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...
Golang
Pion
"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
And GStreamer can do it all!
WebRTC
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
     NULL);

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

/* Wait until error or EOS */
bos = 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.
pipe, 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**
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
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 Schiewe, 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
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.SetParentPresentationTimestamp(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
appsink

Appsink is a sink plugin that supports many different methods for making the application get a handle on the GStreamer data in a pipeline. Unlike most GStreamer elements, Appsink provides external API functions.

For the documentation of the API, please see the 
libgstapp section in the GStreamer Plugins Base Libraries documentation.

Hierarchy

Implemented interfaces

Factory details

Pad Templates

sink

https://gstreamer.freedesktop.org/documentation/
You get pushed your data
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
    },
})

new-sample
signal
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 to the 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 rtpssrcdemux element. The rtpssrcdemux element will demux the packets based on the payload type and will create a unique pad recv RTP src.%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 RTP 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 RTP src.%u pad with the session number in the pad name. Packet pushed on this pad contain SRRR 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 (GetRtpBinRequestRtpEncoder, request RTP decoder, request RTP decoder, request RTP 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 RTP sink.%u and RTP src.%u for RTCP. The session number will be used in the pad name. The decoders must provide RTP sink and RTP sink and RTP src for RTP. The decoders will be placed before the rtpsession element, thus they must support SSRC demuxing internally.

rtpbin has signals (GetRtpBinRequestAuxSender and request RTP 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 srccc_%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 must 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**
rtcpSink := rtpbin.GetRequestPad("recv_rtcp_sink_%u")
rtcpSrc := 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
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
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