Developing a Gstreamer-Based Custom Camera System for Long-Range Biometric Data Collection

Gavin Jager

Researcher and Lab Space Manager

Oak Ridge National Laboratory

This research is based upon work supported by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA), via D20202007300010. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of ODNI, IARPA, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation therein.

IARPA
CREATING ADVANTAGE THROUGH RESEARCH & TECHNOLOGY

OAK RIDGE National Laboratory

Background

IARPA BRIAR:

Biometric **R**ecognition and **I**dentification at **A**ltitude and **R**ange

Challenging imaging conditions:

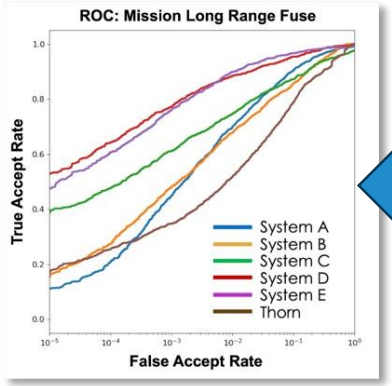
- Atmospheric distortion
- Camera motion

Existing algorithms:

- Are trained on low-angle, close-range data
- Rely on face recognition as the only biometric modality

ORNL's role: data collection and T&E

Oak Ridge National Laboratory's Role as BRIAR T&E Lead



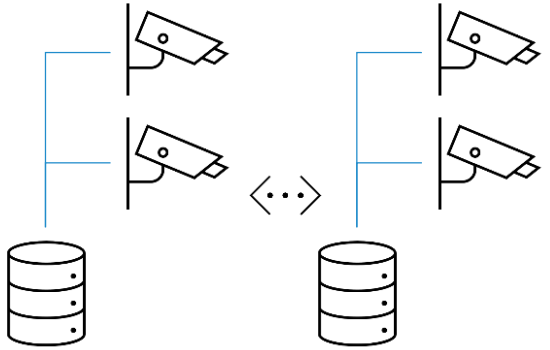
Testing and evaluation of performer team recognition models

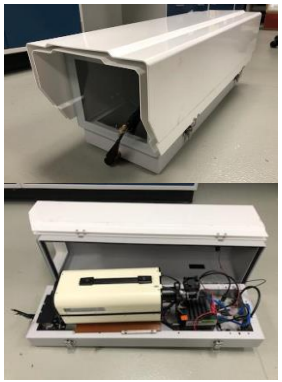
Experiment design, subject recruiting, dataset collection, curation, and annotation

Engineering and integration of support technology, cameras, and equipment

Custom Cameras Use-Case: Specialized Data Collection

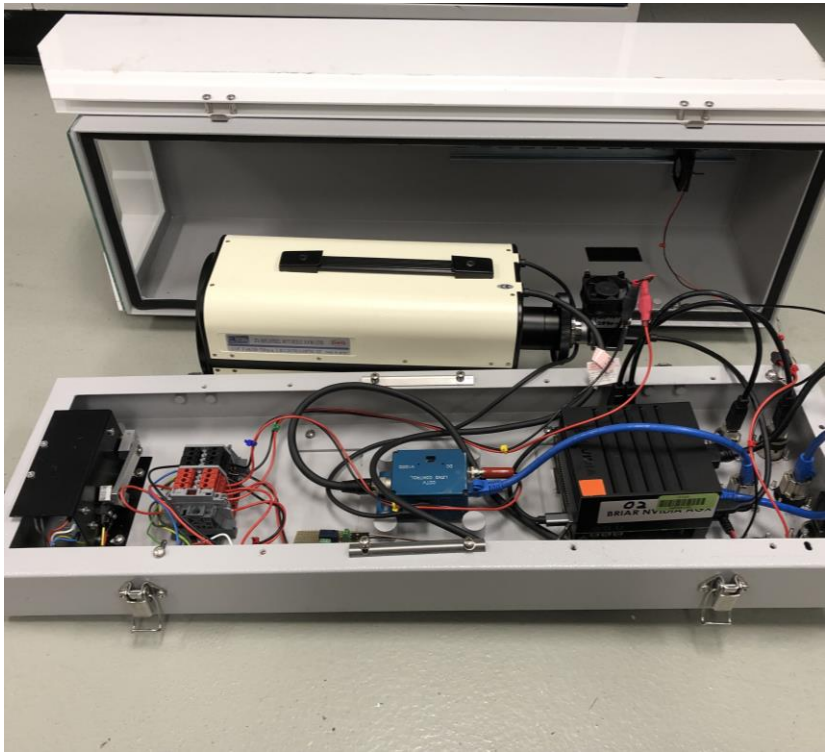
- Integration with commercial surveillance infrastructure: dozens of IP cameras, network video recorders at stations 5 m to 1 km from subject area(s)
- Interchangeable optics and sensors
- Network streaming and control of multiple custom cameras
- Raw capture recording





BRIAR Specialized Camera

- Nvidia Jetson AGX
- Basler Ace sensors
- High power lenses
- Control hardware (lens-specific)
- Weatherproof enclosure
- Mounting hardware for tripods, pan/tilt positioners, etc.

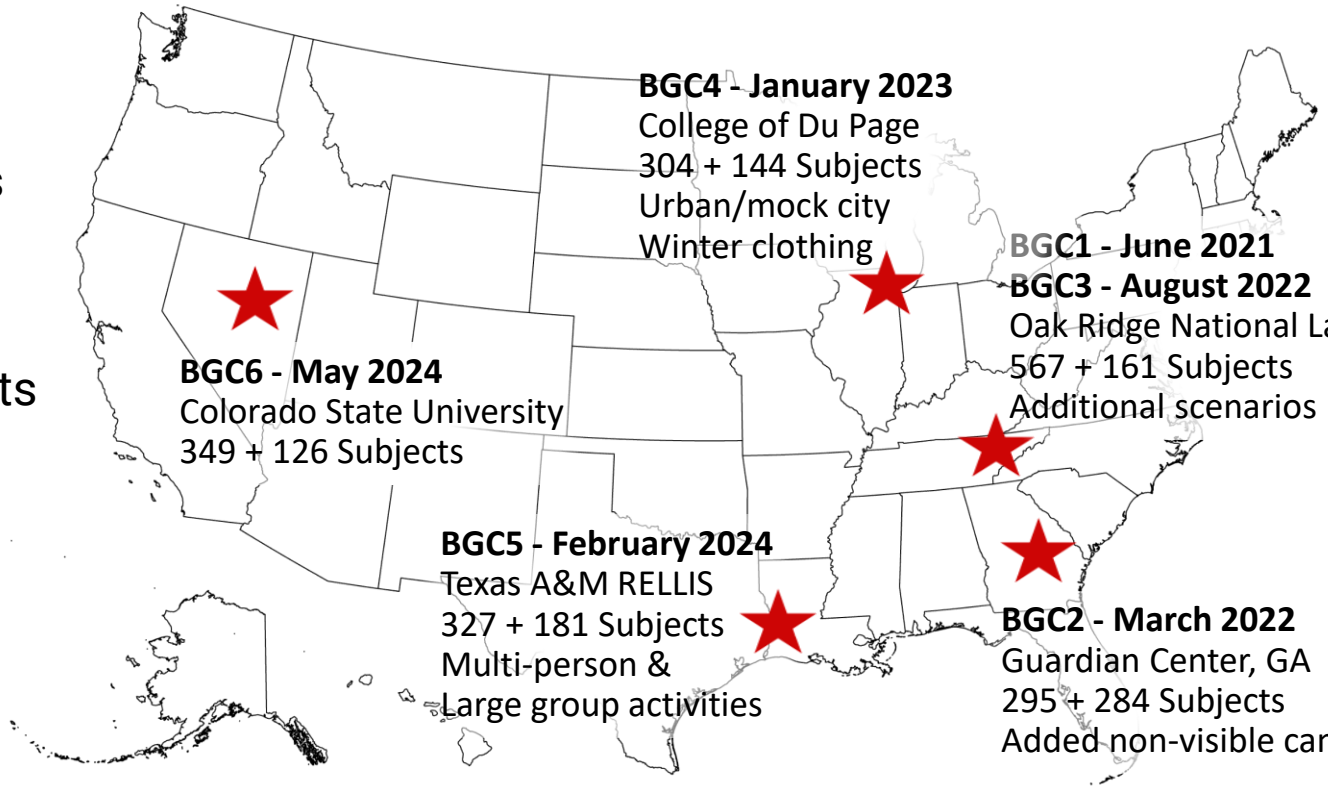


Collection Details

1842 Full Subjects

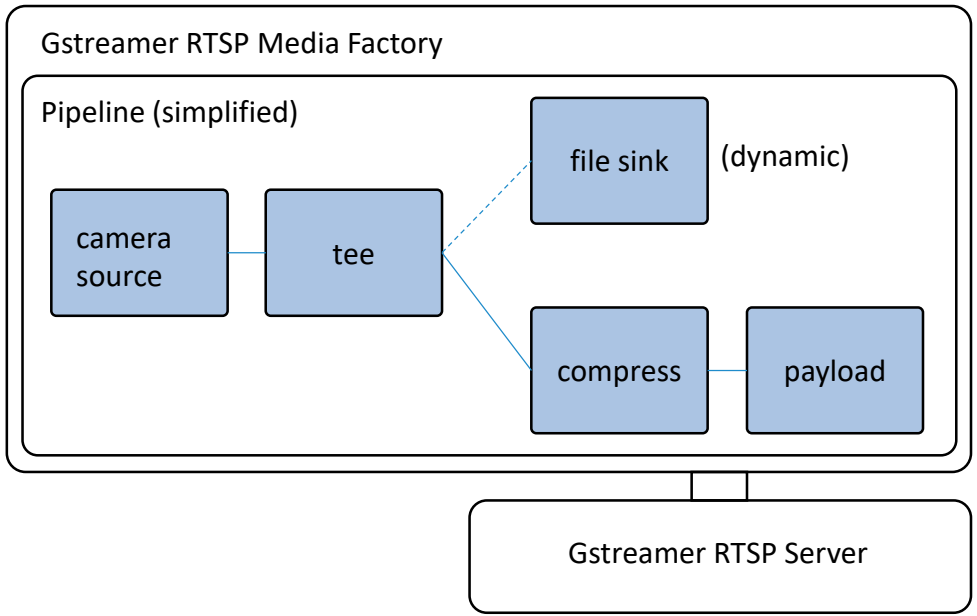
896 Indoor Only

2738 Total Subjects



Why Gstreamer?

- C-based framework for pipelines of data sources, filters, and sinks
- Extensive libraries of plugins implement additional elements
- Built on top of GLib (for main loop abstraction and data types) and GObject (OOP shoehorned into C)
- Dynamic pipelines are edited with callbacks triggered by signals, timers, or probed data

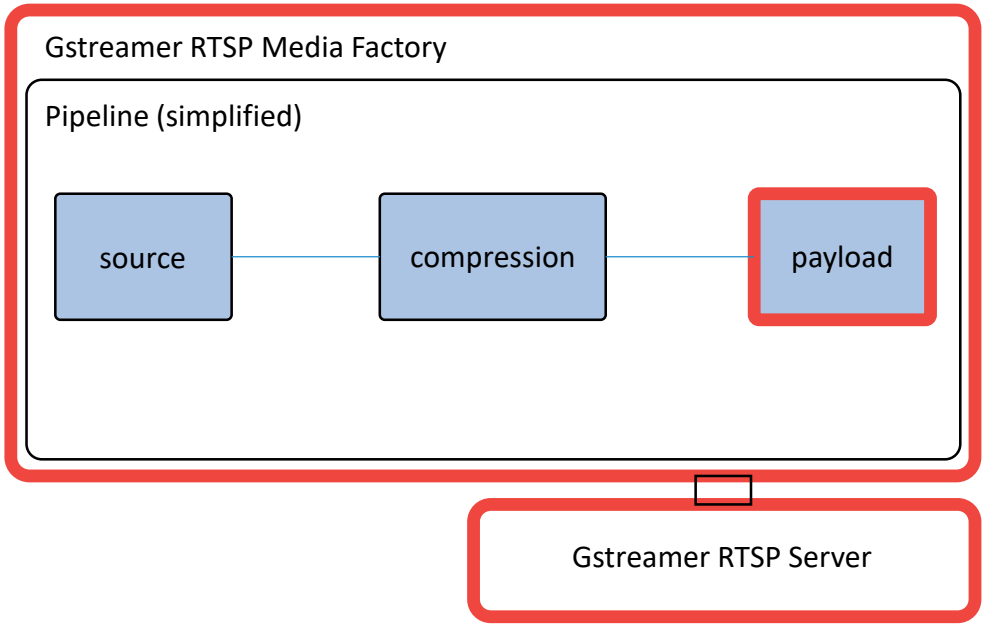


RTSP Streaming - GstRTSPServer

- The server signals a media factory object to create its pipeline when an RTSP client sends a request

`rtsp://<ip_address>:8554/<tag>`

- The `<tag>` is associated with a specific media factory
- Once the pipeline is running, the server streams the payload output over RTSP



GstRTSPServer – Pipeline Creation on Client Request

- The media factory creates and links pipeline elements from the description in `pipeline_string`

- The `media-configure` signal triggers a user-defined callback to perform any pipeline configuration or additional actions needed

```
// ----------  Set up RTSP ----------

GstRTSPServer *rtsp_server;
GstRTSPMountPoints *rtsp_mount_points;
GstRTSPMediaFactory *rtsp_media_factory;

rtsp_server = gst_rtsp_server_new();
gst_rtsp_server_set_address(rtsp_server, control_data->RTSPaddress);
rtsp_mount_points = gst_rtsp_server_get_mount_points(rtsp_server);
rtsp_media_factory = gst_rtsp_media_factory_new();
gst_rtsp_media_factory_set_launch(rtsp_media_factory, pipeline_string);
gst_rtsp_media_factory_set_shared(rtsp_media_factory, TRUE);
g_signal_connect(rtsp_media_factory, "media-configure",
    (GCallback)media_configure, (gpointer) &pipeline_data);
// EX: rtsp://127.0.0.1:8554/config1_acA2040-120uc
gchar *mount_point_str = g_strdup_printf("/%s_%s", metadata->name, sensor_setup_info->name);
gst_rtsp_mount_points_add_factory(rtsp_mount_points, mount_point_str, rtsp_media_factory);
g_object_unref(rtsp_mount_points);
guint rtsp_server_id = gst_rtsp_server_attach(rtsp_server, NULL);

timestamp_prefix_log();
g_print("RTSP server ready at rtsp://%s:8554%s\n", control_data->RTSPaddress, mount_point_str);
```

System Configuration

- JSON files: `config.json` and `sensor.json`
  - Set important metadata, as well as system, network, sensor, and peripheral hardware settings
  - These files are read into structs using cJSON on startup
- Text file: `pipeline.txt`
  - Creative string formatting for pipeline configuration
  - Minor changes without recompile

```
{
    "Metadata":
    {
        "Name":             "config1",
        "Location":         "field",
        "Range":            500,
        "Yaw":              12,
        "Elevated":         false,
        "Lens Name":        "Sigma 600",
        "Focal Length":     600,
        "Collection ID":    "BGC6"
    },
    "Control Data":
    {
        "Record Raw":               true,
        "Raw Storage":              "/data/briar_rec",
        "Control IP":               "127.0.0.1",
        "Control Port":             9000,
        "Status Broadcast Port":    9001,
        "Raw Time Limit":           600,
        "RTSP Address":             "127.0.0.1",
```

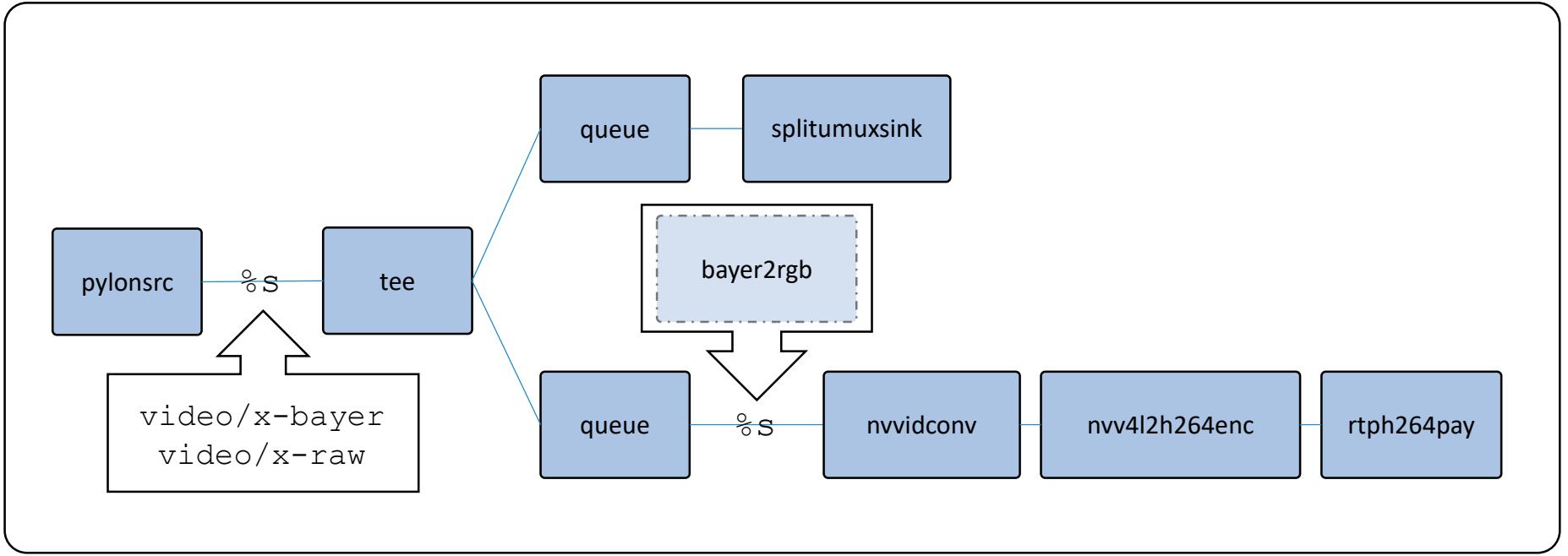
A Gstreamer Pipeline Description (a mess, but bear with me…)

```
pylonsrc name=source ! %s !

tee name=t ! queue name=raw_queue !

splitmuxsink name=sink muxer=identity
location=/dev/null

t. ! queue name=rtsp_queue !%s
nvvidconv name=converter !

video/x-raw(memory:NVMM),
format=I420,framerate=30/1 !

nvv4l2h264enc bitrate=20000000 !
h264parse !

rtph264pay name=pay0
```

- Gst-plugin-pylon camera source element*
- Tees and Queues abstract multithreading
- `splitmuxsink` element for timed/toggled local recording
- Second branch of the tee is used for RTSP stream, format converted*
- Capabilities or “Caps” describe data frames and transmission between elements
- Nvidia-specific hardware encoder element encodes video stream to h264. Data buffers are parsed into packets
- Payload element exposes stream to GstRTSPServer

*** `%s` -- C string formatting; caps and element insertion based on sensor characteristics**

Pipeline Description



Basler gst-plugin-pylon

- **pylonsrc:** Gstreamer source element plugin for all Basler 2D cameras
- All camera features exposed as gstreamer properties
- We used color and mono sensors as well as a variety of resolutions and formats
- **Issues with the beta versions**: broken cleanup of camera elements, difficulties getting/setting effective resolutions
- v1.0.0 released Aug ’24, but throughout development and field-use we were working with v0.5.1 through v0.7.0

Sensor Setup: It's not a bug, it's a feature!

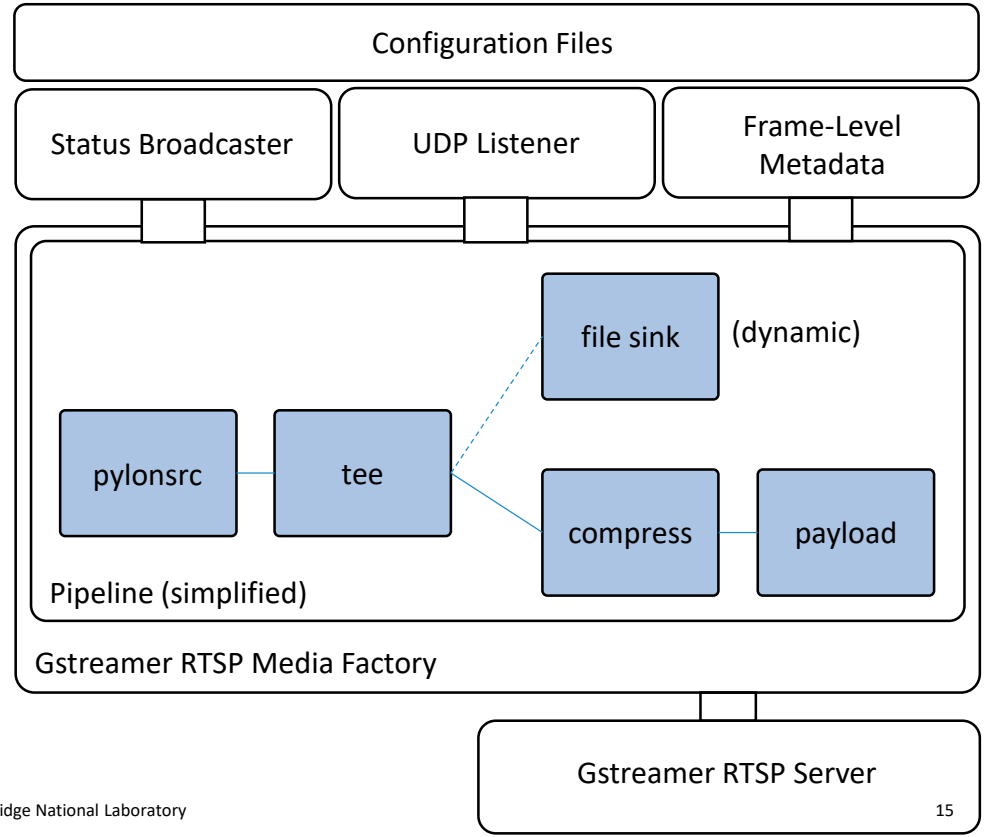
- `sensor_setup` – separate executable run any time a new sensor was added
- Workaround for gst-plugin-pylon beta version bugs
- Writes JSON configuration file with editable sensor settings

```
{
    "Model":          "acA2040-90um",
    "Serial Number":"00000000",
    "Caps Type":      "video/x-raw",
    "Color Filter":  "GRAY",
    "Width":          2048,
    "Height":         2048,
    "Exposure Time":1000,
    "Gain":           3,
    "Auto Exposure":"AUTO_ONCE",
    "Auto Gain":      "AUTO_OFF"
}
```

```
SensorSetupInfo *loadSensorSetupInfo(void) {
    /*
    Load sensor information from a JSON file written by
    writeSensorSetupJSON.
    Returns a SensorSetupInfo struct with a fully formatted caps string
    for the gstreamer pipeline
    */
```

Application Interfaces

- Status Broadcaster: sends system information over UDP
- UDP Listener: handles incoming messages to toggle raw recordings, change sensor settings
- Frame-Level Metadata: probes the frame buffers moving through the pipeline for detailed info, uses another gstreamer pipeline to write to CSV files.



“media-configure” and Useful Callbacks

`g_signal_connect:` attach a callback to a specific signal

- In this case, the callback changes the path of the recording file that `splitmuxsink` element writes

`g_io_add_watch:` attach a callback to some I/O action

- In this case, the callback processes incoming UDP messages

```
130     // get the raw recording elements if they exist
131     if (pipeline_data->control_data->recordRaw) {
132         pipeline_data->sink =

// allow the location of the splitmuxsink to be changed by signal
g_signal_connect(pipeline_data->sink, "format-location",
    (GCallback)format_recording_filepath, pipeline_data);

139         pipeline_data->raw_queue =
140             gst_bin_get_by_name_recurse_up(GST_BIN(media_bin), "raw_queue");
141     } else {
142         pipeline_data->sink = NULL;
143         pipeline_data->raw_queue = NULL;
144     }

// listen for incoming UDP packets
pipeline_data->udp_channel_watch_id =
    g_io_add_watch(pipeline_data->udp_channel, G_IO_IN,
    (GIOFunc)udp_received_callback, pipeline_data);

152     get filepaths for the sensor and frame-level metadata
153     then create those files
154     */
155     char *recording_dir = NULL;
156     if (format_recording_directory(pipeline_data->control_data, &recording_di
157         timestamp_prefix_err();
```

“media-configure” and Useful Callbacks

`g_timeout_add_seconds`:
execute a timed callback

- Used to enforce a time limit on local recordings and metadata files

`gst_pad_add_probe`: access data buffers at an input/output of an element

- Access/record frame-level metadata while pipeline plays

```
203     // start the file limit timer
        pipeline_data->file_limit_callback_id =
            g_timeout_add_seconds(time_limit,
            (GSourceFunc) file_limit,
            (gpointer) pipeline_data);
211     frame_metadata_filepath = NULL;
212
213     // install the frame-level metadata probe at the source element
        src_pad = gst_element_get_static_pad(pipeline_data->source, "src");
        pipeline_data->metadata_probe_id =
            gst_pad_add_probe(src_pad, GST_PAD_PROBE_TYPE_BUFFER,
            (GstPadProbeCallback) frame_level_metadata,
            (gpointer) pipeline_data,
            (GDestroyNotify) metadata_cleanup);
```

Peripheral Hardware Interfaces: Lens Control

Standardized Description

UDP listener settings defined in `config.json`

```
"Name":             "ISSI Focus Controller",
"Filepath":         "hardware/issi_focus_control.py",
"Listener Port":    "7925",
"Device IP":        "192.168.2.252",
"Device Port":      "1339",
"Capabilities": [
    {
        "Name": "Focus",
        "Commands":["+", "++", "-", "--"]
    }
]
```

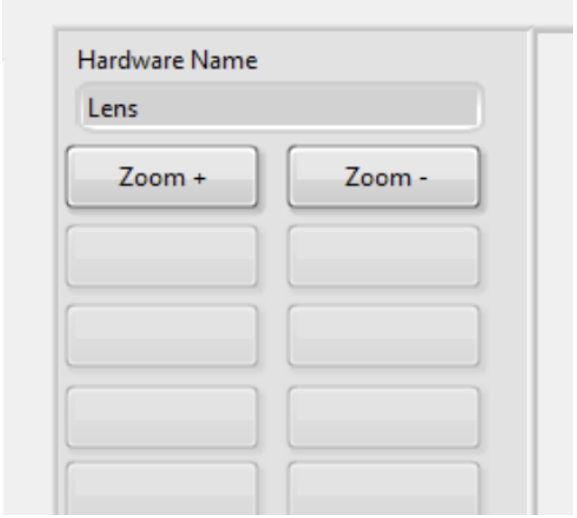
Python Subclassing

Device specific implementation, e.g., network device, GPIO

```
from interface import HardwareInterface

class ISSIFocusControl(HardwareInterface):
    """Implements the EF-lens focus control

    capabilities = {
        "Focus":["+", "++", "-", "--"],
    }

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        self.step_size = 50

    def Focus(self, arg):
        if arg == "+":
```

Status Broadcast

JSON informs monitoring and control GUI.



Conclusions

Challenges

- Choosing the correct elements for the hardware and application
- Configuring a dynamic pipeline within an RTSP Server

Lessons Learned

- Stress testing should be *really* stressful

Insights and Advice

- It'll take longer than you think
- Friends and documentation
- Don't be afraid of workarounds

Cameras in the Wild!

Resources/Links

Oak Ridge National Laboratory	https://www.ornl.gov/
IARPA – BRIAR Home Page	https://www.iarpa.gov/research-programs/briar
Gstreamer Homepage	https://gstreamer.freedesktop.org/
Basler gst-pylon-plugin	https://github.com/basler/gst-plugin-pylon
cJSON	https://github.com/DaveGamble/cJSON

ORNL Open Source: https://github.com/ornl

Open-source license approval in progress at time of submission…