Using GStreamer to build real-time applications with Golang

Dan Jenkins
Nimble Ape / Everycast Labs

FOSDEM, Brussels, February 2024

A Little About Me

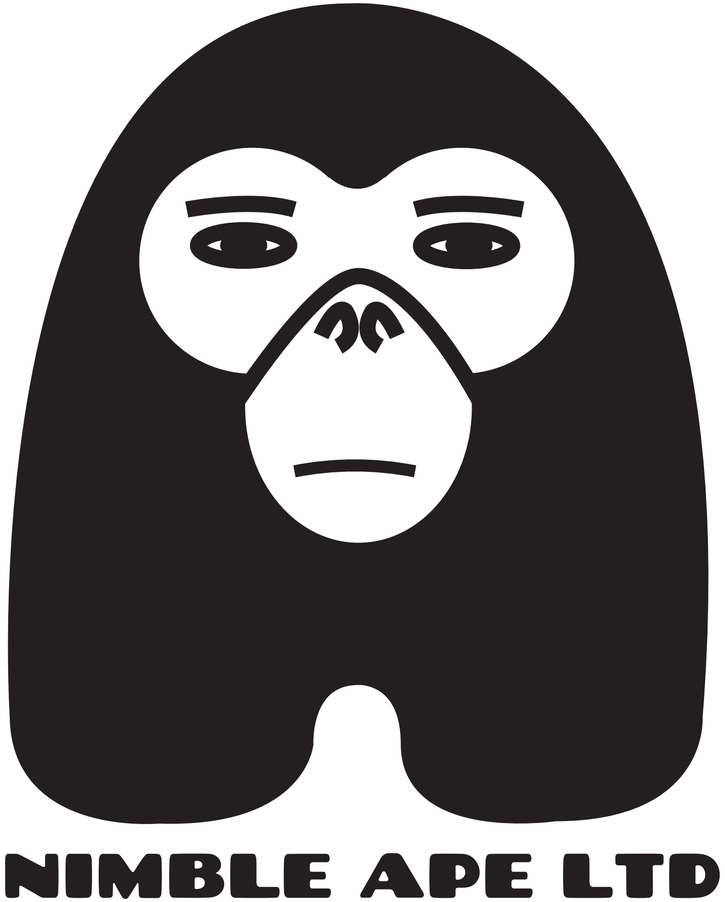
Dan Jenkins

- Founder of Everycast Labs, Nimble Ape
- Creator of CommCon
- Was the first "Google Developer Expert" in WebRTC
- Loves LEGO
- Loves Real-Time Media
- Loves developing for "the web"
- Over 10 years in Open Source VoIP

- @dan_jenkins / @danjenkins@fosstodon.org

Nimble Ape

- Real-Time Communication consultancy
- Based in the UK
- Work with Open Source Real-Time Comms
  - VoIP, WebRTC, Broadcast
- Got a problem you want help with? Let us know
- hello@nimblea.pe



Everycast Labs

- Creators of Broadcast Bridge (broadcastbridge.app)
  - A Platform as a Service for bringing in remote talent into production AV workflows
  - We work with WebRTC / SRT / NDI / Decklink & AJA cards.
- hello@everycastlabs.uk

CommCon

- "Open Media" - Real-Time or not
- Top quality production values
- 5 Years worth of content on our YouTube Channel
- commcon.xyz
- News about 2024 coming SOON! We'll be going on tour!

Using GStreamer to build real-time applications with Golang

We're going to talk about...

GStreamer

Golang

go-gst

Pion

https://gstreamer.freedesktop.org/

"open source multimedia framework"

You might know GStreamer as this...

```
gst-launch-1.0 rtspsrc location="rtsp://192.168.5.90/axis-media/media.amp?
videocodec=h264&resolution=1280x720&fps=25&videobitrate=4000&compression=50" is-
live=true latency=0 protocols=tcp ! rtph264depay ! video/x-h264,stream-
format=avc,alignment=au,profile=baseline ! h264parse config-interval=-1 ! queue
silent=true ! rtph264pay mtu=1400 config-interval=-1 ! application/x-
rtp,media=video,clock-rate=${$channels.video.clockRate},encoding-
name=${$channels.video.encodingName},ssrc=(uint)${$channels.video.SSRC} ! queue
silent=true ! udpsink host=127.0.0.1 port=${$pipeline.port} sync=false async=true
```

GStreamer is incredibly powerful

Ingress

Do Something

Egress

And GStreamer can do it all!

NDI

WebRTC

SRT

RTP

HLS/DASH

RTMP/RTSP

But GStreamer has a real super power.

AppSink

&

AppSrc

This is what we use in Broadcast Bridge

But we don't write C like this

```
/* Initialize GStreamer */
  gst_init (&argc, &argv);

  /* Build the pipeline */
  pipeline =
      gst_parse_launch
      ("playbin uri=https://gstreamer.freedesktop.org/data/media/sintel_trailer-
480p.webm",
      NULL);

  /* Start playing */
  gst_element_set_state (pipeline, GST_STATE_PLAYING);

  /* Wait until error or EOS */
  bus = gst_element_get_bus (pipeline);
  msg =
      gst_bus_timed_pop_filtered (bus, GST_CLOCK_TIME_NONE,
      GST_MESSAGE_ERROR | GST_MESSAGE_EOS);

  /* See next tutorial for proper error message handling/parsing */
  if (GST_MESSAGE_TYPE (msg) == GST_MESSAGE_ERROR) {
    g_error ("An error occurred! Re-run with the GST_DEBUG=*:WARN environment "
        "variable set for more details.");
  }
```

We write Go like this

```
gst.Init(&os.Args)

  // Let GStreamer create a pipeline from the parsed launch syntax on the cli.
  pipeline, err := gst.NewPipelineFromString(strings.Join(os.Args[1:], " "))
  if err != nil {
    return err
  }

  // Add a message handler to the pipeline bus, printing interesting information to
the console.
  pipeline.GetPipelineBus().AddWatch(func(msg *gst.Message) bool {
    switch msg.Type() {
    case gst.MessageEOS: // When end-of-stream is received stop the main loop
      pipeline.BlockSetState(gst.StateNull)
      mainLoop.Quit()
    case gst.MessageError: // Error messages are always fatal
      err := msg.ParseError()
      fmt.Println("ERROR:", err.Error())
      if debug := err.DebugString(); debug != "" {
        fmt.Println("DEBUG:", debug)
      }
      mainLoop.Quit()
    default:
      // All messages implement a Stringer. However, this is
      // typically an expensive thing to do and should be avoided.
      fmt.Println(msg)
    }
    return true
  })

  // Start the pipeline
  pipeline.SetState(gst.StatePlaying)
```

And that's because of the go-gst bindings

Originally created by

tinyzimmer

Now in their own GitHub Organisation

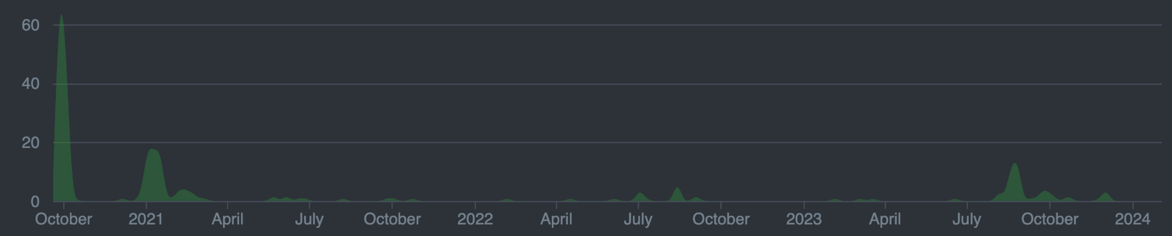
https://github.com/go-gst/go-gst

Under the new GitHub Org we have 3 main contributors

Sep 20, 2020 – Feb 3, 2024

Contributions: Commits

Contributions to main, excluding merge commits



Less forks =
great for everyone

Broadcast Bridge uses a mixture of SRT, NDI and WebRTC among others...

So why would we need to use AppSrc and AppSink?

Greater Control

For us, we use Pion to "do" WebRTC

https://github.com/pion

This means we're handling WebRTC in a language we know

Pion is hugely powerful

And easily upgradeable

Unlike
GStreamer

(for us and our team's skillset)

If we found a bug in "WebRTC" in GStreamer, getting it fixed and released might be a multi week/month process

With Pion handling WebRTC we're better in control.

We then use AppSrc to pass in the resulting RTP & RTCP

We let GStreamer do its thing...

And use AppSink to get the media back out...

Is it the most optimised way of doing it?

Absolutely not.

But it gives us huge flexibility.

Move fast.
Add new features.
Win business.

So let's take a look at AppSrc

appsrc

The appsrc element can be used by applications to insert data into a GStreamer pipeline. Unlike most GStreamer elements, Appsrc provides external API functions.

For the documentation of the API, please see the

libgstapp section in the GStreamer Plugins Base Libraries documentation.

Hierarchy

```
GObject
    └─GInitiallyUnowned
        └─GstObject
            └─GstElement
                └─GstBaseSrc
                    └─appsrc
```

Implemented interfaces

```
GstURIHandler
```

Factory details

Authors: – David Schleef , Wim Taymans

Classification: – `Generic/Source`

Rank – none

Plugin – app

Package – GStreamer Base Plug-ins

Pad Templates

`src`

```
ANY
```

Presence – *always*

Direction – *src*

Object type – GstPad

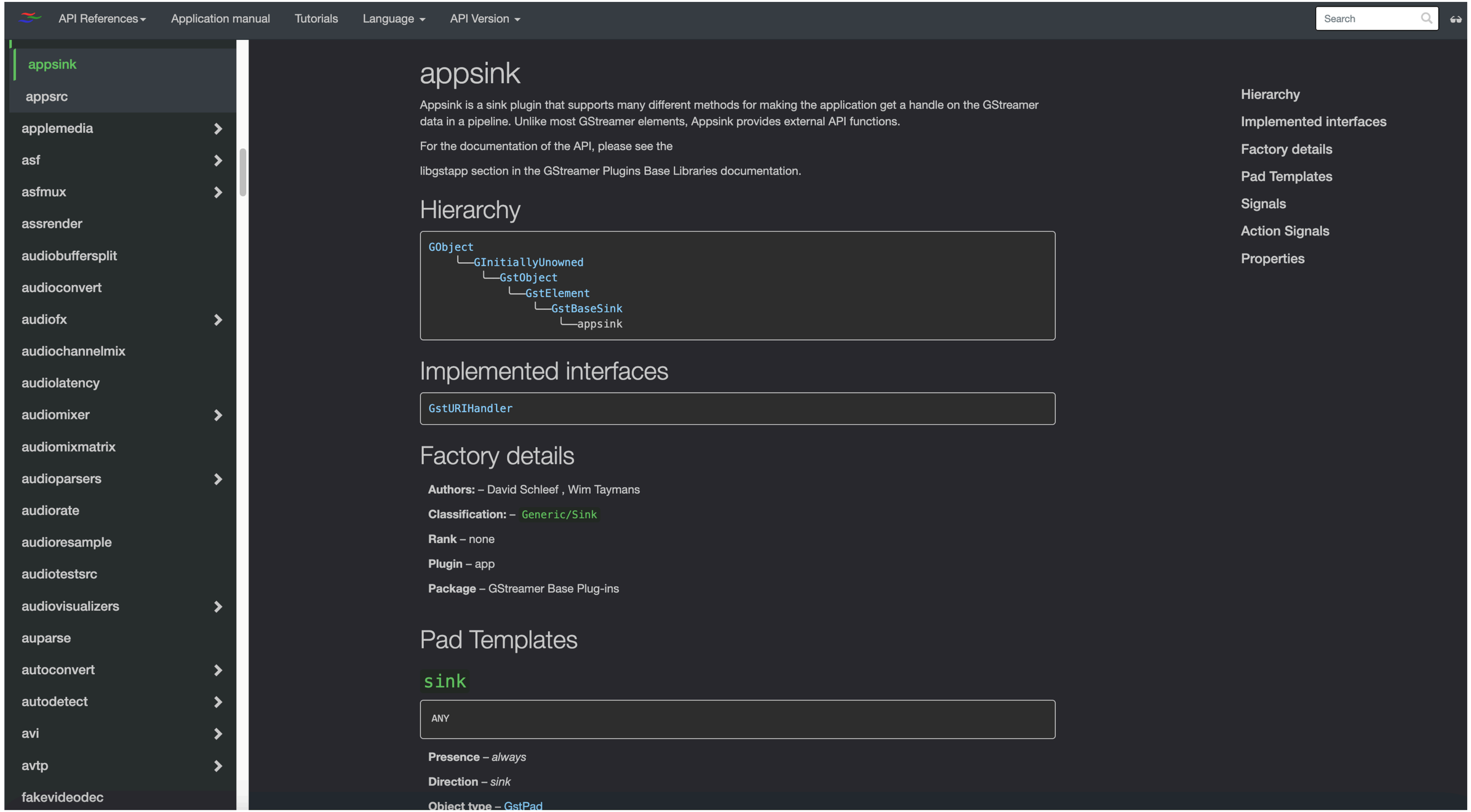
https://gstreamer.freedesktop.org/documentation/

GStreamer can ask you for data or you can just push it in

need-data signal

```
src.SetCallbacks(&app.SourceCallbacks{
    NeedDataFunc: func(self *app.Source, _ uint) {

      // If we've reached the end of the palette, end the stream.
      if i == len(palette) {
        src.EndStream()
        return
      }

      fmt.Println("Producing frame:", i)

      // Create a buffer that can hold exactly one video RGBA frame.
      buffer := gst.NewBufferWithSize(videoInfo.Size())

      // For each frame we produce, we set the timestamp when it should be displayed
      // The autovideosink will use this information to display the frame at the right time.
      buffer.SetPresentationTimestamp(gst.ClockTime(time.Duration(i) * 500 *
time.Millisecond))

      // Produce an image frame for this iteration.
      pixels := produceImageFrame(palette[i])
      buffer.Map(gst.MapWrite).WriteData(pixels)
      buffer.Unmap()

      // Push the buffer onto the pipeline.
      self.PushBuffer(buffer)

      i++
    },
  })
```

And AppSink is no different



https://gstreamer.freedesktop.org/documentation/

You get pushed your data

new-sample signal

```
sink.SetCallbacks(&app.SinkCallbacks{
    // Add a "new-sample" callback
    NewSampleFunc: func(sink *app.Sink) gst.FlowReturn {

      // Pull the sample that triggered this callback
      sample := sink.PullSample()
      if sample == nil {
        return gst.FlowEOS
      }

      // Retrieve the buffer from the sample
      buffer := sample.GetBuffer()
      if buffer == nil {
        return gst.FlowError
      }

      samples := buffer.Map(gst.MapRead).AsInt16LESlice()
      defer buffer.Unmap()

      // Calculate the root mean square for the buffer
      // (https://en.wikipedia.org/wiki/Root_mean_square)
      var square float64
      for _, i := range samples {
        square += float64(i * i)
      }
      rms := math.Sqrt(square / float64(len(samples)))
      fmt.Println("rms:", rms)

      return gst.FlowOK
    },
  })
```

For us, we need to handle RTP & RTCP

But GStreamer makes that easy

RTPBin

rtpbin

RTP bin combines the functions of rtpsession, rtpssrcdemux, rtpjitterbuffer and rtpptdemux in one element. It allows for multiple RTP sessions that will be synchronized together using RTCP SR packets.

rtpbin is configured with a number of request pads that define the functionality that is activated, similar to the rtpsession element.

To use rtpbin as an RTP receiver, request a recv_rtp_sink_%u pad. The session number must be specified in the pad name. Data received on the recv_rtp_sink_%u pad will be processed in the rtpsession manager and after being validated forwarded on rtpssrcdemux element. Each RTP stream is demuxed based on the SSRC and send to a rtpjitterbuffer. After the packets are released from the jitterbuffer, they will be forwarded to a rtpptdemux element. The rtpptdemux element will demux the packets based on the payload type and will create a unique pad recv_rtp_src_%u_%u_%u on rtpbin with the session number, SSRC and payload type respectively as the pad name.

To also use rtpbin as an RTCP receiver, request a recv_rtcp_sink_%u pad. The session number must be specified in the pad name.

If you want the session manager to generate and send RTCP packets, request the send_rtcp_src_%u pad with the session number in the pad name. Packet pushed on this pad contain SR/RR RTCP reports that should be sent to all participants in the session.

To use rtpbin as a sender, request a send_rtp_sink_%u pad, which will automatically create a send_rtp_src_%u pad. If the session number is not provided, the pad from the lowest available session will be returned. The session manager will modify the SSRC in the RTP packets to its own SSRC and will forward the packets on the send_rtp_src_%u pad after updating its internal state.

The session manager needs the clock-rate of the payload types it is handling and will signal the request-pt-map signal when it needs such a mapping. One can clear the cached values with the clear-pt-map signal.

Access to the internal statistics of rtpbin is provided with the get-internal-session property. This action signal gives access to the RTPSession object which further provides action signals to retrieve the internal source and other sources.

rtpbin also has signals (#GstRtpBin::request-rtp-encoder, request-rtp-decoder, request-rtcp-encoder and request-rtp-decoder) to dynamically request for RTP and RTCP encoders and decoders in order to support SRTP. The encoders must provide the pads rtp_sink_%u and rtp_src_%u for RTP and rtcp_sink_%u and rtcp_src_%u for RTCP. The session number will be used in the pad name. The decoders must provide rtp_sink and rtp_src for RTP and rtcp_sink and rtcp_src for RTCP. The decoders will be placed before the rtpsession element, thus they must support SSRC demuxing internally.

rtpbin has signals (#GstRtpBin::request-aux-sender and request-aux-receiver to dynamically request an element that can be used to create or merge additional RTP streams. AUX elements are needed to implement FEC or retransmission (such as RFC 4588). An AUX sender must have one sink_%u pad that matches the sessionid in the signal and it should have 1 or more src_%u pads. For each src_%\u pad, a session will be made (if needed) and the pad will be linked to the session send_rtp_sink pad. Each session will then expose its source pad as send_rtp_src_%u on rtpbin. An AUX receiver has 1 src_%u pad that much match the sessionid in the signal and 1 or more sink_%u pads. A session will be made for each sink_%u pad when the corresponding recv_rtp_sink_%u pad is requested on rtpbin. The request-jitterbuffer signal can be used to provide a custom element to perform arrival time smoothing, reordering and optionally packet loss detection and retransmission requests.

https://gstreamer.freedesktop.org/documentation/

RTPBin implements everything you need to handle RTP & RTCP

jitter buffer, ssrc demuxer, payload type demuxer, rtcp, rtp depayloading

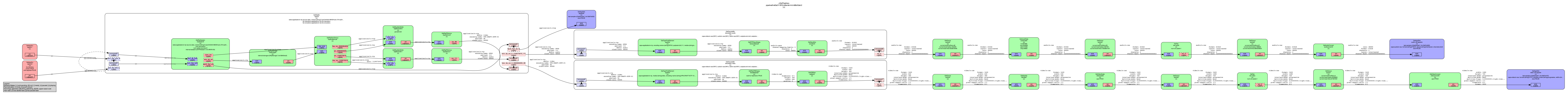
Connect the AppSrc sink Pad(s) to RTPBin src pads

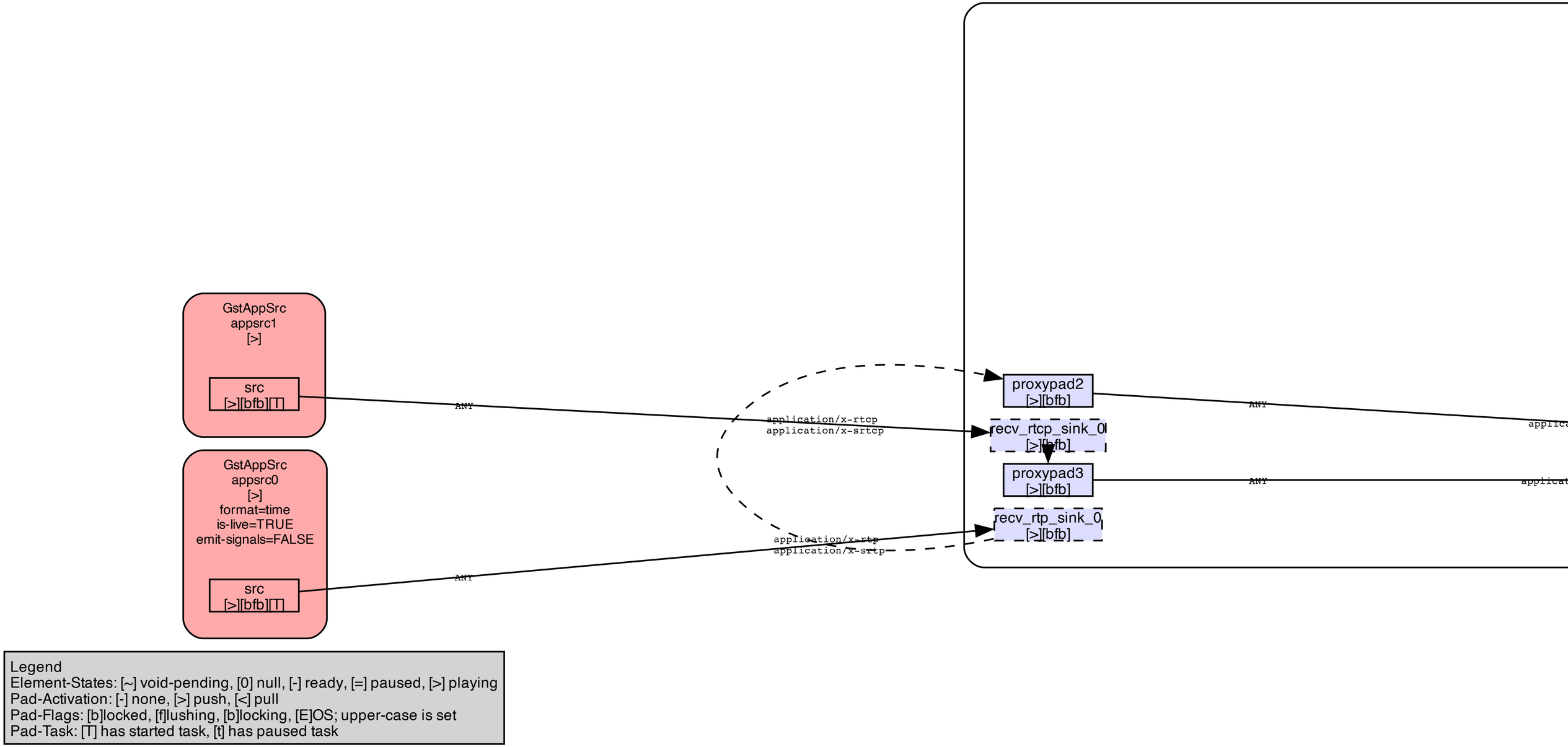
```
rtcpSinkPad := rtpbin.GetRequestPad("recv_rtcp_sink_%u")
rtcpSrcPad := rtpbin.GetRequestPad("send_rtcp_src_%u")

rtcpAppSink, err := app.NewAppSink()
if err != nil {
  log.Warnf("Error creating Gstreamer app sink %s", err)
  return
}
rtcpAppsrc, err := app.NewAppSrc()
if err != nil {
  log.Warnf("Error creating Gstreamer app src %s", err)
  return
}

err = pipeline.Add(rtcpAppsrc.Element)
if err != nil {
  log.Warnf("Error adding rtcp src to pipeline, %s", err)
  return
}
err = pipeline.Add(rtcpAppSink.Element)
if err != nil {
  log.Warnf("Error adding rtcp src to pipeline, %s", err)
  return
}

linkedRTCPSrc := rtcpAppsrc.GetStaticPad("src").Link(rtcpSinkPad)
if linkedRTCPSrc.String() != "ok" {
  log.Warnf("Error connecting RTCP sink to rtpbin, %s", err)
  return
}
```

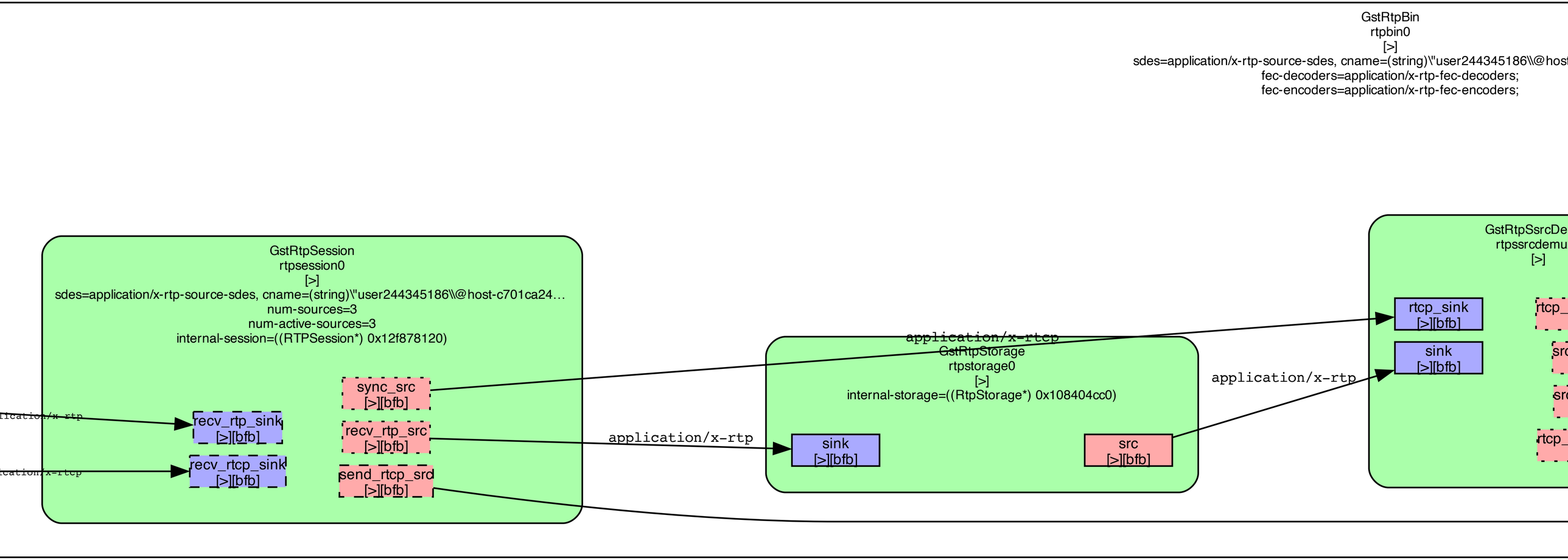
And you'll get RTP in, RTCP in and out!

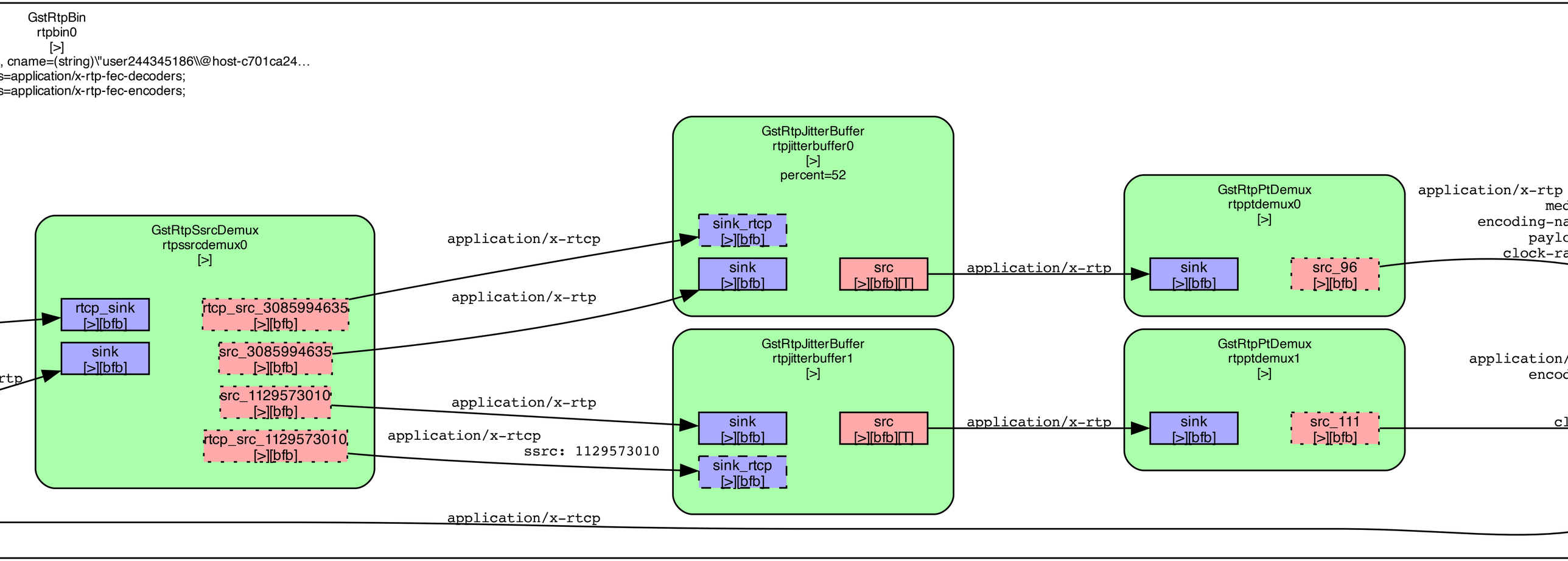
Don't forget about the RTCP!

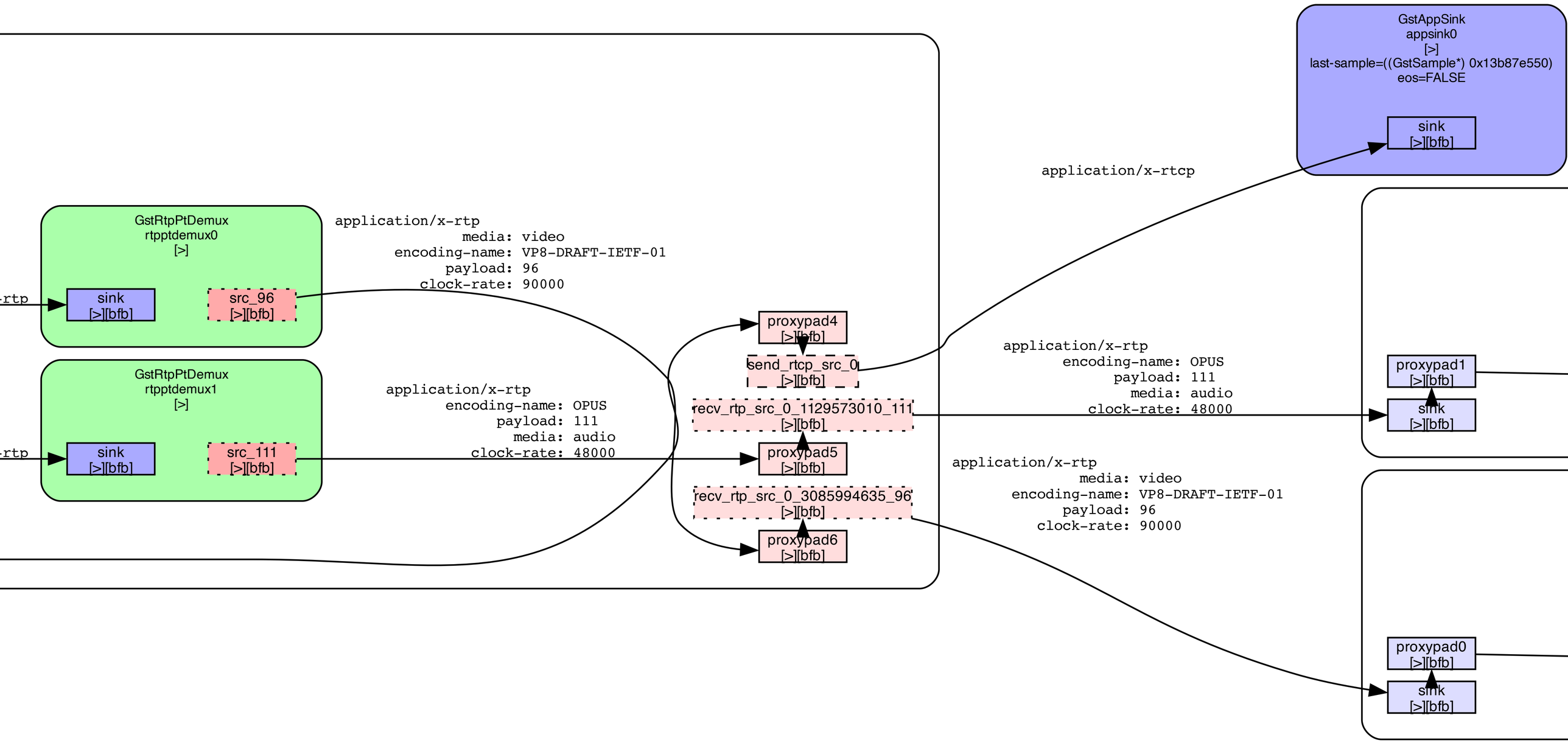
And you end up with something looking like this

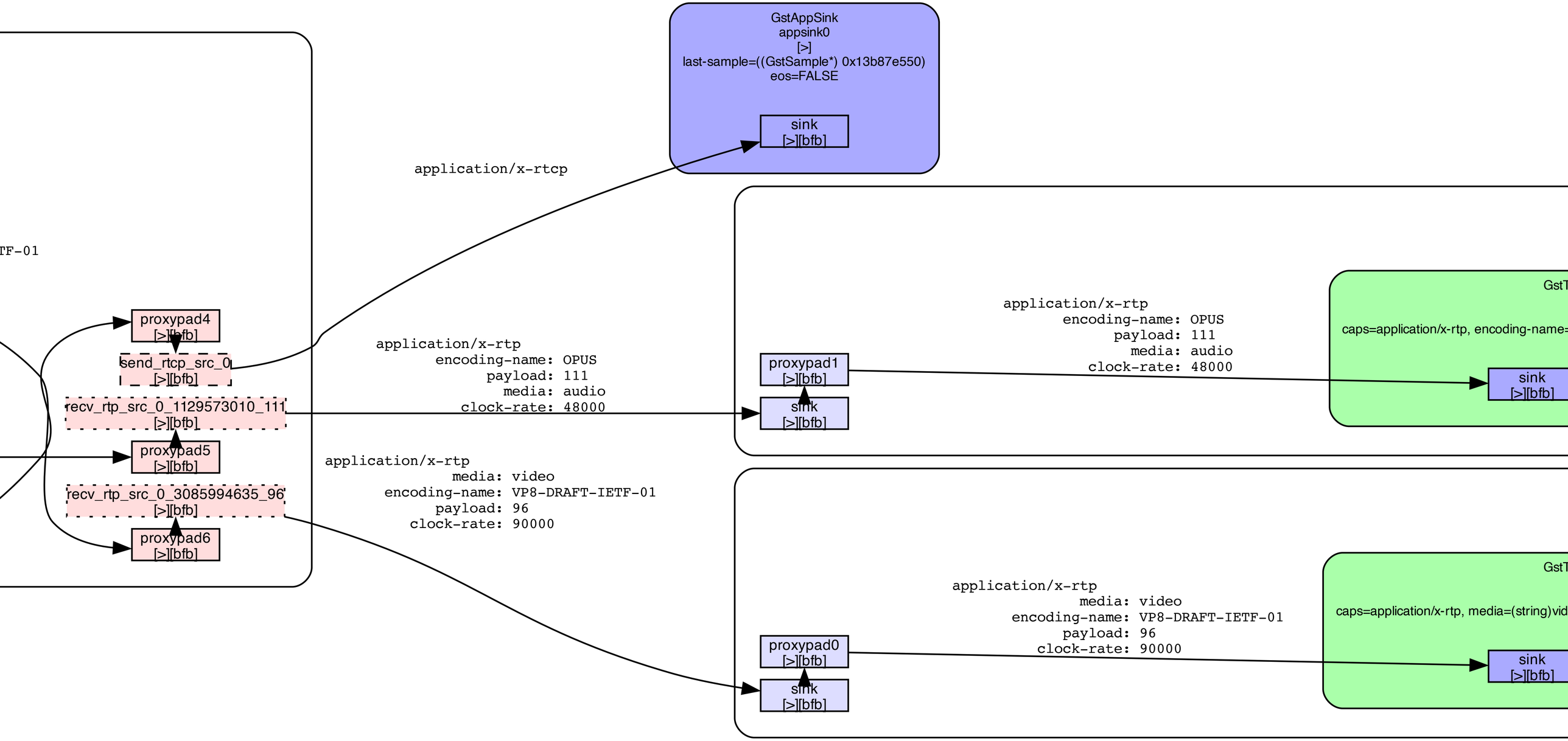

<GstPipeline>
pipeline81e63a7f-7870-446e-8e14-b1d89cf0afc2
[>]
GstDecodeBin
decodebin-audio
[>]
caps=video/x-raw(ANY); audio/x-raw(ANY); text/x-raw(ANY); subpicture/x-dvd; subpictur…
application/x-rtp
encoding-name: OPUS
payload: 111
media: audio
clock-rate: 48000
GstRTPOpusDepay
rtpopusdepay0
[>]
sink
[>][bfb]
src
[>][bfb]
audio/x-opus
channel-mapping-family: 0
channels: 2
rate: 48000
GstOpusDec
opusdec0
[>]
sink
[>][bfb]
src
[>][bfb]
audio/x-raw
format: S16LE
layout: interleaved
rate: 48000
channels: 2
channel-mask: 0x0000000000000003

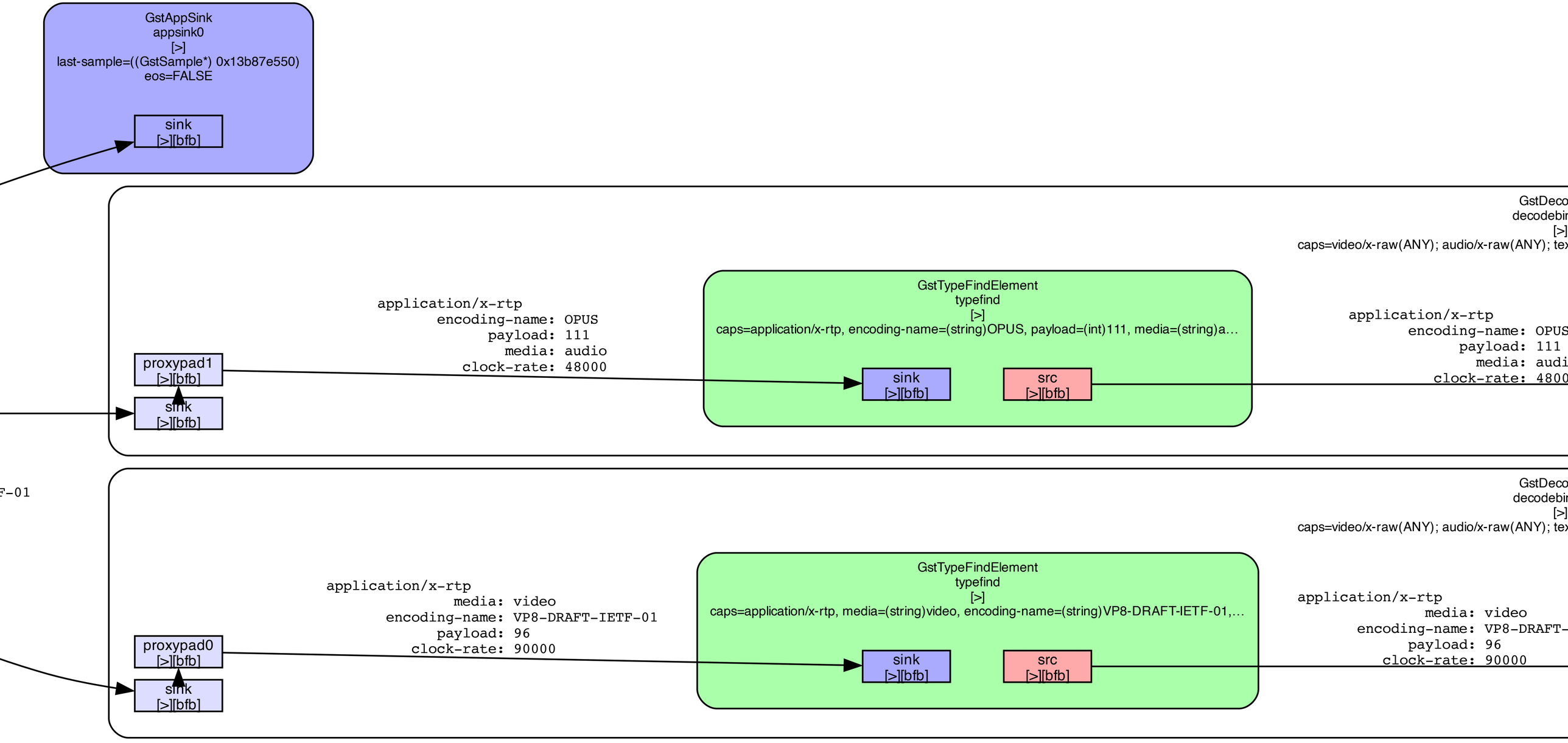

GstAppSrc
appsrc1
[>]
src
[>][bfb][T]
ANY
GstAppSrc
appsrc0
[>]
format=time
is-live=TRUE
emit-signals=FALSE
src
[>][bfb][T]
ANY
application/x-rtcp
application/x-srtcp
application/x-rtp
application/x-srtp
proxypad2
[>][bfb]
recv_rtcp_sink_0
[>][bfb]
proxypad3
[>][bfb]
recv_rtp_sink_0
[>][bfb]
ANY
ANY
Legend
Element-States: [~] void-pending, [0] null, [-] ready, [=] paused, [>] playing
Pad-Activation: [-] none, [>] push, [<] pull
Pad-Flags: [b]locked, [f]lushing, [b]locking, [E]OS; upper-case is set
Pad-Task: [T] has started task, [t] has paused task

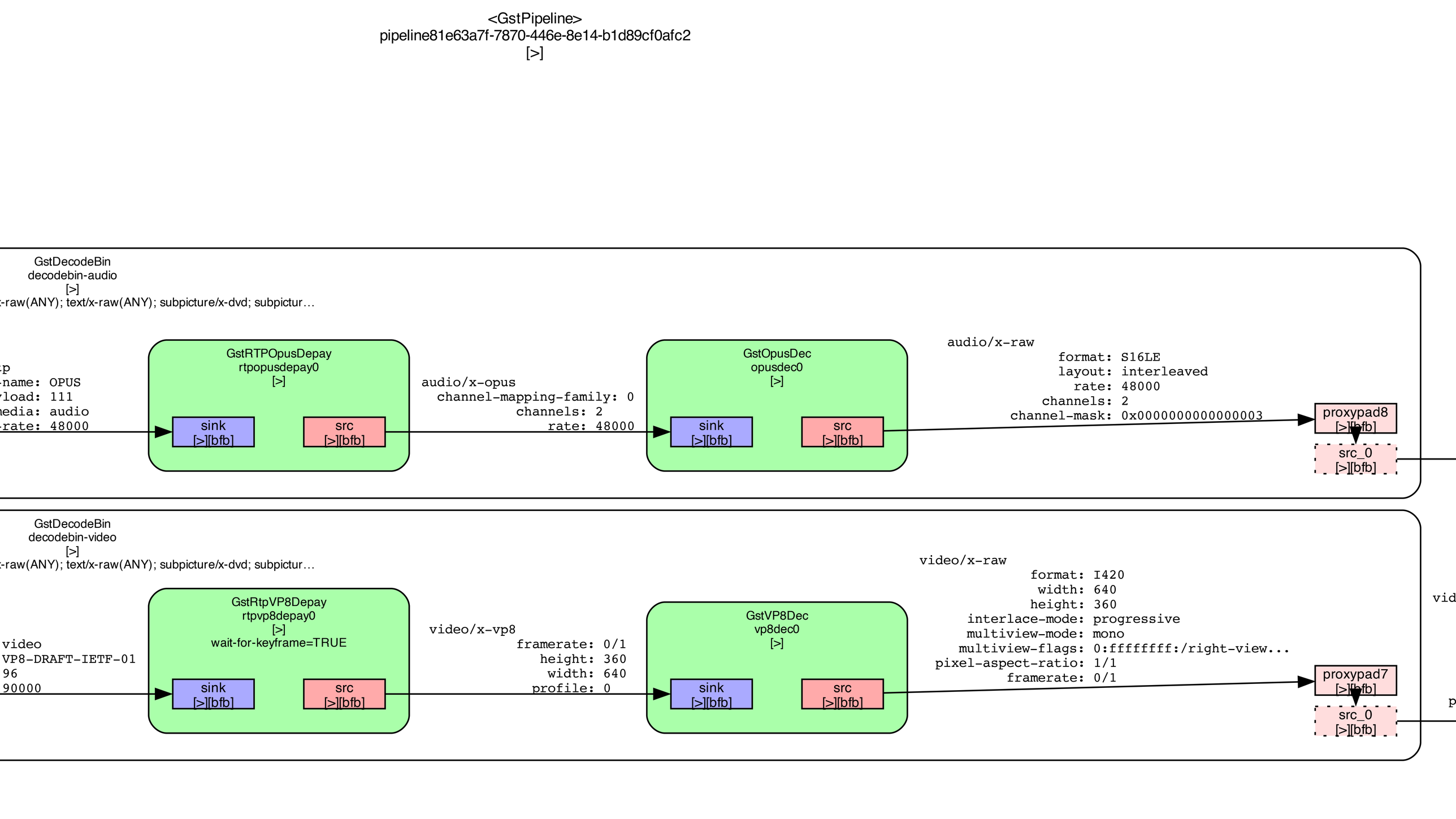

GstRtpBin
rtpbin0
[>]
sdes=application/x-rtp-source-sdes, cname=(string)\"user244345186\\@host
fec-decoders=application/x-rtp-fec-decoders;
fec-encoders=application/x-rtp-fec-encoders;
GstRtpSession
rtpsession0
[>]
sdes=application/x-rtp-source-sdes, cname=(string)\"user244345186\\@host-c701ca24…
num-sources=3
num-active-sources=3
internal-session=((RTPSession*) 0x12f878120)
sync_src
[>][bfb]
recv_rtp_sink
[>][bfb]
recv_rtp_src
[>][bfb]
recv_rtcp_sink
[>][bfb]
send_rtcp_src
[>][bfb]
application/x-rtcp
GstRtpStorage
rtpstorage0
[>]
internal-storage=((RtpStorage*) 0x108404cc0)
application/x-rtp
sink
[>][bfb]
src
[>][bfb]
application/x-rtp
GstRtpSsrcDe
rtpssrcdemu
[>]
rtcp_sink
[>][bfb]
sink
[>][bfb]


GstRtpBin
rtpbin0
[>]
, cname=(string)\"user244345186\\@host-c701ca24…
s=application/x-rtp-fec-decoders;
s=application/x-rtp-fec-encoders;
GstRtpSsrcDemux
rtpssrcdemux0
[>]
rtcp_sink
[>][bfb]
sink
[>][bfb]
rtcp_src_3085994635
[>][bfb]
src_3085994635
[>][bfb]
src_1129573010
[>][bfb]
rtcp_src_1129573010
[>][bfb]
application/x-rtcp
application/x-rtp
application/x-rtp
application/x-rtcp
ssrc: 1129573010
GstRtpJitterBuffer
rtpjitterbuffer0
[>]
percent=52
sink_rtcp
[>][bfb]
sink
[>][bfb]
src
[>][bfb][T]
GstRtpJitterBuffer
rtpjitterbuffer1
[>]
sink
[>][bfb]
src
[>][bfb][T]
sink_rtcp
[>][bfb]
application/x-rtp
application/x-rtp
GstRtpPtDemux
rtpptdemux0
[>]
sink
[>][bfb]
src_96
[>][bfb]
GstRtpPtDemux
rtpptdemux1
[>]
sink
[>][bfb]
src_111
[>][bfb]
application/x-rtp
application/


GstAppSink
appsink0
[>]
last-sample=((GstSample*) 0x13b87e550)
eos=FALSE
sink
[>][bfb]
application/x-rtcp
GstRtpPtDemux
rtpptdemux0
[>]
-rtp
sink
[>][bfb]
src_96
[>][bfb]
application/x-rtp
media: video
encoding-name: VP8-DRAFT-IETF-01
payload: 96
clock-rate: 90000
GstRtpPtDemux
rtpptdemux1
[>]
-rtp
sink
[>][bfb]
src_111
[>][bfb]
application/x-rtp
encoding-name: OPUS
payload: 111
media: audio
clock-rate: 48000
proxypad4
[>][bfb]
send_rtcp_src_0
[>][bfb]
recv_rtp_src_0_1129573010_111
[>][bfb]
proxypad5
[>][bfb]
recv_rtp_src_0_3085994635_96
[>][bfb]
proxypad6
[>][bfb]
application/x-rtp
encoding-name: OPUS
payload: 111
media: audio
clock-rate: 48000
proxypad1
[>][bfb]
sink
[>][bfb]
application/x-rtp
media: video
encoding-name: VP8-DRAFT-IETF-01
payload: 96
clock-rate: 90000
proxypad0
[>][bfb]
sink
[>][bfb]


GstAppSink
appsink0
[>]
last-sample=((GstSample*) 0x13b87e550)
eos=FALSE
sink
[>][bfb]
application/x-rtcp
TF-01
proxypad4
[>][bfb]
send_rtcp_src_0
[>][bfb]
recv_rtp_src_0_1129573010_111
[>][bfb]
proxypad5
[>][bfb]
recv_rtp_src_0_3085994635_96
[>][bfb]
proxypad6
[>][bfb]
application/x-rtp
encoding-name: OPUS
payload: 111
media: audio
clock-rate: 48000
application/x-rtp
media: video
encoding-name: VP8-DRAFT-IETF-01
payload: 96
clock-rate: 90000
proxypad1
[>][bfb]
sink
[>][bfb]
application/x-rtp
encoding-name: OPUS
payload: 111
media: audio
clock-rate: 48000
GstT
caps=application/x-rtp, encoding-name=
sink
[>][bfb]
proxypad0
[>][bfb]
sink
[>][bfb]
application/x-rtp
media: video
encoding-name: VP8-DRAFT-IETF-01
payload: 96
clock-rate: 90000
GstT
caps=application/x-rtp, media=(string)vid
sink
[>][bfb]


GstAppSink
appsink0
[>]
last-sample=((GstSample*) 0x13b87e550)
eos=FALSE
sink
[>][bfb]
GstDeco
decodebi
[>]
caps=video/x-raw(ANY); audio/x-raw(ANY); te
application/x-rtp
encoding-name: OPUS
payload: 111
media: audio
clock-rate: 48000
GstTypeFindElement
typefind
[>]
caps=application/x-rtp, encoding-name=(string)OPUS, payload=(int)111, media=(string)a…
sink
[>][bfb]
src
[>][bfb]
application/x-rtp
encoding-name: OPUS
payload: 111
media: audi
clock-rate: 4800
proxypad1
[>][bfb]
sink
[>][bfb]
F-01
GstDeco
decodebi
[>]
caps=video/x-raw(ANY); audio/x-raw(ANY); te
application/x-rtp
media: video
encoding-name: VP8-DRAFT-IETF-01
payload: 96
clock-rate: 90000
GstTypeFindElement
typefind
[>]
caps=application/x-rtp, media=(string)video, encoding-name=(string)VP8-DRAFT-IETF-01,…
sink
[>][bfb]
src
[>][bfb]
application/x-rtp
media: video
encoding-name: VP8-DRAFT-
payload: 96
clock-rate: 90000
proxypad0
[>][bfb]
sink
[>][bfb]


<GstPipeline>
pipeline81e63a7f-7870-446e-8e14-b1d89cf0afc2
[>]
GstDecodeBin
decodebin-audio
[>]
x-raw(ANY); text/x-raw(ANY); subpicture/x-dvd; subpictur…
GstRTPOpusDepay
rtpopusdepay0
[>]
sink
[>][bfb]
src
[>][bfb]
audio/x-opus
channel-mapping-family: 0
channels: 2
rate: 48000
GstOpusDec
opusdec0
[>]
sink
[>][bfb]
src
[>][bfb]
audio/x-raw
format: S16LE
layout: interleaved
rate: 48000
channels: 2
channel-mask: 0x0000000000000003
proxypad8
[>][bfb]
src_0
[>][bfb]
GstDecodeBin
decodebin-video
[>]
x-raw(ANY); text/x-raw(ANY); subpicture/x-dvd; subpictur…
GstRtpVP8Depay
rtpvp8depay0
[>]
wait-for-keyframe=TRUE
sink
[>][bfb]
src
[>][bfb]
video/x-vp8
framerate: 0/1
height: 360
width: 640
profile: 0
GstVP8Dec
vp8dec0
[>]
sink
[>][bfb]
src
[>][bfb]
video/x-raw
format: I420
width: 640
height: 360
interlace-mode: progressive
multiview-mode: mono
multiview-flags: 0:ffffffff:/right-view...
pixel-aspect-ratio: 1/1
framerate: 0/1
proxypad7
[>][bfb]
src_0
[>][bfb]

We're using Go purely because of Pion

Pion gives us control.

WebRTC in Pure Golang

But you can do this with any of the GStreamer bindings

gstreamer

open source multimedia framework

- Home
- Features
- News
- Annual Conference
- Planet (Blogs)
- Download
- Applications

- Security Center
- GitLab
- Developers
- Documentation
- Mailing Lists
- Forum
- File a Bug

- Artwork
- @gstreamer on Twitter
- @gstreamer on Mastodon
- #gstreamer on Matrix

Bindings

There are bindings for GStreamer for quite a few languages already.

Here's a quick overview of all of our bindings :

language	status
Python	Released. See the gst-python module.
Perl	Released, See the CPAN module GStreamer1.
Rust	See crates.io.
D	Released as part of the GtkD project.
.NET	The bindings are still under development. See the gstreamer-sharp module.
C++	Released. See the gstreamermm GNOME module.
Qt	Unmaintained. See the qt-gstreamer module.
Guile	Released under the aegis of the guile-gnome project.
Haskell	See Haskell.org.
Java	Released as part of the gstreamer-java project.
Ruby	The Ruby bindings are released as part of Ruby-GNOME2.
Vala	Released as part of Vala.

Go isn't on the list! (yet)

https://gitlab.freedesktop.org/gstreamer/www/-/merge_requests/92

Got a problem and no plugin available?

Build it yourself with AppSrc and AppSink!

So...
Why
GStreamer?

Why not FFmpeg?

GStreamer

does everything we need.

It has a great community

A Super friendly community

Gstreamer is super flexible and easier for us to work with

GStreamer FTW

Don't wait for others.

Build with GStreamer and AppSrc and AppSink

Thanks for Having Me!

Thanks!

nimblea.pe

everycastlabs.uk

broadcastbridge.app

commcon.xyz

@dan_jenkins

@danjenkins@fosstodon.org

@nimbleape@nimblea.pe

@everycastlabs@everycastlabs.uk

@commcon@commcon.xyz