Gstreamer is a tool for manipulating video streams. It is both a software library and that library called from a command line tool. The purposes I have used it for is mainly to stream video in real time over a local area IP network. Doing that I found a lack of basic tutorials on how to do that using the command line.

This tutorial covers the basics of live streaming. To read or write files will not be covered, though the basics of that is easy following the same principles as broadcasting the streams. If you want to see a comparison of the real time capabilities of the streams, that is a post coming soon.

“Real time” in this context is if the streams passed a sort of Turing test. If you cannot distinguish between the original and the copy, it passes. Note that because of the quite short deadline for the images I didn’t add any buffers to my stream, if the image is not within its deadline it should be discarded. The use of a queue element could have smoothed the stream but to the price of higher latency.

This tutorial assumes you are using gstreamer 1.0.

## **Basics**

Gstreamer consists of several command line applications. In this tutorial we focus on two of them: gst-launch-1.0 and gst-inspect-1.0  
gst-launch-1.0 launches a new stream pipeline with the properties you set. The main part of the tutorial covers how that is done. gst-inspect-1.0 provides information on installed gstreamer modules.  
**The gstreamer architecture**

Gstreamer is constructed using a pipes and filter architecture. The pipes and filters can be added to each other much like unix pipelines but within the scope of gstreamer. The basic structure of a stream pipeline is that you start with a stream source (camera, screengrab, file etc) and end with a stream sink (screen window, file, network etc)

In gstreamer the pipe command is the exclamation mark – !

The ! connects the filters, but in gstreamer terminology they are called pads. The entire system of pads and filters is called a pipeline.

### The most basic stream

This stream launches the video test source and pipes it to the screen. Autovideosink is a useful abstraction. If you don’t have a particular preference for a screen sink it selects the standard one on your system. Use that.

gst-launch-1.0 videotestsrc ! autovideosink

**Adding caps to the stream**

Gstreamer has a filter called capabilities, caps for short. That changes some properties of the stream. What properties can be set depends on the type of stream.  
To start manipulating your stream, one of the first things you might want to do is change the properties of the raw stream. The following example changes the resolution to 800 x 600 pixels.

gst-launch-1.0 videotestsrc ! video/x-raw,width=640,height=480 ! autovideosink  
this is actually a short hand for  
gst-launch-1.0 videotestsrc ! capsfilter caps=video/x-raw,width=640,height=480 ! autovideosink  
There are more parameters to this, but for now this is enough.

### Feed from the camera

This step assumes you have a working camera attached to your system. The suorce for the linux camera is v4l2src

gst-launch-1.0 v4l2src device="/dev/video0" ! video/x-raw,width=640,height=480 ! autovideosink

This could actually fail depending on your cameras aspect ratio. If the proportions of the image is not 640/480 you will get an error saying something like streaming task paused, reason not-negotiated (-4). Just get used to it, that’s gstreamers notoriously uninformative error messages. I will come back to that later.  
Autovideosink is much more easy to get working, and is by that quite useful to debug your pipelines.

### Feed from screengrabber

For my purpose I wanted to use either a camera or a portion of the screen as a source. Gstreamer has screengrabbers. If you are on Linux it is ximagesrc, on windows it is XXX. Those are different sources and work in different ways. I will try to cover them both.

gst-launch-1.0 ximagesrc ! videoconvert ! autovideosink

videoconvert is a element that reads the input video and adapts it to a format that is understood by the output. In this case it is needed since the xvimagesrc has not defined its output format.

The video displayed by the previous command has a few downsides, it casts the entire screen and it can be quite jumpy.

For me the choppiness was mitigated by not using x-damage by adding the line x-damage=false. And to select just one window you can use xid=xxx or xname=xxx this is the names or id:s of the particular X-window you want as source. To get the names and ID:s try using wmctrl -l this gives you a list of all open windows.  
Finally you probably want to resize the stream for your needs.

The complete command is:

gst-launch-1.0 ximagesrc use-damage=false xname=<name of window> ! videoconvert ! videoscale ! video/x-raw,width=800,height=600 ! autovideosink

videoscale is a pad that once again negotiates the format of the stream so that the original images can be scaled using the capsfilter.

### Encode your stream

So far your stream has consisted of the uncompressed readings from the camera device and the testvideo. This is heavy to send over a network since it is unpacked video data with a high bit rate. But if the network capacity is high and the processors on the sending or receiving ends are slow this might be a solution to consider.

For this tutorial I will cover three types of video compression, Mjpeg, Mpeg 2 and V8. There are many more, and encoding those follow the same basic pattern as the examples in here.

This step is pretty simple. The videostreams that previously has been piped to autovideosink is now piped to the encoder pad with for example ! jpegdec for mjpeg encoding and ! vp8enc or ! avenc\_mpeg4 for VP8 and Mpeg 2. Those encoders have multiple elements that can be changed to create the stream you want. A complete list can be provided by gst-inspect-1.0 <encoder name>. Some of the more useful are

Executing those lines will result in an error since the encoder is not a sink. Which brings us to the next step which can either be payloading for network transfer or sending it to a sink. I this case the sink can’t be a network sink, since that requires the images to be loaded in a network protocol to be reliably transmitted.

### Payload the stream

Payloading is the step of packing the data, raw or compressed into a network protocol. This is a section where gstreamer gives you very few options. You can either use GDP (Gstreamer data protocol) or RTP (Real time protocol). RTP is an established standard from Internet engineering taskforce and the protocol you want to use if the stream is to be read by some application that is not gstreamer itself. And those are the only cases I cover in this tutorial.  
The payloader pad is simply added after the decoder pad in this fashion:  
! jpegenc ! rtpjpegpay  
! vp8enc ! rtpv8pay  
! avenc\_mpeg4 ! rtpmp4vpay

If you have a dedicated high bandwidth connection you could skip the encoding step and just payload the raw stream:

! rtpvrawpay

This step also has a numer of possible settings, those are availablye though gst-inspect-1.0 there is one setting you ned to do. And that is if you want to use mpeg2.

rtpmp4vpay config-interval=3 This sets the interval in seconds on how often the mpeg 2 specific settings are sent. The effects it has on a one to one stream is how many seconds it will take before the stream can be displayed.

### Send the stream

There are multiple ways to send this stream to be used by other recipients on a network.gst-inspect-1.0 | grep sink will show you all possibilities. In this tutorial I will only cover the autovideosink, the udpsink and the multiudpsink.

udpsink and multiudpsink are two similar sinks. The data piped to them is sent to one (udpsink) or several (multiudpsink) udp-adresses. They can of course be used to broadcast the data to whole subnets, but that should be a separate tutorial on broadcasting udp. The syntax of the sinks are simple. udpsink host=127.0.0.1 port=5000 sends the stream to the localhoast multiudpsink clients=127.0.0.1:5000,127.0.0.1:5004,192.168.2.15:2000 sends it to three different destinations.

### Complete examples for sending

When everything is combined it gives the following send commands:

#### Mjpeg

gst-launch-1.0 -v ximagesrc use-damage=false xname=/usr/lib/torcs/torcs-bin ! videoconvert ! videoscale ! video/x-raw,format=I420,width=800,height=600,framerate=25/1 ! jpegenc ! rtpjpegpay ! udpsink host=127.0.0.1 port=5000

#### VP8

gst-launch-1.0 -v ximagesrc use-damage=false xname=/usr/lib/torcs/torcs-bin ! videoconvert ! videoscale ! video/x-raw,width=800,height=600 ! vp8enc ! rtpvp8pay ! udpsink host=127.0.0.1 port=5100

#### Mpeg-2

gst-launch-1.0 -v ximagesrc use-damage=false xname=/usr/lib/torcs/torcs-bin ! videoconvert ! videoscale ! video/x-raw,width=800,height=600 ! avenc\_mpeg4 ! rtpmp4vpay config-interval=3 ! udpsink host=127.0.0.1 port=5200

The -v option after gst-launch-1.0 is telling the program to do verbose output. Some of the verbose output you get is required in order to construct the receivers of the streams. The last lines you get looks something like:

/GstPipeline:pipeline0/GstCapsFilter:capsfilter0.GstPad:src: caps = "video/x-raw\,\ width\=\(int\)800\,\ height\=\(int\)600\,\ framerate\=\(fraction\)25/1\,\ pixel-aspect-ratio\=\(fraction\)1/1\,\ format\=\(string\)I420"  
/GstPipeline:pipeline0/avenc\_mpeg4:avenc\_mpeg4-0.GstPad:sink: caps = "video/x-raw\,\ width\=\(int\)800\,\ height\=\(int\)600\,\ framerate\=\(fraction\)25/1\,\ pixel-aspect-ratio\=\(fraction\)1/1\,\ format\=\(string\)I420"  
/GstPipeline:pipeline0/GstCapsFilter:capsfilter0.GstPad:sink: caps = "video/x-raw\,\ width\=\(int\)800\,\ height\=\(int\)600\,\ framerate\=\(fraction\)25/1\,\ pixel-aspect-ratio\=\(fraction\)1/1\,\ format\=\(string\)I420"  
/GstPipeline:pipeline0/GstVideoScale:videoscale0.GstPad:sink: caps = "video/x-raw\,\ width\=\(int\)800\,\ height\=\(int\)600\,\ framerate\=\(fraction\)25/1\,\ pixel-aspect-ratio\=\(fraction\)1/1\,\ format\=\(string\)I420"  
/GstPipeline:pipeline0/GstVideoConvert:videoconvert0.GstPad:sink: caps = "video/x-raw\,\ format\=\(string\)BGRx\,\ width\=\(int\)800\,\ height\=\(int\)600\,\ framerate\=\(fraction\)25/1\,\ pixel-aspect-ratio\=\(fraction\)1/1"  
/GstPipeline:pipeline0/avenc\_mpeg4:avenc\_mpeg4-0.GstPad:src: caps = "video/mpeg\,\ mpegversion\=\(int\)4\,\ systemstream\=\(boolean\)false\,\ profile\=\(string\)simple\,\ width\=\(int\)800\,\ height\=\(int\)600\,\ framerate\=\(fraction\)25/1\,\ pixel-aspect-ratio\=\(fraction\)1/1"  
/GstPipeline:pipeline0/GstRtpMP4VPay:rtpmp4vpay0.GstPad:sink: caps = "video/mpeg\,\ mpegversion\=\(int\)4\,\ systemstream\=\(boolean\)false\,\ profile\=\(string\)simple\,\ width\=\(int\)800\,\ height\=\(int\)600\,\ framerate\=\(fraction\)25/1\,\ pixel-aspect-ratio\=\(fraction\)1/1"  
/GstPipeline:pipeline0/GstRtpMP4VPay:rtpmp4vpay0.GstPad:src: caps = "application/x-rtp\,\ media\=\(string\)video\,\ payload\=\(int\)96\,\ encoding-name\=\(string\)MP4V-ES\,\ ssrc\=\(uint\)1133315284\,\ timestamp-offset\=\(uint\)3049853928\,\ seqnum-offset\=\(uint\)23820"  
/GstPipeline:pipeline0/GstUDPSink:udpsink0.GstPad:sink: caps = "application/x-rtp\,\ media\=\(string\)video\,\ payload\=\(int\)96\,\ encoding-name\=\(string\)MP4V-ES\,\ ssrc\=\(uint\)1133315284\,\ timestamp-offset\=\(uint\)3049853928\,\ seqnum-offset\=\(uint\)23820"  
/GstPipeline:pipeline0/GstRtpMP4VPay:rtpmp4vpay0.GstPad:src: caps = "application/x-rtp\,\ media\=\(string\)video\,\ clock-rate\=\(int\)90000\,\ encoding-name\=\(string\)MP4V-ES\,\ profile-level-id\=\(string\)1\,\ config\=\(string\)000001b001000001b58913000001000000012000c48d8800cd19044b1443000001b24c61766335362e312e30\,\ payload\=\(int\)96\,\ ssrc\=\(uint\)1133315284\,\ timestamp-offset\=\(uint\)3049853928\,\ seqnum-offset\=\(uint\)23820"  
/GstPipeline:pipeline0/GstUDPSink:udpsink0.GstPad:sink: caps = "application/x-rtp\,\ media\=\(string\)video\,\ clock-rate\=\(int\)90000\,\ encoding-name\=\(string\)MP4V-ES\,\ profile-level-id\=\(string\)1\,\ config\=\(string\)000001b001000001b58913000001000000012000c48d8800cd19044b1443000001b24c61766335362e312e30\,\ payload\=\(int\)96\,\ ssrc\=\(uint\)1133315284\,\ timestamp-offset\=\(uint\)3049853928\,\ seqnum-offset\=\(uint\)23820"  
/GstPipeline:pipeline0/GstRtpMP4VPay:rtpmp4vpay0: timestamp = 3049854058  
/GstPipeline:pipeline0/GstRtpMP4VPay:rtpmp4vpay0: seqnum = 23820

This information is the capabilities of the streams sent from one pad to another. This is also information needed to recreate the stream by the receiver. The information needed is in one of the last lines starting with caps = "application/x-rtp, that is the caps of the output of the udpsink.

#### Receive the stream

This command is assumed to be executed on a client that can receive the udp stream sent by the udpsink. If it is a one-to-one udp connection it is the client with the ip-number on one of the sinks.

gst-launch-1.0 udpsrc port=5000 will give you a connection, but that is not going to be interpreted correctly. To do that we need to add the caps filter from the sender. Just copy paste it from the sender, there a many places to misspell the line. And remember, this is constructed from the pipeline you create when sending the stream so changes there requires a new caps filer at the udpsrc. Some formats requires more details in their caps filter than others. Mpeg is one of the more demanding.

gst-launch-1.0 udpsrc port=5000 caps = "application/x-rtp\,\ media\=\(string\)video\,\ clock-rate\=\(int\)90000\,\ encoding-name\=\(string\)MP4V-ES\,\ profile-level-id\=\(string\)1\,\ config\=\(string\)000001b001000001b58913000001000000012000c48d8800cd3204709443000001b24c61766335362e312e30\,\ payload\=\(int\)96\,\ ssrc\=\(uint\)2873740600\,\ timestamp-offset\=\(uint\)391825150\,\ seqnum-offset\=\(uint\)2980"

### Unpack the payload

This stream is still payloaded in rtp frames that needs to be unpackaed. This is handled by the right rtpdepayloader.

! rtpjpegdepay, ! rtpvp8depay and ! rtpmp4vdepay are the pads that can handle the unpacking.

The options for the depayloaders are quite slim. They mainly provide you with an option to display statistics. This pad provides a stream of the encoded data you sent earlier.

### Decode the format

The decoding of the format is also mostly quite straight forward.  
add the pads ! jpegdec, ! vp8dec or ! avdec\_mpeg4 to your pipeline to decode the video.

The decoding pads has some properties to be set in order to optimize the stream for your needs. But nothing that needs setting to get a basic stream running.

Display the video  
From the decoding step you now have the raw video from your initial source. Because all the encoding/decoding schemes are lossy to some extent you are left with a stream that is to some extent distorted from your source. But how much distorted and in what way (resolution, framerate, size etc) has to be decided by the application you are using it for.

To the last step, add ! autovideosink again and it will be displayed on your screen.

### Complete commands for receiving

#### Mjpeg

gst-launch-1.0 udpsrc port=5000 ! application/x-rtp,encoding-name=JPEG,payload=26 ! rtpjpegdepay ! jpegdec ! autovideosink

#### VP8

gst-launch-1.0 udpsrc port=5100 caps="application/x-rtp, media=(string)video, clock-rate=(int)90000, encoding-name=(string)VP8-DRAFT-IETF-01, payload=(int)96, ssrc=(uint)2990747501, clock-base=(uint)275641083, seqnum-base=(uint)34810" ! rtpvp8depay ! vp8dec ! autovideosink

#### MPEG2

gst-launch-1.0 -v udpsrc port=5200 caps = "application/x-rtp\,\ media\=\(string\)video\,\ clock-rate\=\(int\)90000\,\ encoding-name\=\(string\)MP4V-ES\,\ profile-level-id\=\(string\)1\,\ config\=\(string\)000001b001000001b58913000001000000012000c48d8800cd3204709443000001b24c61766335362e312e30\,\ payload\=\(int\)96\,\ ssrc\=\(uint\)2873740600\,\ timestamp-offset\=\(uint\)391825150\,\ seqnum-offset\=\(uint\)2980" ! rtpmp4vdepay ! avdec\_mpeg4 ! autovideosink

## Gst-inspect

This is a versatile tool when developing a stream to get an overview of your installed pads and their properties.  
gst-inspect-1.0 lists all elements on your system that is reachable by gstreamer. This is a long list, 1300 lines on my system. Grepping their content for words you think is included in the element you are looking for. gst-inspect-1.0 | grep sink gives you a list of available sinks and gst-inspect-1.0 | grep src gives you the available sources.  
From that you can get more information of the properties of each element gst-inspect-1.0 fpsdisplaysink shows information on a special kind of video sink for example.