.h>

static gboolean bus\_call(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 \*error;

gst\_message\_parse\_error(msg, &error,&debug);

g\_free(debug);

g\_printerr("Error :%s \n",error->message);

g\_error\_free(error);

g\_main\_loop\_quit(loop);

break;

}

default:

{

break;

}

}

return TRUE;

}

static void on\_pad\_added(GstElement \*element,GstPad \*pad, gpointer data)

{

GstPad \*sinkpad;

GstElement \*queue = (GstElement \*)data;

/\*we can now link this pad with the decoder sink pad\*/

sinkpad = gst\_element\_get\_static\_pad(queue, "sink");

gst\_pad\_link(pad, sinkpad);

gst\_object\_unref(sinkpad);

}

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

{

GMainLoop \*loop;

GstElement \*pipeline, \*source, \*demuxer,\*decoder, \*conv, \*sink,\*videodecoder,\*videoconv,\*videosink,\*videoqueue,\*audioqueue;

GstBus \*bus;

/\*Initialization\*/

gst\_init(&argc, &argv);

loop = g\_main\_loop\_new(NULL, FALSE);

/\*Check the input arguments\*/

if(argc != 2)

{

g\_printerr("Usage: %s <mp4 filename>\n", argv[0]);

return -1;

}

/\*Create gstremer elements \*/

pipeline = gst\_pipeline\_new("videoaudio-player");

source = gst\_element\_factory\_make("filesrc", "file-source");

demuxer = gst\_element\_factory\_make("qtdemux", "mp4-demuxer.");

decoder = gst\_element\_factory\_make("faad", "AAC-decoder");

conv = gst\_element\_factory\_make("audioconvert", "converter");

sink = gst\_element\_factory\_make("autoaudiosink", "audio-output");

videodecoder = gst\_element\_factory\_make("ffdec\_h264", "H264-decoder");

videoconv = gst\_element\_factory\_make("ffmpegcolorspace", "vconverter");

videosink = gst\_element\_factory\_make("autovideosink", "video-output");

audioqueue = gst\_element\_factory\_make("queue","audio-queue");

videoqueue = gst\_element\_factory\_make("queue", "video-queue");

if(!pipeline || !source || !demuxer || !decoder || !conv || !sink || !videodecoder || !videoconv || !videosink || !audioqueue || !videoqueue)

{

g\_printerr("One element could not be created .Existing.\n");

return -1;

}

/\*Set up the pipeline \*/

/\*We set the input filename to the source element\*/

g\_object\_set(G\_OBJECT(source), "location", argv[1], NULL);

/\*We add a message handler\*/

bus = gst\_pipeline\_get\_bus(GST\_PIPELINE(pipeline));

gst\_bus\_add\_watch(bus, bus\_call, loop);

gst\_object\_unref(bus);

/\*We add all elements into the pipeline \*/

gst\_bin\_add\_many(GST\_BIN(pipeline),source,demuxer,audioqueue, decoder,conv,sink,videoqueue,videodecoder,videoconv,videosink,NULL);

/\*we link the elements together\*/

gst\_element\_link(source, demuxer);

gst\_element\_link\_many(audioqueue,decoder,conv,sink, NULL);

gst\_element\_link\_many(videoqueue,videodecoder,videoconv,videosink, NULL);

g\_signal\_connect(demuxer, "pad-added", G\_CALLBACK(on\_pad\_added), audioqueue);

g\_signal\_connect(demuxer, "pad-added", G\_CALLBACK(on\_pad\_added), videoqueue);

/\*set the pipeline to playing state\*/

g\_print("Now playing %s\n", argv[1]);

gst\_element\_set\_state(pipeline, GST\_STATE\_PLAYING);

/\*Iterate\*/

g\_print("Running...\n");

g\_main\_loop\_run(loop);

/\*out of the main loop, clean up nicely\*/

g\_print("Returned, stopping playback\n");

gst\_element\_set\_state(pipeline, GST\_STATE\_NULL);

g\_print("Deleting pipeline \n");

gst\_object\_unref(GST\_OBJECT(pipeline));

return 0;

}

**Output:**

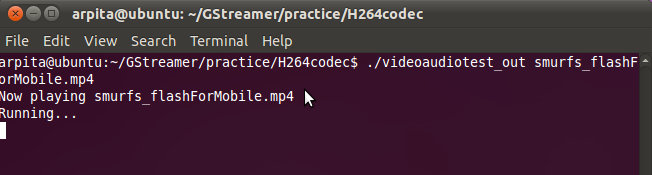
****

Figure :Output of Video and Audio(media file) playing together

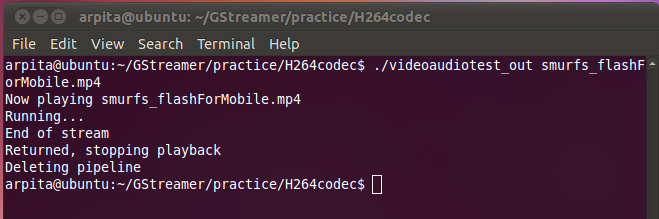
****

Figure :Output of Video and Audio(media file) Playing stopped and End of Stream reached

**Source Code for Streaming Server server\_H264\_AAC.c**

#include <string.h>

#include <math.h>

#include <gst/gst.h>

/\* A simple RTP server\*/

/\* change this to send the RTP data and RTCP to another host \*/

#define DEST\_HOST "127.0.0.1"

#define VOFFSET "0"

#define AOFFSET "0"

/\*H264 encode from the source\*/

#define VIDEO\_SRC "v4l2src"

#define VIDEO\_ENC "x264enc"

#define VIDEO\_PAY "rtph264pay"

/\*AAC encode from the source\*/

#define AUDIO\_SRC "autoaudiosrc"

#define AUDIO\_ENC "faac"

#define AUDIO\_PAY "rtpmp4gpay"

/\* print the stats of a source \*/

static void print\_source\_stats (GObject \* source)

{

GstStructure \*stats;

gchar \*str;

/\* get the source stats \*/

g\_object\_get (source, "stats", &stats, NULL);

/\* simply dump the stats structure \*/

str = gst\_structure\_to\_string (stats);

g\_print ("source stats: %s\n", str);

gst\_structure\_free (stats);

g\_free (str);

}

/\* this function is called every second and dumps the RTP manager stats \*/

static gboolean

print\_stats (GstElement \* rtpbin)

{

GObject \*session;

GValueArray \*arr;

GValue \*val;

guint i;

g\_print ("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

/\* get session 0 \*/

g\_signal\_emit\_by\_name (rtpbin, "get-internal-session", 0, &session);

/\* print all the sources in the session, this includes the internal source \*/

g\_object\_get (session, "sources", &arr, NULL);

for (i = 0; i < arr->n\_values; i++) {

GObject \*source;

val = g\_value\_array\_get\_nth (arr, i);

source = g\_value\_get\_object (val);

print\_source\_stats (source);

}

g\_value\_array\_free (arr);

g\_object\_unref (session);

return TRUE;

}

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

{

GstElement \*videosrc,\*audiosrc, \*videorate,\*audioconv,\*videoenc, \*audioenc,\*videopay, \*audiopay;

GstElement \*rtpbin, \*rtpsink, \*rtcpsink, \*rtcpsrc,\*audiortpsink,\*audiortcpsink,\*audiortcpsrc;

GstElement \*pipeline,\*videoqueue;

GMainLoop \*loop;

GstPad \*srcpad, \*sinkpad;

GstCaps \*caps;

gboolean link\_ok;

/\*Initializes the GStreamer library,registering built-in

\*elements, and loading standard plugins.Always init first.

\*/

gst\_init (&argc, &argv);

/\* The pipeline to hold everything \*/

pipeline = gst\_pipeline\_new (NULL);

g\_assert (pipeline);

/\*the video capture and format conversion\*/

videosrc = gst\_element\_factory\_make (VIDEO\_SRC,"videosrc");

g\_assert(videosrc);

videoqueue = gst\_element\_factory\_make("queue", "videoqueue");

g\_assert(videoqueue);

videorate = gst\_element\_factory\_make ("videorate","videorate");

g\_assert(videorate);

/\*Encodes video to H264 stream and RTP payloading\*/

videoenc = gst\_element\_factory\_make (VIDEO\_ENC, "videoenc");

g\_assert (videoenc);

g\_object\_set (videoenc, "tune",0x00000004,"byte-stream",TRUE,"bitrate",300,NULL);

videopay = gst\_element\_factory\_make (VIDEO\_PAY, "videopay");

g\_assert (videopay);

/\* The audio capture and format conversion \*/

audiosrc = gst\_element\_factory\_make (AUDIO\_SRC, "audiosrc");

g\_assert (audiosrc);

audioconv = gst\_element\_factory\_make ("audioconvert", "audioconv");

g\_assert (audioconv);

/\*Encodes audio to AAC stream and RTP payloading\*/

audioenc = gst\_element\_factory\_make (AUDIO\_ENC, "audioenc");

g\_assert (audioenc);

audiopay = gst\_element\_factory\_make (AUDIO\_PAY, "audiopay");

g\_assert (audiopay);

/\*Add video capture and payloading to the pipeline and link \*/

gst\_bin\_add\_many (GST\_BIN (pipeline),videosrc, videoqueue ,videorate,

videoenc, videopay,NULL);

if (!gst\_element\_link\_many (videosrc, videoqueue, videorate,NULL))

{

g\_error ("Failed to link videosrc, videoqueue, videorate");

}

caps = gst\_caps\_new\_simple ("video/x-raw-yuv",

"framerate", GST\_TYPE\_FRACTION, 15, 1,

NULL);

link\_ok = gst\_element\_link\_filtered (videorate, videoenc, caps);

gst\_caps\_unref (caps);

if (!link\_ok)

{

g\_warning ("Failed to link videorate and videoenc");

}

if (!gst\_element\_link\_many (videoenc,videopay, NULL))

{

g\_error ("Failed to link video encoder and video payloader");

}

/\* The rtpbin element \*/

rtpbin = gst\_element\_factory\_make ("gstrtpbin", "rtpbin");

g\_assert (rtpbin);

gst\_bin\_add (GST\_BIN (pipeline), rtpbin);

/\* The udp sinks and source we will use for RTP and RTCP \*/

rtpsink = gst\_element\_factory\_make ("udpsink", "rtpsink");

g\_assert (rtpsink);

g\_object\_set (rtpsink, "port", 5000,"host", DEST\_HOST, NULL);

g\_object\_set(rtpsink,"ts-offset",VOFFSET,NULL);

rtcpsink = gst\_element\_factory\_make ("udpsink", "rtcpsink");

g\_assert (rtcpsink);

g\_object\_set (rtcpsink, "port", 5001,"host",DEST\_HOST, NULL);

/\* No need for synchronisation or preroll on the RTCP sink \*/

g\_object\_set (rtcpsink, "async", FALSE, "sync", FALSE, NULL);

rtcpsrc = gst\_element\_factory\_make ("udpsrc", "rtcpsrc");

g\_assert (rtcpsrc);

g\_object\_set (rtcpsrc, "port", 5005, NULL);

gst\_bin\_add\_many (GST\_BIN (pipeline), rtpsink, rtcpsink, rtcpsrc, NULL);

/\* Now link all to the rtpbin,start by getting an RTP sinkpad for session 0\*/

sinkpad = gst\_element\_get\_request\_pad (rtpbin, "send\_rtp\_sink\_0");

srcpad = gst\_element\_get\_static\_pad (videopay, "src");

if (gst\_pad\_link (srcpad, sinkpad) != GST\_PAD\_LINK\_OK)

g\_error ("Failed to link video payloader to rtpbin");

gst\_object\_unref (srcpad);

/\* Get the RTP srcpad that was created when we requested the sinkpad above

\* and link it to the rtpsink sinkpad.

\*/

srcpad = gst\_element\_get\_static\_pad (rtpbin, "send\_rtp\_src\_0");

sinkpad = gst\_element\_get\_static\_pad (rtpsink, "sink");

if (gst\_pad\_link (srcpad, sinkpad) != GST\_PAD\_LINK\_OK)

g\_error ("Failed to link rtpbin to rtpsink");

gst\_object\_unref (srcpad);

gst\_object\_unref (sinkpad);

/\* get an RTCP srcpad for sending RTCP to the receiver \*/

srcpad = gst\_element\_get\_request\_pad (rtpbin, "send\_rtcp\_src\_0");

sinkpad = gst\_element\_get\_static\_pad (rtcpsink, "sink");

if (gst\_pad\_link (srcpad, sinkpad) != GST\_PAD\_LINK\_OK)

g\_error ("Failed to link rtpbin to rtcpsink");

gst\_object\_unref (sinkpad);

/\* We also want to receive RTCP, request an RTCP sinkpad for session 0 and

\* link it to the srcpad of the udpsrc for RTCP \*/

srcpad = gst\_element\_get\_static\_pad (rtcpsrc, "src");

sinkpad = gst\_element\_get\_request\_pad (rtpbin, "recv\_rtcp\_sink\_0");

if (gst\_pad\_link (srcpad, sinkpad) != GST\_PAD\_LINK\_OK)

g\_error ("Failed to link rtcpsrc to rtpbin");

gst\_object\_unref (srcpad);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Audio\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*Add audio capture and payloading to the pipeline and link \*/

gst\_bin\_add\_many (GST\_BIN (pipeline), audiosrc, audioconv, audioenc, audiopay,NULL);

if (!gst\_element\_link\_many (audiosrc,audioconv,audioenc,audiopay,NULL))

{

g\_error ("Failed to link audiosrc, audioconv, audioenc,audiopay");

}

/\* The udp sinks and source we will use for RTP and RTCP \*/

audiortpsink = gst\_element\_factory\_make ("udpsink", "audiortpsink");

g\_assert (audiortpsink);

g\_object\_set (audiortpsink, "port", 5002,"host", DEST\_HOST, NULL);

g\_object\_set(audiortpsink,"ts-offset",AOFFSET,NULL);

audiortcpsink = gst\_element\_factory\_make ("udpsink", "audiortcpsink");

g\_assert (audiortcpsink);

g\_object\_set (audiortcpsink, "port", 5003,"host",DEST\_HOST, NULL);

/\* No need for synchronisation or preroll on the RTCP sink \*/

g\_object\_set (audiortcpsink, "async", FALSE, "sync", FALSE, NULL);

audiortcpsrc = gst\_element\_factory\_make ("udpsrc", "audiortcpsrc");

g\_assert (audiortcpsrc);

g\_object\_set (audiortcpsrc, "port", 5007, NULL);

gst\_bin\_add\_many (GST\_BIN (pipeline), audiortpsink, audiortcpsink, audiortcpsrc, NULL);

/\* Now link all to the rtpbin,start by getting an RTP sinkpad for session 1\*/

sinkpad = gst\_element\_get\_request\_pad (rtpbin, "send\_rtp\_sink\_1");

srcpad = gst\_element\_get\_static\_pad (audiopay, "src");

if (gst\_pad\_link (srcpad, sinkpad) != GST\_PAD\_LINK\_OK)

g\_error ("Failed to link audio payloader to rtpbin");

gst\_object\_unref (srcpad);

/\* Get the RTP srcpad that was created when we requested the sinkpad above

\* and link it to the audiortpsink sinkpad.

\*/

srcpad = gst\_element\_get\_static\_pad (rtpbin, "send\_rtp\_src\_1");

sinkpad = gst\_element\_get\_static\_pad (audiortpsink, "sink");

if (gst\_pad\_link (srcpad, sinkpad) != GST\_PAD\_LINK\_OK)

g\_error ("Failed to link rtpbin to audiortpsink");

gst\_object\_unref (srcpad);

gst\_object\_unref (sinkpad);

/\* get an RTCP srcpad for sending RTCP to the receiver \*/

srcpad = gst\_element\_get\_request\_pad (rtpbin, "send\_rtcp\_src\_1");

sinkpad = gst\_element\_get\_static\_pad (audiortcpsink, "sink");

if (gst\_pad\_link (srcpad, sinkpad) != GST\_PAD\_LINK\_OK)

g\_error ("Failed to link rtpbin to audiortcpsink");

gst\_object\_unref (sinkpad);

/\* We also want to receive RTCP, request an RTCP sinkpad for session 1 and

\* link it to the srcpad of the udpsrc for RTCP

\*/

srcpad = gst\_element\_get\_static\_pad (audiortcpsrc, "src");

sinkpad = gst\_element\_get\_request\_pad (rtpbin, "recv\_rtcp\_sink\_1");

if (gst\_pad\_link (srcpad, sinkpad) != GST\_PAD\_LINK\_OK)

g\_error ("Failed to link rtcpsrc to rtpbin");

gst\_object\_unref (srcpad);

/\* Set the pipeline to playing \*/

g\_print ("starting sender pipeline\n");

gst\_element\_set\_state (pipeline, GST\_STATE\_PLAYING);

/\* print stats every second \*/

g\_timeout\_add (1000, (GSourceFunc) print\_stats, rtpbin);

/\* we need to run a GLib main loop to get the messages \*/

loop = g\_main\_loop\_new (NULL, FALSE);

g\_main\_loop\_run (loop);

g\_print ("stopping sender pipeline\n");

gst\_element\_set\_state (pipeline, GST\_STATE\_NULL);

return 0;

}

**Compile code and create the output file:**

$**gcc -Wall $(pkg-config --cflags --libs gstreamer- 0.10) server\_H264\_AAC.c -o server\_H264\_AAC\_out**

GStreamer uses pkg-config to assist applications with compilation and linking flags. Here passed the --cflags and --libs arguments to pkg-config.

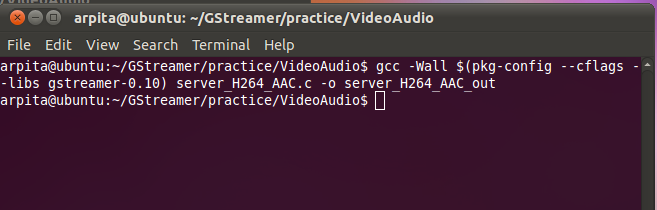
****

Figure : Compilation of server code

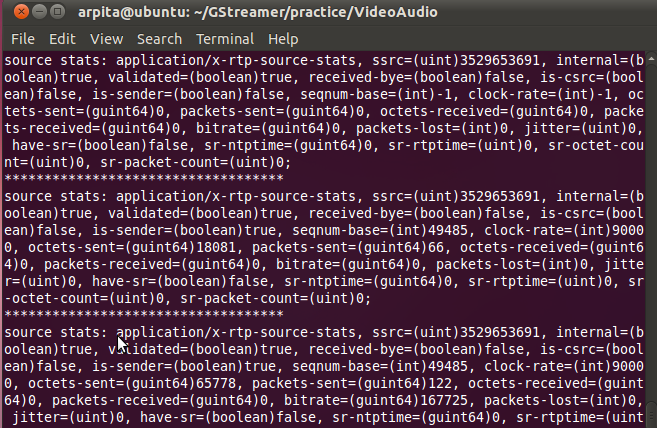
****

Figure :Output of streaming Video+Audio Server application

**Source Code for Streaming Client new\_client\_H264\_AAC.c:**

#include <string.h>

#include <math.h>

#include <gst/gst.h>

/\*

\* A simple RTP receiver

\*/

/\*

\* the caps of the sender RTP stream. This is usually negotiated out of band

\* with SDP or RTSP.

\*/

#define VIDEO\_CAPS "application/x-rtp,media=(string)video,clock-rate=(int)90000,encoding-name=(string)H264"

#define AUDIO\_CAPS "application/x-rtp,media=(string)audio,clock-rate=(int)44100,encoding-name=(string)MPEG4-GENERIC,encoding-params=(string)1,streamtype=(string)5,profile-level-id=(string)2,mode=(string)AAC-hbr,config=(string)1208,sizelength=(string)13,indexlength=(string)3,indexdeltalength=(string)3,ssrc=(uint)853015980,payload=(int)96,clock-base=(uint)2040203639,seqnum-base=(uint)52067"

#define VIDEO\_DEPAY "rtph264depay"

#define VIDEO\_DEC "ffdec\_h264"

#define VIDEO\_SINK "autovideosink"

#define VIDEO\_CONV "ffmpegcolorspace"

#define AUDIO\_DEPAY "rtpmp4gdepay"

#define AUDIO\_DEC "faad"

#define AUDIO\_SINK "autoaudiosink"

/\*

\*the destination machine to send RTCP to. This is the address of the sender

\*and is used to send back the RTCP reports of this receiver. If the data is

\*sent from another machine, change this address.

\*/

#define DEST\_HOST "127.0.0.1"

#define LATENCY 100

/\* print the stats of a source \*/

static void

print\_source\_stats (GObject \* source)

{

GstStructure \*stats;

gchar \*str;

g\_return\_if\_fail (source != NULL);

/\* get the source stats \*/

g\_object\_get (source, "stats", &stats, NULL);

/\* simply dump the stats structure \*/

str = gst\_structure\_to\_string (stats);

g\_print ("source stats: %s\n", str);

gst\_structure\_free (stats);

g\_free (str);

}

/\* will be called when gstrtpbin signals on-ssrc-active. It means that an RTCP

\* packet was received from another source. \*/

static void

on\_ssrc\_active\_cb (GstElement \* rtpbin, guint sessid, guint ssrc,

GstElement \* depay)

{

GObject \*session, \*isrc, \*osrc;

g\_print ("got RTCP from session %u, SSRC %u\n", sessid, ssrc);

/\* get the right session \*/

g\_signal\_emit\_by\_name (rtpbin, "get-internal-session", sessid, &session);

/\* get the internal source (the SSRC allocated to us, the receiver \*/

g\_object\_get (session, "internal-source", &isrc, NULL);

print\_source\_stats (isrc);

/\* get the remote source that sent us RTCP \*/

g\_signal\_emit\_by\_name (session, "get-source-by-ssrc", ssrc, &osrc);

print\_source\_stats (osrc);

}

/\* will be called when rtpbin has validated a payload that we can depayload \*/

static void

pad\_added\_cb (GstElement \* rtpbin, GstPad \* new\_pad, GstElement\* depayarray[])

{

GstPad \*sinkpad;

GstPadLinkReturn lres;

GstElement\* depay;

g\_print ("new payload on pad: %s\n", GST\_PAD\_NAME (new\_pad));

if(strstr(GST\_PAD\_NAME (new\_pad),"recv\_rtp\_src\_0"))

{

depay=depayarray[0];

sinkpad = gst\_element\_get\_static\_pad (depay, "sink");

g\_assert (sinkpad);

lres = gst\_pad\_link (new\_pad, sinkpad);

g\_assert (lres == GST\_PAD\_LINK\_OK);

gst\_object\_unref (sinkpad);

}

else if(strstr(GST\_PAD\_NAME (new\_pad),"recv\_rtp\_src\_1"))

{

depay=depayarray[1];

sinkpad = gst\_element\_get\_static\_pad (depay, "sink");

g\_assert (sinkpad);

lres = gst\_pad\_link (new\_pad, sinkpad);

g\_assert (lres == GST\_PAD\_LINK\_OK);

gst\_object\_unref (sinkpad);

}

else

{

g\_print ("pad: %s not connected\n", GST\_PAD\_NAME (new\_pad));

}

}

/\* build a pipeline equivalent to:

gst-launch -v gstrtpbin name=rtpbin latency=$LATENCY \

udpsrc caps=$VIDEO\_CAPS port=5000 ! rtpbin.recv\_rtp\_sink\_0 \

rtpbin. ! $VIDEO\_DEC ! $VIDEO\_SINK \

udpsrc port=5001 ! rtpbin.recv\_rtcp\_sink\_0 \

rtpbin.send\_rtcp\_src\_0 ! udpsink port=5005 host=$DEST sync=false async=false\*/

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

{

GstElement \*rtpbin, \*rtpsrc, \*rtcpsrc, \*rtcpsink,\*audiortpsrc,\*audiortcpsrc,\*audiortcpsink;

GstElement \*videodepay, \*videodec, \*videoconv, \*videosink;

GstElement \*audiodepay, \*audiodec, \*audiosink;

GstElement\* depayarray[2];

GstElement \*pipeline;

GMainLoop \*loop;

GstCaps \*caps;

gboolean res;

GstPadLinkReturn lres;

GstPad \*srcpad, \*sinkpad;

/\* always init first \*/

gst\_init (&argc, &argv);

/\* the pipeline to hold everything \*/

pipeline = gst\_pipeline\_new (NULL);

g\_assert (pipeline);

/\* the udp src and source we will use for RTP and RTCP \*/

rtpsrc = gst\_element\_factory\_make ("udpsrc", "rtpsrc");

g\_assert (rtpsrc);

g\_object\_set (rtpsrc, "port", 5000, NULL);

/\* we need to set caps on the udpsrc for the RTP data \*/

caps = gst\_caps\_from\_string (VIDEO\_CAPS);

g\_object\_set (rtpsrc, "caps", caps, NULL);

gst\_caps\_unref (caps);

rtcpsrc = gst\_element\_factory\_make ("udpsrc", "rtcpsrc");

g\_assert (rtcpsrc);

g\_object\_set (rtcpsrc, "port", 5001, NULL);

rtcpsink = gst\_element\_factory\_make ("udpsink", "rtcpsink");

g\_assert (rtcpsink);

g\_object\_set (rtcpsink, "port", 5005, "host", DEST\_HOST, NULL);

/\* no need for synchronisation or preroll on the RTCP sink \*/

g\_object\_set (rtcpsink, "async", FALSE, "sync", FALSE, NULL);

gst\_bin\_add\_many (GST\_BIN (pipeline), rtpsrc, rtcpsrc, rtcpsink, NULL);

/\* the depayloading and decoding \*/

videodepay = gst\_element\_factory\_make (VIDEO\_DEPAY, "videodepay");

g\_assert (videodepay);

videodec = gst\_element\_factory\_make (VIDEO\_DEC, "videodec");

g\_assert (videodec);

/\*the video playback and format conversion \*/

videoconv = gst\_element\_factory\_make ("ffmpegcolorspace", "videoconv");

g\_assert (videoconv);

videosink = gst\_element\_factory\_make (VIDEO\_SINK, "videosink");

g\_assert (videosink);

/\* add depayloading and playback to the pipeline and link \*/

gst\_bin\_add\_many (GST\_BIN (pipeline), videodepay, videodec, videoconv,

videosink, NULL);

res =gst\_element\_link\_many (videodepay, videodec, videoconv, videosink, NULL);

g\_assert (res == TRUE);

/\* the rtpbin element \*/

rtpbin = gst\_element\_factory\_make ("gstrtpbin", "rtpbin");

g\_assert (rtpbin);

gst\_bin\_add (GST\_BIN (pipeline), rtpbin);

/\* now link all to the rtpbin, start by getting an RTP sinkpad for session 0\*/

srcpad = gst\_element\_get\_static\_pad (rtpsrc, "src");

sinkpad = gst\_element\_get\_request\_pad (rtpbin, "recv\_rtp\_sink\_0");

lres = gst\_pad\_link (srcpad, sinkpad);

g\_assert (lres == GST\_PAD\_LINK\_OK);

gst\_object\_unref (srcpad);

gst\_object\_unref (sinkpad);

/\* get an RTCP sinkpad in session 0 \*/

srcpad = gst\_element\_get\_static\_pad (rtcpsrc, "src");

sinkpad = gst\_element\_get\_request\_pad (rtpbin, "recv\_rtcp\_sink\_0");

lres = gst\_pad\_link (srcpad, sinkpad);

g\_assert (lres == GST\_PAD\_LINK\_OK);

gst\_object\_unref (srcpad);

gst\_object\_unref (sinkpad);

/\* get an RTCP srcpad for sending RTCP back to the sender \*/

srcpad = gst\_element\_get\_request\_pad (rtpbin, "send\_rtcp\_src\_0");

sinkpad = gst\_element\_get\_static\_pad (rtcpsink, "sink");

lres = gst\_pad\_link (srcpad, sinkpad);

g\_assert (lres == GST\_PAD\_LINK\_OK);

gst\_object\_unref (sinkpad);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Audio\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* the udp src and source we will use for RTP and RTCP \*/

audiortpsrc = gst\_element\_factory\_make ("udpsrc", "audiortpsrc");

g\_assert (audiortpsrc);

g\_object\_set (audiortpsrc, "port", 5002, NULL);

/\* we need to set caps on the udpsrc for the RTP data \*/

caps = gst\_caps\_from\_string (AUDIO\_CAPS);

g\_object\_set (audiortpsrc, "caps", caps, NULL);

gst\_caps\_unref (caps);

audiortcpsrc = gst\_element\_factory\_make ("udpsrc", "audiortcpsrc");

g\_assert (audiortcpsrc);

g\_object\_set (audiortcpsrc, "port", 5003, NULL);

audiortcpsink = gst\_element\_factory\_make ("udpsink", "audiortcpsink");

g\_assert (audiortcpsink);

g\_object\_set (audiortcpsink, "port", 5005, "host", DEST\_HOST, NULL);

/\* no need for synchronisation or preroll on the RTCP sink \*/

g\_object\_set (audiortcpsink, "async", FALSE, "sync", FALSE, NULL);

gst\_bin\_add\_many (GST\_BIN (pipeline), audiortpsrc, audiortcpsrc, audiortcpsink, NULL);

/\* the depayloading and decoding \*/

audiodepay = gst\_element\_factory\_make (AUDIO\_DEPAY, "audiodepay");

g\_assert (audiodepay);

audiodec = gst\_element\_factory\_make (AUDIO\_DEC, "audiodec");

g\_assert (audiodec);

/\*the audio playback \*/

audiosink = gst\_element\_factory\_make (AUDIO\_SINK, "audiosink");

g\_assert (audiosink);

/\* add depayloading and playback to the pipeline and link \*/

gst\_bin\_add\_many (GST\_BIN (pipeline), audiodepay, audiodec, audiosink, NULL);

res =gst\_element\_link\_many (audiodepay, audiodec, audiosink, NULL);

g\_assert (res == TRUE);

/\* now link all to the rtpbin, start by getting an RTP sinkpad for session 1\*/

srcpad = gst\_element\_get\_static\_pad (audiortpsrc, "src");

sinkpad = gst\_element\_get\_request\_pad (rtpbin, "recv\_rtp\_sink\_1");

lres = gst\_pad\_link (srcpad, sinkpad);

g\_assert (lres == GST\_PAD\_LINK\_OK);

gst\_object\_unref (srcpad);

gst\_object\_unref (sinkpad);

/\* get an RTCP sinkpad in session 1 \*/

srcpad = gst\_element\_get\_static\_pad (audiortcpsrc, "src");

sinkpad = gst\_element\_get\_request\_pad (rtpbin, "recv\_rtcp\_sink\_1");

lres = gst\_pad\_link (srcpad, sinkpad);

g\_assert (lres == GST\_PAD\_LINK\_OK);

gst\_object\_unref (srcpad);

gst\_object\_unref (sinkpad);

/\* get an RTCP srcpad for sending RTCP back to the sender \*/

srcpad = gst\_element\_get\_request\_pad (rtpbin, "send\_rtcp\_src\_1");

sinkpad = gst\_element\_get\_static\_pad (audiortcpsink, "sink");

lres = gst\_pad\_link (srcpad, sinkpad);

g\_assert (lres == GST\_PAD\_LINK\_OK);

gst\_object\_unref (sinkpad);

depayarray[0]=videodepay;

depayarray[1]=audiodepay;

/\* the RTP pad that we have to connect to the depayloader will be created

\* dynamically so we connect to the pad-added signal, pass the depayloader as

\* user\_data so that we can link to it. \*/

g\_signal\_connect (rtpbin, "pad-added", G\_CALLBACK (pad\_added\_cb), depayarray);

/\* give some stats when we receive RTCP \*/

g\_signal\_connect (rtpbin, "on-ssrc-active", G\_CALLBACK (on\_ssrc\_active\_cb),

videodepay);

/\* set the pipeline to playing \*/

g\_print ("starting receiver pipeline\n");

gst\_element\_set\_state (pipeline, GST\_STATE\_PLAYING);

/\* we need to run a GLib main loop to get the messages \*/

loop = g\_main\_loop\_new (NULL, FALSE);

g\_main\_loop\_run (loop);

g\_print ("stopping receiver pipeline\n");

gst\_element\_set\_state (pipeline, GST\_STATE\_NULL);

gst\_object\_unref (pipeline);

return 0;

}

**Compile code and create the output file:**

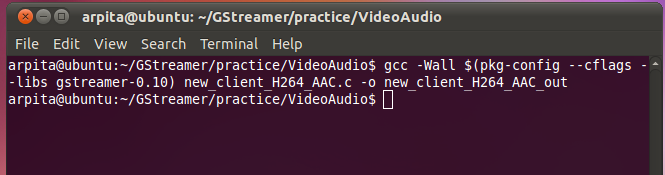


Figure : Compilation of client code

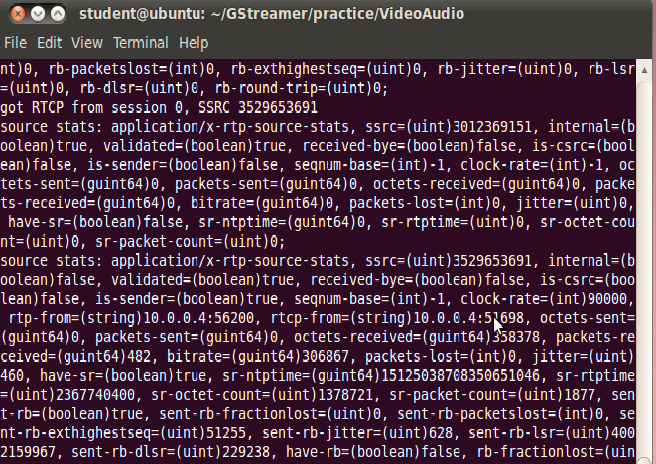


Figure : Output of streaming Video+Audio Client application

1. **Conclusion**

This project involves a deal of knowledge in the area of video and audio streaming over wireless network, Gstreamer framework to write streaming media application .I have gathered more knowledge in Gstreamer and able to do live streaming H.264 video and AAC audio over wireless network using RTP/UDP in unicast environment. Next I will try for multicast environment and I am planning for Adaptive Video Streaming that will adjusts the quality of a video delivered based on changing network conditions to ensure better quality video in client side.

**References**

1. **Google Search Engine**

[**www.google.com**](http://www.google.com)

1. [**http://gstreamer.freedesktop.org**](http://gstreamer.freedesktop.org/)