asymptotic

The evolution of HTTP based Signaling for WebRTC in GStreamer

Taruntej Kanakamalla

This talk is about ...

- Brief about WebRTC and Signaling
- WebRTC plugins in GStreamer
- Introduction to the standards WHIP and WHEP
- Initial version of plugins for WHIP and WHEP clients
- Signallable in the `rswebrtc` plugin
  - Adaptation of WHIP Client and WHIP Server
- Future plans for WHIP and WHEP in GStreamer

About me

- Consultant - Open Source software
- Asymptotic Inc. helps customers build Multimedia solutions
- GStreamer, PulseAudio, FreeSWITCH etc
- Experience involving low level firmware to user level applications
- From Hyderabad, India

What is WebRTC?

- Real Time Communication for Web
- Peer-Peer Exchange of Audio/Video/Data
- Mandatory encryption – DTLS and SRTP
- Inbuilt support in all modern browsers
- Quick and easy with W3C PeerConnection API
- Different libraries available for native apps

What is Signaling?

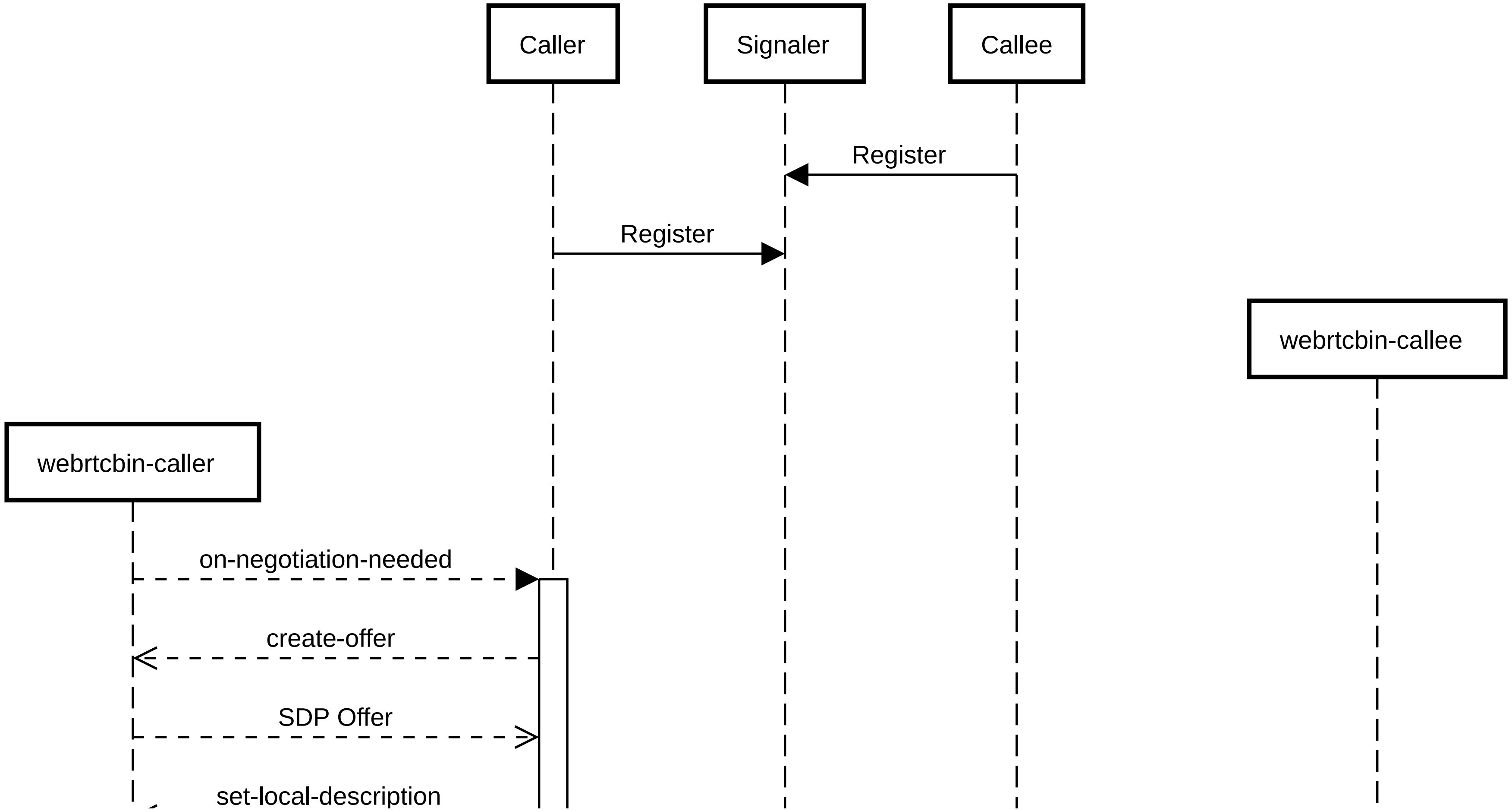
- A process of exchanging control information between two devices
  - Media codecs, Channels, Formats etc. (SDP)
  - Connectivity details (ICE candidates)
- WebRTC does not mandate any standard protocol
- Can use WebSockets, gRPC, HTTP etc
- Uses SDP O/A (Offer/Answer) model

asymptotic

GstWebRTC

- `webrtcbin` - GStreamer's implementation
- Authored by Centricular Ltd; First merged in 2017
- Built for native apps, servers etc
- Uses libnice, SRTP, DTLS and RTP plugins
- Written to be inline with the W3C PeerConnection API

Signaling


Caller
Signaler
Callee
Register
Register
webrtcbin-callee
webrtcbin-caller
on-negotiation-needed
create-offer
SDP Offer
set-local-description

HTTP for Signaling

- Lack of standardized signaling in WebRTC
    - Can’t use as a plug-n-play solution
    - Obstacle for adoption in broadcasting and streaming industry
- Standards WHIP and WHEP aim to fill this gap

What is WHIP?

- WebRTC-HTTP Ingestion protocol (WHIP)
    - WebRTC producer → HTTP Endpoint → Media Server
    - to ingest (push) a stream to media server
    - SendOnly

WHIP Session

https://www.ietf.org/archive/id/draft-ietf-wish-whip-09.html

```
+-------------+    +---------------+ +--------------+ +---------------+
| WHIP client |    | WHIP endpoint | | Media Server | | WHIP resource |
+--+----------+    +---------+-----+ +------+-------+ +--------|------+
   |                         |              |                  |
   |                         |              |                  |
   |HTTP POST (SDP Offer)    |              |                  |
   +------------------------>+              |                  |
   |201 Created (SDP answer) |              |                  |
   +<------------------------+              |                  |
   |          ICE REQUEST                   |                  |
   +--------------------------------------->+                  |
   |          ICE RESPONSE                  |                  |
   |<---------------------------------------+                  |
   |          DTLS SETUP                    |                  |
   |<======================================>|                  |
   |          RTP/RTCP FLOW                 |                  |
   +<-------------------------------------->+                  |
   | HTTP DELETE                                               |
   +---------------------------------------------------------->+
   | 200 OK                                                    |
   <-----------------------------------------------------------x
```

WHIP API

asymptotic

- `POST /endpoint/`
  - Request Body : SDP Offer
  - Response Body : SDP Answer
  - Response Headers :
    - Location: Resource URL
    - Link: STUN/TURN servers

- `PATCH /resource/id`
  - ICE Restart
  - ICE Trickle

- `DELETE /resource/id`
  - Teardown

What is WHEP?

- WebRTC-HTTP Egress Protocol (WHEP)
  - WebRTC consumer → HTTP Endpoint → Media Server
  - to consume a stream from a media server
  - RecvOnly

asymptotic

WHEP Session

https://www.ietf.org/id/draft-murillo-whep-02.html

```
+-------------+    +---------------+ +--------------+ +---------------+
| WHEP Player |    | WHEP Endpoint | | Media Server | | WHEP Resource |
+--+----------+    +---------+-----+ +------+-------+ +--------|------+
   |                         |              |                  |
   |                         |              |                  |
   |HTTP POST (SDP Offer)    |              |                  |
   +------------------------>+              |                  |
   |201 Created (SDP answer) |              |                  |
   +<------------------------+              |                  |
   |          ICE REQUEST                   |                  |
   +--------------------------------------->+                  |
   |          ICE RESPONSE                  |                  |
   |<---------------------------------------+                  |
   |          DTLS SETUP                    |                  |
   |<======================================>|                  |
   |          RTP/RTCP FLOW                 |                  |
   +<-------------------------------------->+                  |
   | HTTP DELETE                                               |
   +---------------------------------------------------------->+
   | 200 OK                                                    |
   <-----------------------------------------------------------x
```

WHEP API

asymptotic

- `POST /endpoint`
  - Request Body : SDP Offer
  - Response Body : SDP Answer
  - Response Headers :
    - Location: Resource URL
    - Link: STUN/TURN servers

- `PATCH /resource/id`
  - ICE Trickle
  - ICE Restart

- `DELETE /resource/id`
  - Teardown

WHEP extensions

- Server Sent Events
  - server-to-client communication using WHATWG server sent events
  - active: indicating that there is an active publication
  - inactive: indicating that there is no active publication
  - layers: the video layers being published for the resource
  - viewercount: the number of viewers currently connected
  - WHEP player can request to create server-to-client event stream
- Video Layer Selection
  - Allows WHEP Player to request a desired video layer or rendition
  - `{ "encodingId": "1", "simulcastIdx": 1, "width": 640, "height": 360, "spatialLayerId": 0, "temporalLayerId": 1,"bitrate": 557112 }`
  - In cases SVC (scalable video codecs) and simulcast are supported by the server

WHEP extension API

- `POST /resource/id/sse`
  - Request: Events List
  - Response: 200 OK
  - Response Header:
    - Location: sse url

- `POST /resource/id/layer`
  - Request : Desired video layer
  - Response: 200 OK

webrtchttp plugin

- Consists of two elements
  - `whipsink` - WHIP Client
  - `whepsrc` - WHEP Client
- Client side implementations of WHIP and WHEP
- Wrappers around `webrtcbin` element
- Simple and transparent HTTP clients
- Do not bother about encoding and RTP payloading
- Written in Rust language

webrtchttp plugin

- Tested against various media server implementations
  - Cloudflare
  - Dolby IO
  - Janus
  - MediaMTX
  - Live777

whipsink

- WebRTC producer i.e., SendOnly
- accepts an RTP encoded stream from upstream

```
Pad Templates:
  SINK template: 'sink_%u'
  Availability: On request
  Capabilities: application/x-rtp
```

Example Pipeline:

```
gst-launch-1.0 whipsink name=whip auth-token=$WHIP_TOKEN whip-endpoint=$WHIP_ENDPOINT \
videotestsrc ! videoconvert ! openh264enc ! rtph264pay ! whip.sink_0 \
audiotestsrc ! audioconvert ! opusenc ! rtpopuspay ! whip.sink_1
```

whipsink

Create offer and set local description:

```
self.webrtcbin.connect("on-negotiation-needed", false, {
...
  // define a promise which returns after offer is created
  let promise = gst::Promise::with_change_func(move |reply| {
    let offer_sdp = match reply {
...
      }
...
    }}
    self.webrtcbin.emit_by_name::<()>("set-local-description",  &[&offer_sdp, &None::<gst::Promise>],
    );
  })

  self.webrtcbin.emit_by_name::<()>("create-offer", &[&None::<gst::Structure>, &promise]);
});
```

whipsink

asymptotic

Send Offer:

```
self.webrtcbin.connect_notify(Some("ice-gathering-state"), move |webrtcbin, _pspec| {
...
  match state {
  ...
    WebRTCICEGatheringState::Complete => {
    // We got all the ICE candidates in the SDP
      ...
      self_ref.send_offer().await
      ...
  }
...
```

HTTP POST:

```
async fn send_offer(&self) {
...
  wait_async(&self.canceller, self.do_post(offer_sdp), timeout).await
...
}
```

whipsink

Parse response and set remote description:

```
async fn parse_endpoint_response(
....
){
...
  match resp.status() {
    StatusCode::OK | StatusCode::CREATED => {

    set_ice_servers(&self.webrtcbin, resp.headers());
    ...
    resp.headers().get(reqwest::header::LOCATION);
    ...
    //extract the SDP Answer from the response
    match resp.bytes().await {
      Ok(ans_bytes) => match sdp_message::SDPMessage::parse_buffer(&ans_bytes) {
        Ok(ans_sdp) => {

          let answer = gst_webrtc::WebRTCSessionDescription::new(
            gst_webrtc::WebRTCSDPType::Answer,
            ans_sdp, );
```

asymptotic

whepsrc

- WebRTC consumer i.e., RecvOnly
- Provides an RTP encoded stream to downstream

```
Pad Templates:
  SRC template: 'src_%u'
  Availability: Sometimes
  Capabilities: application/x-rtp
```

Example pipeline:

```
gst-launch-1.0 whepsrc name=whep auth-token=$WHEP_TOKEN whep-endpoint=$WHEP_ENDPOINT \
whep.src_0 ! rtph264depay ! ... ! autovideosink
whep.src_1 ! rptopusdeay ! ... ! autoaudiosink
```

rswebrtc plugin

- High level WebRTC elements
- `webrtcsink` - WebRTC producer
- `webrtcsrc` - WebRTC consumer
- The "all-batteries included" WebRTC solution
- Inbuilt support for raw and encoded streams
- Inbuilt congestion control algorithm
- Continuous improvements and feature additions
- …

Signallable

- Interface for Signaling
  - WebRTC elements can implement their own protocol with this
- Makes easy to write custom protocols with almost no change in the core (sink/src)
- WebSockets by default
- Operates on a set of signals that the WebRTC elements and Signaler code can communicate with

Signallable

asymptotic

```
Signals
-------
  consumer-added
  consumer-removed
  end-session
  error
  producer-added
  producer-removed
  request-meta
  send-ice
  send-session-description
  session-requested
  session-started

Action Signals
--------------
  handle-ice
  session-description
  session-ended
  shutdown
  start
  stop
```

Signallable

Methods, overridden based on the signaling protocol

```
fn request_meta(_iface: &super::Signallable) -> Option<gst::Structure> {}

fn start(_iface: &super::Signallable) {}

fn stop(_iface: &super::Signallable) {}

fn send_sdp(
  _iface: &super::Signallable,
  _session_id: &str,
  _sdp: &gst_webrtc::WebRTCSessionDescription,){}

fn add_ice(
  _iface: &super::Signallable,
  _session_id: &str,
  _candidate: &str,
```

WHIP Client as a Signaler

- A newer version of `whipsink` adapting Signallable. Thanks to Mathieu
- To leverage all good things from `webrtcsink` e.g., congestion control
- `whipclientsink` (a.k.a `whipwebrtcsink`)
- Implements `Signallable` Interface

```
gst-launch-1.0 whipclientsink name=whip signaller::whip-endpoint=$WHIP_ENDPOINT \
videotestsrc ! whip. \
audiotestsrc ! whip.
```

```
impl ObjectImpl for WhipWebRTCSink {
  fn constructed(&self) {
...
    let _ = ws.set_signaller(WhipClientSignaller::default().upcast());
  }}
```

WhipClient implementation

Does all the WHIP Client related functions in the implementation. Same as `whipsink`

```
impl WhipClient {
...
...
  // exactly same as whipsink
  async fn send_offer(&self, webrtcbin: &gst::Element) {
  ...
  ...
  }

  async fn do_post(&self, offer: gst_webrtc::WebRTCSessionDescription, webrtcbin: &gst::Element) {
  ...
  ...
  }

  async fn parse_endpoint_response(... ) {
  ...
```

Signallable for WhipClient

```
impl SignallableImpl for WhipClient {
fn start(&self) {
  ...
  ...
  // wait for underlying webrtcsink to signal consumer-added
  self.obj().connect_closure("consumer-added",
  false,
  glib::closure!(|signaller: &super::WhipSignaller,
    _consumer_identifier: &str,
    webrtcbin: &gst::Element| {
...
      webrtcbin.connect_notify(Some("ice-gathering-state"), move |webrtcbin, _pspec| {
        match state {

        WebRTCICEGatheringState::Complete => {
...
            obj.imp().send_offer(&webrtcbin).await

        }
        ...
    );

...
    // lets webrtcsink create a consumer-pipeline
    // passing None makes it generate offer
    self.obj().emit_by_name::<()>("session-requested",
    &[&"unique",
    &"unique",
```

WhipServer Implementation

- `whipserversrc` element for WHIP Endpoint plus Media Server
  - The other side of the WHIP story
  - Based on `webrtcsrc`
  - Merge request in progress
  - Initial version can accept stream from only single producer (WHIP client)

```
gst-launch-1.0 whipserversrc signaller::host-addr=$WHIP_ENDPOINT  name=ws ! \
queue ! videoconvert ! autovideosink \
ws. ! queue ! audioconvert ! autoaudiosink
```

WhipServer Implementation

```
// called when webrtcsrc emits `webrtcbin-ready`
pub fn on_webrtcbin_ready(&self) -> RustClosure {
    webrtcbin.connect_notify(Some("ice-gathering-state"), move |webrtcbin, _pspec| {
    match state {
        ...
        WebRTCICEGatheringState::Complete => {
            let ans = webrtcbin.property::<Option<WebRTCSessionDescription>>("local-description")
            tx.send(ans).unwrap()
        }

async fn post_handler(
    &self,
    body: warp::hyper::body::Bytes,
) -> Result<http::Response<warp::hyper::Body>, warp::Rejection> {

  // communicate the peer id with webrtcsrc
    self.obj().emit_by_name::<()>("session-started", &[&ROOT, &peer_id]);

    let offer_sdp = gst_sdp::SDPMessage::parse_buffer(body.as_ref());
    // Create an SDP of Offer type and set it on the webrtcbin

    self.obj().emit_by_name::<()>("session-description", &[&"unique", &offer]);

    // wait for the answer through tx.send in on_webrtcbin_ready
    let ans = rx.recv_timeout(Duration::from_secs(wait_timeout as u64))
    // unwrap and send response
}

async fn delete_handler(&self, id: String) -> Result<impl warp::Reply, warp::Rejection> {
```

Existing challenges

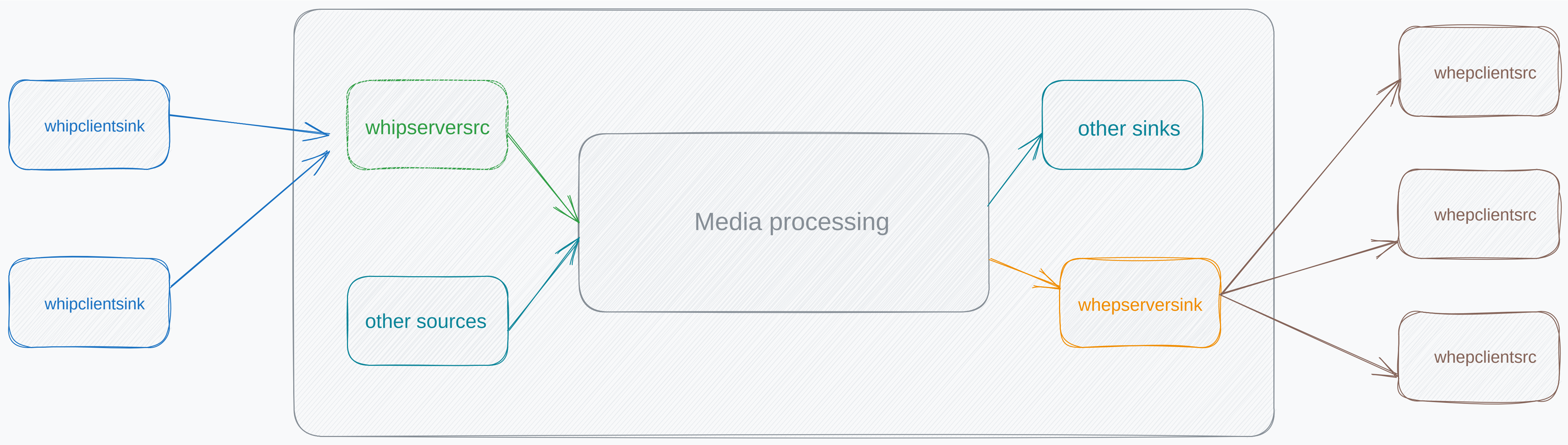
- Handling concurrent multiple producers and multiple sessions unsupported yet in `webrtcsrc`
  - sends EOS downstream when a session ends
  - applications need to launch multiple instances of `whipserversrc` for concurrent sessions
  - increases the complexity at application level

What's further?

- Implement WHEP Server and WHEP Client as Signalers
  - `whepclientsrc` - WHEP client as a `webrtcsrc` type element
  - `whepserversink` - WHEP server as a `webrtcsink` element
- Add support for multiple producers in `webrtcsrc`
- Add multi client support in `whipserversrc`
- `webrtcsink` and `webrtcsrc` to accept and produce RTP streams respectively
  - retire `whipsink` and `whepsrc` eventually

asymptotic

Finally... the entire ecosystem



Questions?