`potential-engine`: A simple RTSP server for FIRST Robotics Competition

h.264 video streaming and resulting bandwidth reduction in FRC

Introduction

Video streaming has been a major component of having a competition viable robot for several years. In prior years (e.g. 2017 and 2016) large obstacles have obstructed parts of the field, and for a significant portion of the match this year driver vision is completely obstructed by the SANDSTORM.

The default (and, at time of writing, only) video codec supported by WPILib's camera server is the Motion JPEG (MJPEG) format. MJPEG does *work*, and arguably has advantages for FRC. Encoding each frame independently reduces latency, but comes at a bandwidth cost. This is particularly stinging with the bandwidth cut for 2019 (from 7Mb/s per team to 4Mb/s per team.) Formats such as MPEG (note the lack of J) and h.264 can be more than two times more effective in terms of compression ratio. With hardware-accelerated encoding being available for USD$35 in the form of a Raspberry Pi, the computational difficulty of encoding h.264 video becomes a largely moot point. This document will outline how we have implemented h.264 video encoding on a Raspberry Pi, driver station video decoding, and network usage/latency measurements.

Video Encoding

The Raspberry Pi 3B has the capability to do hardware-accelerated video encoding out of the box, with no extra hardware required. However, the default Raspbian Lite image does not contain the required software to take advantage of this. (Other images might, but running a X server on a system that is largely "set and forget" is not the greatest idea due to additional resource consumption.)

Software setup

Install and Setup Raspbian

Teams that already know how to set up a Pi with Raspbian Lite, skip to "Install Dependencies."

Starting with the obvious, find a good microSD card that doesn't have anything important on it. All data on this card **will be lost.** The Raspberry Pi Foundation has an excellent guide (https://www.raspberrypi.org/documentation/installation/installing-images/README.md)on installing a particular image, but to sum up:

1. Download the Raspbian Lite image from the Raspberry Pi Foundation. It's not required to extract the zip file. Be careful not to choose "Download Torrent".
2. Install Etcher. Etcher is a fantastic tool for writing disk images in general, keep it handy.
3. Connect/insert the SD card.
4. Flash!
    a. Open Etcher

b. Click “Select image” and select the zip file from step one.
c. Click “Select drive” and find the drive in the list. If it doesn’t appear, make sure the SD card is connected. Etcher should protect you from overwriting your hard drive, but **be careful anyway.**
d. Hit “Flash!” There will likely be a prompt for some form of password or authorization. This will take a while.

After burning the card, insert it into the Pi. Connect a keyboard and monitor - the CLI does not understand mice or other rodents. Power on with a good USB power supply. Most Micro USB phone chargers are up to the task. To connect to a VRM, use a 5V/2A channel. After a little waiting, a login prompt should appear. The default username is `pi`, with the password `raspberry`.

There are a few settings to change sooner rather than later. Run `sudo raspi-config` at the command prompt, and do the following:

- If black bars surround the screen, disable overscan compensation (Advanced Options -> Overscan)
- Enable SSH for remote (“headless”) access later (Interfacing Options -> SSH.) You will want this when the Pi is mounted to a robot and difficult to remove and connect to a keyboard and monitor.
- Change your keyboard layout to match the keyboard you’re using (Localisation Options -> Change keyboard layout)
- Expand the filesystem to use the whole card (this may happen automatically on first boot. Advanced Options -> Expand Filesystem)

Hit the right arrow key twice to select “Finish”, hit Enter, and reboot.

Install Dependencies

Before the streaming and encoding software is installed, it’s a good idea to update and upgrade Raspbian. Connect the Pi to the Internet and run the following command to update and upgrade all at once:

```
sudo apt update && sudo apt upgrade
```

When prompted if you wish to continue by `apt`, hit enter (or `Y`, then enter.) You’re now ready to install a long list of dependencies:

```
# n.b.: you may not need the good, the bad, and the ugly all installed,
depending on your camera of choice
sudo apt install git cmake gstreamer1.0-omx-rpi gstreamer1.0-tools
gstreamer1.0-plugins-good gstreamer1.0-plugins-bad
gstreamer1.0-plugins-ugly libgstreamer1.0-dev
```

That is a mouthful of a command, but it’s not necessary to install all those packages in one go - they can be done one at a time (`sudo apt install git`; `sudo apt install cmake`, etc.)

Raspberry Pi Camera Source

Teams using the Raspberry Pi Camera module will need to install a dependency from source code. Luckily, the process has been made as simple as possible and only takes five commands:

```
# install required dependencies from apt
sudo apt install autoconf automake libtool pkg-config libgstreamer1.0-dev
libgstreamer-plugins-base1.0-dev libraspberrypi-dev
# fetch the source code
git clone "https://github.com/thaytan/gst-rpicamsrc.git" && cd
gst-rpicamsrc
# build and install
./autogen.sh --prefix=/usr --libdir=/usr/lib/arm-linux-gnueabihf/
make
sudo make install
```

Make sure the module installed OK:

```
# this should output a detailed explanation of the rpicamsrc element
gst-inspect-1.0 rpicamsrc
```

Stream away!

RTSP

We've created a simple streaming server for RTSP streaming. It supports the Raspberry Pi Camera Module, as well as any webcam available with Video4Linux (provided that the V4L driver provides YUY2 raw color.) The GPL'd source code is on GitHub (https://github.com/BHSSFRC/potential-engine), and pull requests are welcome!
RTSP requires some additional dependencies:

```
# apt packages
sudo apt install libgstrtspserver-1.0-dev gstreamer1.0-rtsp
# fmt library
# sadly the Raspbian version of this package is out of date
cd # go home
git clone "https://github.com/fmtlib/fmt.git" # download {fmt} source
cd fmt && mkdir build && cd build # create build files directory
cmake .. # create build files
sudo make install # build and install {fmt}
```

After installing dependencies, download and build the streaming server:

```
cd && git clone "https://github.com/BHSSFRC/potential-engine.git"
cd potential-engine && mkdir build && cd build
```

```
cmake ..
make
```

The program is controlled with a large number of run-time switches.

- `--rpi_cam`: Use the Raspberry Pi camera module. Requires the Raspberry Pi GStreamer source to be available (see: "Raspberry Pi Camera Source")
- `--no_rpi_cam`: Don't use the Raspberry Pi camera module. Default option.
- `--hardware_accel`: Use OpenMAX hardware acceleration. Does not have any effect if `rpi_cam` is set (the Raspberry Pi Camera Module does hardware encoding automatically.)
- `--height` (or `-h`): Sets video height. When using a Raspberry Pi Camera Module, the value must be `360`, `480`, or `720`. Otherwise, use `v4l2-ctl --list-formats-ext` to get a list of available framerate/resolution combinations. Setting this is the simplest way to reduce your bandwidth use. Defaults to `480`.
- `--fps` (or `-f`): Sets video frames per second (FPS.) Use `v4l2-ctl --list-formats-ext` to get a list of available framerate/resolution combinations for non-Raspberry Pi cameras. Defaults to `30`.
- `--url` (or `-u`): Sets the stream URL. Defaults to `/stream`. Examples (assume the Pi is at 10.34.94.66):
    - `./potential-engine -u /foobar -h ...`
        - The stream is available at `rtsp://10.34.94.66:1181/foobar`
    - `./potential-engine --url i_like_ponies -h ...`
        - The stream is available at `rtsp://10.34.94.66:1181/i_like_ponies`
- `--port` (or `-p`): Sets the stream port. Must be an integer. Defaults to `1181` (since this port should be open even when behind the Field Management System.) Please note that by default Linux restricts binding to ports below 1024 to the root user.

RTP

RTP streaming doesn't require any more dependencies, and saves a small amount of bandwidth, but it does have a few small catches to its use. Most stream viewers will require a Session Description Protocol (SDP) file to be somehow transferred to the client prior to streaming. This file is consistent as long as the address of the **client** and the video encoding settings remain the same. Additionally, the "server" must know the address of the client before it will be able to start streaming.

To stream h.264 video from a webcam connected to a Raspberry Pi, run the following command:

```
ffmpeg -f v4l2 -video_size 640x360 -framerate 10 -pixel_format yuyv422 -i
/dev/video0 -vcodec h264_omx -f rtp rtp://DR.I.V.ER:2222 -sdp_file
stream.sdp
```

This command will generate an SDP file in the current directory on the Raspberry Pi. Most video players will need this file to read the stream.

To stream from the Raspberry Pi Camera Module, use the following:

```
gst-launch-1.0 rpicamsrc !
"video/x-h264,height=640,width=360,framerate=30/1" ! h264parse ! rtph264pay
! udpsink host=DR.I.V.ER port=2222
```

Note that this does **not** create an SDP file automatically. Stack Overflow has an example file (https://stackoverflow.com/a/13234988/3551604):

```
v=0
m=video 2222 RTP/AVP 96
c=IN IP4 DR.I.V.ER
a=rtpmap:96 H264/90000
```

This file leaves a lot of settings to be "guessed" by your decoder, and it's best to either find a way to make it consistent (so the SDP file can be stored on the client prior to the stream) or to use a streaming protocol that avoids the issue entirely (such as RTSP.)

Video Decoding

Unfortunately, no FIRST-provided software is capable of decoding h.264 streams (over *any* protocol.) The options are currently:

- Use new (i.e. non-FIRST) software to display video.
- Extend FIRST-provided software to enable h.264 encoding

As of 2019-04-16, both solutions are legal for competition play. FIRST's only restriction on OPERATOR CONSOLE software is that the National Instruments Driver Station is used to command the robot to change modes, send joystick input to the robot, etc.

Using non-FIRST Software

GStreamer

GStreamer (https://gstreamer.freedesktop.org/) can be controlled with a simple, easy-to-read pipeline syntax and handles RTP streams without SDP files. It's also highly extensible (in stark contrast to FFmpeg requiring recompiling to add a single codec.) However, documentation ranges from sparse to arcane to simply nonexistent, official or not.

RTSP

On Linux, make sure you have `gstreamer1.0-libav` (or your distribution's equivalent) installed. On Windows, ensure that "GStreamer 1.0 libav wrapper" is selected to be installed during installation. On both platforms, run the following command to view the stream:

```
gst-launch-1.0 rtspsrc location="rtsp://SE.R.V.ER:8554/stream_url"
latency=0 ! rtph264depay ! avdec_h264 ! autovideosink sync=false
```

To use TCP instead of UDP, add `protocols=tcp` before the first exclamation mark (`!`).

RTP

RTP is nearly identical to RTSP for GStreamer.

```
gst-launch-1.0 -v udpsrc port=2222 ! "application/x-rtp" ! rtph264depay !
avdec_h264 ! videoconvert ! autovideosink sync=false
```

FFmpeg

FFmpeg (https://ffmpeg.org/) is known for being *the* software for video conversion. It's older than GStreamer (by about a month), and has extensive documentation - either in the form of `man` pages or online. Enough codecs are bundled into it to do nearly any job, however, adding more requires re compiling from the ground up. It will also form the basis for extending FIRST-provided software later.

RTSP

```
ffplay -fflags nobuffer "rtsp://SE.R.V.ER:8554/stream_url"
```

To use TCP instead of UDP, add `-rtsp_transport tcp`.

Note that `ffplay` has a small internal buffer to ensure smooth playback. This has a nasty tendency to introduce artificial latency, so using GStreamer will probably provide better results. An alternative command using `ffmpeg` alone would be `ffmpeg -i rtsp://SE.R.V.ER:1181/stream -f sdl -`. This has no internal buffering.

RTP

```
ffplay -fflags nobuffer -protocol_whitelist file,crypto,rtp,udp
stream_file.sdp
```

Pretty simple. The only complication is actually getting the SDP file in the first place. RTP still suffers from internal buffering.

Extending FIRST-provided Software

SmartDashboard

Due to changes in Java 11, plugin support is currently disabled in the SmartDashboard as of 2019-03-25. This means that to extend the program, we have to fork SmartDashboard and add new elements directly to the source. Luckily, we can modify the existing `MjpgStreamViewer` class to suit other codecs. (In fact, by using FFmpeg, we can support a large number of codecs at once.) Our version of this modification is, again, hosted on GitHub, (https://github.com/BHSSFRC/SmartDashboard) with pull requests welcome. To roughly describe the process:

1. Add JavaCV (https://github.com/bytedeco/javacv) to `build.gradle`:
    1.1. In the `dependencies` block, add the line `compile group: 'org.bytedeco', name: 'javacv-platform', version: '1.4.3'`
2. Create your new element
    2.1. This class will essentially copy `MjpegStreamViewer`. The main difference is that instead of using an `InputStream` to fetch frames, we will make use of a `FFmpegFrameGrabber` to grab frames and a `Java2DFrameConverter` to convert the video frames to `BufferedImage` instances.

A reference for this can be found at https://git.io/fh6G6. Some modifications that you might want to consider:

- Any options that can be passed to `ffplay` can be passed to a `FFmpegFrameGrabber` via `setOption(name, value)`.
- Generalizing the shared code between `MjpegStreamViewer` and your new class into a unified stream viewer class to derive subclasses for particular formats from.

Network Measurements

All measurements are the peak values found for no less than one minute of streaming. Bandwidth use was measured with Wireshark (https://www.wireshark.org/) with a one second interval in formal testing. We used GStreamer to view the stream, since it seemed to have the least latency.

Raspberry Pi Camera v2

FPS	**Resolution**	**Transport**	**Peak bandwidth (Mbps, still image)**	**Peak bandwidth (Mbps, driving)**	**Rough Latency (ms)**	**Networking HW**
30	426x240	UDP	1.052		100	Robot
30	640x360	TCP	2.2488			Robot
30	640x360	TCP	2.2752	#N/A		Building
30	640x360	UDP	1.8176	#N/A		Building
30	640x480	TCP	2.348	#N/A		Building
30	640x480	UDP	2.7616	#N/A		Building

Notes

Significant image errors occur when using UDP on the robot networking hardware as of 2019-02-16. This is probably due to the radio allowing more out-of-order packets than the hardware used in other tests.
We could not sucessfully stream 720p over the stock FRC radio with either TCP or UDP.

Comparison: Limelight v1 MJPEG

FPS	Resolution	Transport	Peak bandwidth (Mbps, still image)	Peak bandwidth (Mbps, driving)	Latency (ms)	Mode
22	320x240	TCP	1.1096			Computer vision
22	320x240	TCP	3			Driver video

Notes

The Limelight is a *fantastic* vision system. The fact that it evidently consumes more bandwidth to stream video should by no means discourage its use (rather, aim to optimize what you can to avoid it becoming a problem.)
We measured this using a “vision processing” pipeline for the first test, and a “driver” pipeline for the second test. “Driving” pipelines (i.e. higher exposure) will use significantly more bandwidth.
In informal testing, we saw these pipelines run as high as 3 Mbps (leaving only 1 Mbps for all other robot comms!)

Troubleshooting

RTSP

If you’re having issues with `potential-engine`, please file an issue on GitHub and we’ll do what we can to fix it. (If you have a *solution* to an issue, write a pull request!)
Some common issues we had are below.

The stream is shown as starting on port -1

Linux doesn’t let non-root users bind to ports < 1024. Either run the program as root with `sudo` or use a higher port. Also, make sure the port you’ve specified is not already bound and the server “owns” the specified address.

The server says any IPv6 address is invalid and uses 0.0.0.0 instead

This is a known issue with `potential-engine` as of 2019-04-16. This may be fixed in the future, but is low priority since IPv6 is relatively uncommon in the context of FRC.

RTP

Image suffers from gray blobs/decoding errors

You are probably losing packets to UDP. This is the nature of RTP - see "Why Does RTP use UDP instead of TCP?" (https://stackoverflow.com/q/361943/3551604) Try using RTSP with TCP transport instead.

Common

Unusually high latency

Various factors can add latency to the camera stream.

TCP

TCP is a double edged sword. On the one hand , you will almost always get every single packet delivered. However, if any packets drop, the latency introduced by retransmission will "pile up" over time. If this becomes a significant problem, consider using UDP or lowering your stream quality.

Video encoding and decoding

Software encoding takes *eons* compared to hardware encoding. Using the hardware encoder available on a Raspberry Pi will drastically reduce stream latency for minimal effort.

Alternatively, your client (driver station) may be too underpowered to decode the stream in a reasonable amount of time. Consider using a more powerful computer or look into hardware-accelerated decoding options available for your hardware. (Most likely these will not be OpenMAX, as the OpenMAX project is primarily concerned with "low power and embedded system devices".) DirectX Video Acceleration (https://en.wikipedia.org/wiki/DirectX_Video_Acceleration) may be a good starting point for Windows devices, and is available in FFmpeg.

v4l2src internal data stream error

Verify that the following are true:

- GStreamer is as up-to-date as your distribution packaging goes. (Even Debian Stable has a version recent enough.)

    - Make sure you didn't accidentally install GStreamer 0.10. Debian stil packages this, and it's easy to slip up when tab-completing long commands.
- You are not pressing the camera to do more than it can provide (for instance, if a camera supports 480p60fps and 720p30fps it will probably not take kindly to 720p60fps.)
- Only `potential-engine` is accessing the camera. `potential-engine` can serve multiple clients, but other processes running on the same machine (such as those present on WPILib's FRCVision image) can lock a camera before `potential-engine` can use it.

GStreamer complains about missing OpenMAX elements

- Did you install both `gstreamer1.0-omx-rpi` and `gstreamer1.0-omx`?
- Does `gst-inspect-1.0 omxh264enc` print `No such element or plugin ‘omxh264enc’`? Does `gst-inspect-1.0 -b | grep omx` print anything?
    - If you answered yes to both questions on the line above, the OpenMAX plugins are blacklisted for some reason. Thankfully, this is easily fixed. Use `dpkg-query -L gstreamer1.0-omx-rpi` to list all files installed by `gstreamer1.0-omx-rpi`. One of these should be named `libgstomx-rpi.so`. Unblacklist it with `gst-inspect-1.0 /path/to/libgstomx-rpi.so` and the elements should be usable.