ROSlight

Publishing ROS Topics from the Limelight 2

Carlos Gross Jones
FRC Team 696
November 17, 2019

Abstract

In this whitepaper, careful examination of the Limelight software is undertaken. After gaining root access to the OS, the core ROS code is compiled and installed on the Limelight. Finally, ROS packages are developed to publish the Limelight output data (target orientation, etc.) as ROS topics.

1 Introduction

ROS, the somewhat poorly-named Robot Operating System, is a set of tools, frameworks, and protocols to allow various software elements to interact with well-defined interfaces. ROS incorporates a standardized system of communication interfaces (such as messages and services), a build and dependency-management architecture, and a rich set of command-line tools for operation and development. ROS is increasingly popular in industry thanks to its easily-extensible architecture and the large variety of existing nodes; open-source packages implementing extended Kalman filtering, LIDAR SLAM, and computer vision, to give a few examples, are available.

While ROS still requires a fair amount of effort and software development knowledge to implement, it is beginning to be taken up in FRC[1].

The Limelight is well-known in the FRC community as an easy-to-use, prepackaged computer vision system. It captures video (at a high framerate) with its built-in camera, runs openCV pipelines onboard, and then outputs the numeric results over Network Tables. Under the hood, the Limelight is not terribly complex. There are essentially four elements:

1. A Raspberry Pi computer;
2. A COTS camera;
3. A custom LED board;
4. And software/user interface.

The first three items are not difficult to come by. Many teams have implemented their own LED+camera solutions, with many employing the Raspberry Pi. What makes the Limelight popular is that it is a turnkey package and comes with an easy-to-use GUI, so that teams need very little knowledge or time to deploy an effective computer vision system.

Given that the Limelight is essentially a Raspberry Pi, it would be straightforward to flash stock Raspbian (or another OS) onto it, install ROS, and develop a custom software stack from the ground up. The goal of this project is to leave the existing Limelight software untouched (or at least functional), while adding ROS communication capabilities.

2 Gaining Access to the Limelight

First, of course, it is necessary to get a terminal and root access on the Limelight. `Nmap` showed that it did, in fact, have an SSH server listening on port 22. Knowing that the Limelight is essentially a Raspberry Pi, login was attempted using the default credentials (un: `pi`, pw: `raspberry`). This was unsuccessful.

The next step was examination of the Limelight image (the imaging process being one of the few ways the user has to modify the software running on the Limelight). At the time of writing, the most recent image is `LL_2019_71_RELEASE`, downloaded from https://www.mediafire.com/file/wjm1yb3ztr2zjen/LL_2019_71_RELEASE.zip/file. This is simply a zip archive containing `LL_2019_71_RELEASE.img`. Expecting that the image would contain at least one root filesystem, binwalk[2] was used to search for filesystem signatures:

```
binwalk -B --include='linux ext filesystem' LL_2019_71_RELEASE.img
```

[1] See https://team900.org/labs/

[2] See https://github.com/ReFirmLabs/binwalk

which revealed an ext4 filesystem called "rootfs" at offset 0x3000000. After mounting the relevant part of the image file with

```
mount -o loop,offset=50331648 LL_2019_71_RELEASE.img temp/
```

the mountpoint did indeed appear to contain the root filesystem of the Limelight. At this point, all of the usual coldboot shenanigans are applicable; in this case, in addition to resetting the password on the `pi` account via `/etc/shadow`, a new user account (`carlos`) was created and given sudoer privileges.

A cursory glance revealed that the Limelight is running a relatively-stock Raspbian image:

```
~/data/Limelight/temp$ cat etc/os-release
PRETTY_NAME="Raspbian GNU/Linux 9 (stretch)"
NAME="Raspbian GNU/Linux"
VERSION_ID="9"
VERSION="9 (stretch)"
ID=raspbian
ID_LIKE=debian
HOME_URL="http://www.raspbian.org/"
SUPPORT_URL="http://www.raspbian.org/RaspbianForums"
BUG_REPORT_URL="http://www.raspbian.org/RaspbianBugs"
```

After unmounting the root filesystem, the (now-modified) `LL_2019_71_RELEASE.img` was re-zipped and flashed onto the Limelight without incident. It was then possible to SSH into the Limelight as either `pi` or `carlos`, and to execute commands as root.

3 Installing ROS

Once the Limelight can be accessed vis SSH, the normal ROS Kinetic Raspberry Pi installation instructions[3] are successful. (Melodic would be preferable, but it seems to not yet be released for the `armhf` architecture.) As discussed in §3.3 of the instructions, the compilation (especially compiling `roscpp`) requires a lot of memory, and stalls when the 1 GB of RAM on the Pi is full. Given that the Raspberry Pi Compute Module 3 at the heart of the Limelight only has 4 GB of eMMC storage, and writing to it frequently is not preferred (see §4), a flash drive was plugged into the limelight to be used as swap space. After editing `/etc/dphys-swapfile` to give 2 GiB of swap space on the flash drive, and running

```
/etc/init.d/dphys-swapfile restart
```

to initialize it, the compilation completed without incident.

4 eMMC Precautions

The Raspberry Pi Compute Module 3 has 4 GB of eMMC storage soldered onto the module. This is different from the Raspberry Pi development boards, which generally have a slot to take a microSD card, and have no built-in secondary memory. As Tesla recently discovered to their chagrin[4], eMMC memory (which is, after all, flash) has a limited number of write cycles that it can tolerate. This suggests certain measures that might be taken to prevent Limelights from being similarly crippled[5].

Most importantly, build ROS offboard, as much as possible. The compilation process involves writing many temporary files to disk, which degrades flash memory. Building ROS packages from source often

[3] http://wiki.ros.org/ROSberryPi/Installing%20ROS%20Kinetic%20on%20the%20Raspberry%20Pi

[4] https://hackaday.com/2019/10/17/worn-out-emmc-chips-are-crippling-older-teslas/

[5] Note, however, that while flash has limited program/erase cycles, the limit is comparatively large– generally at least on the order of 10^3. Furthermore, eMMC controllers incorporate wear-leveling algorithms to attempt to write to all of the blocks on the device at approximately the same rate. Finally, the actual powered-on time of a Limelight is usually small compared to, say, a car. Therefore, this issue is not urgent; it may manifest after many seasons of use of the same Limelight.

requires the compilation of a large number of dependencies. After the initial compilation of `ros_comm` on the Limelight itself (although using an external flash drive as described above), a normal Raspberry Pi 3B+ was used as a "build server" to compile the `limelight_ros` package and dependencies. As opposed to the Compute Module 3 in the Limelight, the Pi has no built-in eMMC, but has a microSD card slot. In addition to providing much more disk space, the SD card can be replaced if it fails. Since the Raspberry Pi 3B+ and Compute Module 3 have the same architecture (`armhf`), binaries compiled on the Pi can simply be SCPed to the Limelight and used. In the case of ROS dependencies, it was possible to `tar` the whole `/opt/ros/kinetic` directory and move it over to the Limelight.

Another measure commonly taken to avoid flash wear in embedded devices is to mount the filesystem read-only. This is not done on the Limelight (although `/boot` is mounted read-only). There are a couple of potential problems that arise when the root filesystem is mounted as read-only. The largest one is that processes can no longer write log files to `/var/log` (or anywhere else). Since processes generally do not like to be denied access to their logfiles, it is common to use `tmpfs` or similar to mount a ramdisk at `/var/log`. This allows logs to be written, although they are not persisted through boot cycles.

In the case of the Limelight, a greater problem is pipeline editing. Pipeline definition files are stored as `/home/pi/visionserver/n.vpr`, where n is an index in $[0, 10)$. Similarly, some settings such as team number are stored in the `/home/pi/visionserver/master.settings` file. Thus, mounting the root filesystem read-only would prevent editing of Limelight and pipeline settings.

Perhaps a better solution would be mount a tmpfs at `/var/log` and any other locations with logfiles, and leave the root filesystem read-write. This would prevent constant log writes to eMMC, while still allowing users to edit the vision processing settings normally.

5 Overview of Limelight Architecture

5.1 Components

5.1.1 Visionserver

The primary binary associated with the actual Limelight vision processing is `/home/pi/visionserver/visionserver`. This program, which is started at boot, captures video, runs the pipeline, publishes data to Network Tables, streams video through various ports (see §5.2), and opens a websocket server to interact with the web interface.

5.1.2 Networkresponder

Another binary, `/home/pi/visionserver/networkresponder`, also runs in the background. Based on functional testing and decompilation, its only purpose seems to be hostname and IP discovery. If a UDP packet with the payload `LLPhoneHome` is sent to port 5809 on the limelight, a string like

```
h:limelight:ip:10.6.96.11
```

(that is, the Limelight's hostname and IP) will be sent back to port 5809 of the host which sent the first packet.

5.1.3 Apache

An Apache2 server is started to serve the static files that make up the web interface. This media, CSS, and Javascript can be found in `/var/www/html` as usual.

5.2 Ports

The Limelight listens on the following ports (in addition to the normal SSH, DHCP, and Avahi/zeroconf ports):

- 5800: Processed video stream (with overlays etc.)
- 5801: Web interface
- 5802: Raw camera stream
- 5805: Websocket
- 5809: `networkresponder`

5.3 Data Output

In order to push data to the web interface, `visionserver` runs a websocket server at `ws://limelight.local:5805`. This websocket publishes, among other things, a string of colon-delimited values, which looks like:

```
status_update 3.698395:4.292603:1.737678:-82.568588:6.157673:1:0:0:0:0:0:0:0:1:0.000000
:0.000000:0.000000:0.000000:0.000000:0.000000:0.000000
```

The meanings of these values are as follows:

Index	Name	Description
0	tx	Horizontal angle (degrees) from crosshair to target
1	ty	Vertical angle (degrees) from crosshair to target
2	ta	Target area (% of image)
3	ts	Target rotation (degrees)
4	tl	Pipeline latency (ms)
5	tv	Whether there are any valid targets in image
6	pipeline	Current pipeline index
7	ignoreNT	Whether NT-specified pipeline is being ignored
8	imageCount	
9	interfaceNeedsRefresh	
10	usingGRIP	
11	pipelineImageCount	
12	snapshotMode	Whether input is camera or snapshot
13	hwType	
14	trX	X translation component of solvePnP solution
15	trY	Y translation component of solvePnP solution
16	trZ	Z translation component of solvePnP solution
17	roX	X rotation component of solvePnP solution
18	roY	Y rotation component of solvePnP solution
19	roZ	Z rotation component of solvePnP solution

Table 1: Value mappings in `status_update` string

This websocket is the only interface between the web GUI presented to the user and the Limelight (other than the video stream itself). As such, any controls and feedback available to the user from the GUI is accessible from the websocket, and some other items that are not presented to the user. For example, the client will send the string

```
flush_settings 0*#
```

to cause `visionserver` to flush the current pipeline settings to one of the `.vpr` files. `visionserver` will then respond on the websocket with

```
flushed_settings
```

To turn the LEDs on for the current pipeline, the GUI sends the string

`set_setting 0*pipeline_led_enabled:1#`

or another value:

Value	Meaning
0	LEDs off
1	LEDs on
2	Left LEDs on
3	Right LEDs on

Table 2: LED Control Values

Further websocket activity can be investigated by using the debug functionality of a web browser to monitor the websocket messages while interacting with the web GUI.

6 ROS Interface

As proof-of-concept, two ROS components have been created: a publisher for output data, and a service for controlling the LEDs. This code can be found on Github[6].

The publisher (`websocketpublish.py`) connects to the websocket, reformats the data, and publishes it as `LimelightStamped.msg`:

```
Header header
geometry_msgs/Quaternion targetLocation
std_msgs/Bool hasTargets
std_msgs/Float32 targetArea
geometry_msgs/Transform cameraPose
std_msgs/Byte pipeline
```

- `header`: Standard header
- `targetLocation`: The angular location of the target, assuming that `tx`, `ty`, and `ts` correspond to rotations about the x, y, and z axes respectively
- `hasTargets`: Reflects `tv` value
- `targetArea`: A float on $[0, 1]$ giving the ratio of target area to image area (derived from `ta`)
- `cameraPose`: A `Transform` generated from the solvePnP solution
- `pipeline`: Index of the current pipeline

In addition, a service (`/LED`) was created. This simply accepts an integer on $[0, 3]$ and uses it to set the LED state according to Table 2.

7 Future Work

- Use the pipeline latency value to try to correct the header timestamp to reflect when the image was actually taken
- Attempt to extract values for `tshort`, etc.; these are currently not published in the `status_update` websocket message, and thus can only be read via Network Tables
- Implement services for setting the pipeline, etc.

[6] https://github.com/carlosgj/limelight_ros