Tracking Round Trip Time Latency in the MSO Network

A Technical Paper prepared for SCTE by

Michael Overcash
Principal Engineer
Cox Communications
6305 Peachtree-Dunwoody Rd, Atlanta, GA 30328
404-269-6595
michael.overcash@cox.com

Alan Skinner
Principal Engineer
Cox Communications
6305 Peachtree-Dunwoody Rd, Atlanta, GA 30328
404-269-0845
alan.skinner@cox.com

Owen Parsons
Engineer
Cox Communications
6305 Peachtree-Dunwoody Rd, Atlanta, GA 30328
404-269-4998
owen.parsons@cox.com

Daniel Sciscoe
Network Engineer
Cox Communications
6305 Peachtree-Dunwoody Rd, Atlanta, GA 30328
daniel.sciscoe2@cox.com

Elizabeth Vitale
Network Engineer
Cox Communications
6305 Peachtree-Dunwoody Rd, Atlanta, GA 30328
elizabeth.vitale@cox.com

Table of Contents

List of Figures

List of Tables

1. Introduction

Forget throughput – latency is the new standard of internet quality. Whether it's a glitchy videoconference or a "laggy game," it's increasingly important to know how latency is impacting customers, and how it interacts with the network components we control. In this paper we will describe work done with Raspberry Pi-based test points that measure roundtrip time, jitter and packet loss, using realistic UDP streams. We will also share some of the early data we've collected and discuss what we've learned so far.

2. Lag Overview and Project Motivation

Subscribers use "lag" to describe an aggregation of latency, jitter, and packet loss; a poorly performing service or application is described as "laggy". Historically, subscriber internet use was dominated by HTTP web browsing and streaming protocols that download a video segment at a time – neither of which is particularly sensitive to lag. Today, popular applications like real-time video conference and online gaming are extremely lag sensitive. In addition, the FCC SamKnows program now tracks latency in addition to speed.

CableLabs is addressing this need in the Low-Latency DOCSIS (LLD) program, and vendors are introducing various product features intended to improve lag. But how can operators know if these new products are effective? Lab testing can help of course, but there is no substitute for field loading and actual customer traffic patterns. Many new features over the years have shined in a lab and provided less-than-stellar results once deployed. Likewise, every new product comes with a slew of configuration parameters … are these parameters tuned correctly? Vendors typically provide recommended starting values, and often these values persist into perpetuity without being critically examined to ensure that they're optimal. We can do better, but only if we can measure the results each time we turn a knob.

Just as operators have monitored node utilization for years, operators need a platform to monitor and track lag over time throughout the network. Thus, the Cox LagSpy program was born.

2.1. LagSpy Goals

Cox has the following goals for the program:

- Use realistic UDP streams to measure latency, rather than ICMP pings.
- Ability to distribute test points widely throughout the Access Network.
- No special configuration of subscriber CPE equipment (e.g. no need for port forwards.)
- Upgradable with ability to add new features and test protocols over time.
- Low hardware cost.
- Configurable network utilization.
- Portable software

2.2. LagSpy Overview

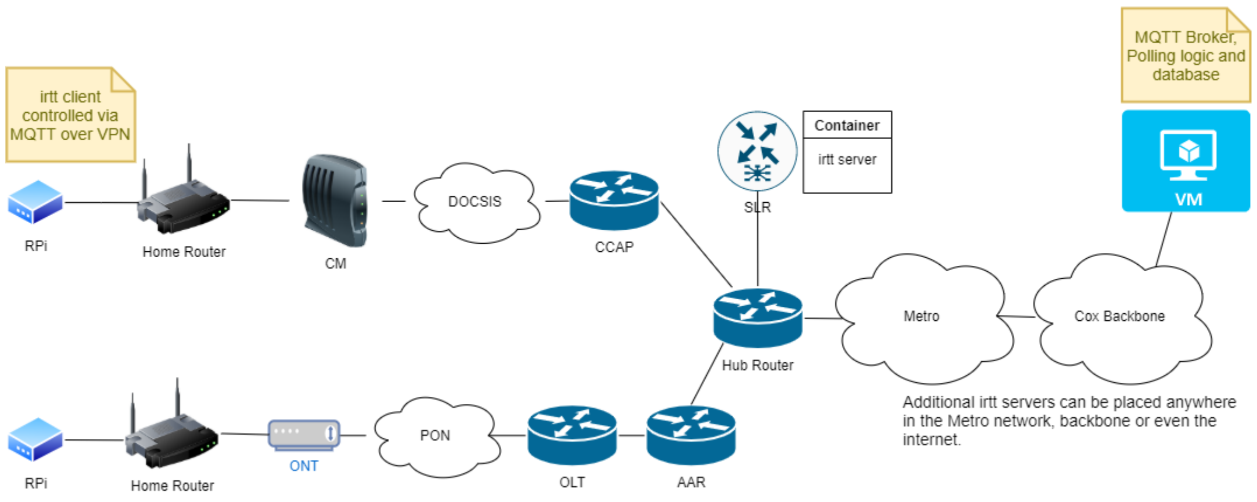

irtt client controlled via MQTT over VPN
RPi
Home Router
CM
DOCSIS
CCAP
SLR
Container
irtt server
MQTT Broker, Polling logic and database
VM
Hub Router
Metro
Cox Backbone
Additional irtt servers can be placed anywhere in the Metro network, backbone or even the internet.
RPi
Home Router
ONT
PON
OLT
AAR

Figure 1 – LagSpy Proof of Concept Architectural Overview

In the proof of concept architecture, Raspberry Pi single board computers were mailed to employee volunteers. These Raspberry Pi devices are nicknamed “Lag-Pis”. Each Lag-Pi implements a network test client called IRTT (Isochronous Round-Trip Time). The Lag-Pis are controlled by a virtual machine on the internet known as the Poller, which also aggregates and stores the test results. The Poller instructs each Lag-Pi when to perform a test and specifies the IRTT server to test against.

The IRTT test involves a client and a server (similar to IPERF). The Lag-Pis implement the client role, and the servers are set up at interesting points in the Cox network. For the proof of concept trial, we are setting up IRTT servers at the Service Layer Router (SLR) connected to the Hub Router. The SLR was selected since neither the Access Router itself (CCAP or OLT) nor the hub router can act as an IRTT server. To minimize latency to the Lag-Pi, we wanted something as close to the access network as possible. We are also setting up IRTT servers at two of our Regional Data Centers (RDCs) for comparison to the SLR.

While our primary focus in the proof of concept is the DOCSIS Network, the architecture is transport agnostic and can run on any access network (DOCSIS, PON, Cellular Data, etc.)

2.3. Why IRTT?

IRTT is a widely available open-source package which generates a customizable UDP stream to measure route trip time and jitter, among other things. Here is an example of an IRTT test and its results:

```
irtt client -i 20ms -l 172 -d 30s --fill=rand --sfill=rand --hmac=0x<redacted> -q irtt-
telemetry.coxlab.net:22112
[Connecting] connecting to irtt-telemetry.coxlab.net:22112
[184.176.185.20:22112] [Connected] connection established
[184.176.185.20:22112] [WaitForPackets] waiting 352ms for final packets

                         Min      Mean    Median       Max   Stddev
                         ---      ----    ------       ---   ------
                RTT  78.92ms   84.55ms   83.55ms   117.3ms   3.17ms
         send delay   -1.24s    -1.23s    -1.23s    -1.21s   2.25ms
      receive delay    1.31s     1.32s     1.32s     1.35s   2.22ms

      IPDV (jitter)   1.93µs    2.32ms    1.13ms   34.77ms   3.15ms
          send IPDV    110ns    1.89ms     925µs   19.11ms   2.41ms
       receive IPDV    754ns     740µs     274µs   34.42ms   2.31ms
     send call time   12.9µs     72µs             932µs  46.5µs
        timer error    100ns    129µs             827µs   107µs
  server proc. time   4.45µs   9.39µs             128µs  4.86µs

                duration: 30.3s (wait 352ms)
   packets sent/received: 1471/1471 (0.00% loss)
 server packets received: 1471/1471 (0.00%/0.00% loss up/down)
     bytes sent/received: 253012/253012
       send/receive rate: 67.5 Kbps / 67.5 Kbps
           packet length: 172 bytes
             timer stats: 28/1499 (1.87%) missed, 0.64% error
```

In this example, the UDP payload size was set to 172 bytes, the inter-packet interval is 20 ms, and the test ran for 30 seconds. The total bandwidth was 67.5 Kbps, approximating a UDP audio stream. The round-trip time was on average 84.55 ms with an average jitter of 2.32 ms. These are the parameters Cox is currently using in the trial, but we are investigating other streams to model (see Section 3.3.2).

IRTT is launched in either client or server mode. The client generates the test traffic, which is reflected back by the server. The client compares the original packet to its reflected version to calculate the performance statistics. In our architecture, the Lag-Pi is acting as the client.

IRTT is extremely easy to deploy. On a Debian/Ubuntu Linux VM, you can install it simply by running `apt-get install irtt`.

2.3.1. UDP versus ICMP for Latency Testing

Many tools (e.g. the Ookla Speed Test) use an ICMP-based tool like ping for latency measurements. We selected a UDP-based tool as we believe this more accurately reports the subscriber experience.

UDP based measurements are more accurate because:

- Different QoS is generally applied to UDP versus ICMP.
- Internally, many devices implement ICMP as a control plane protocol and UDP as a data plane protocol. So for example, many devices employ hardware acceleration for TCP and UDP, but hardware acceleration is rare for ICMP.
- Real applications use TCP or UDP to transmit data. No common applications deliver user data using ICMP.
- Many devices rate limit ICMP handling for DDOS protection. IRTT sends a continuous stream of UDP data, but this is impossible in ICMP. A high ICMP packet rate is often treated as a ping flood DoS attack and will be blocked.

2.4. Why Raspberry Pi?

Raspberry Pis were selected for the pilot based on their low cost and small form factor. They can be cheaply mailed in bubble envelopes. We used the 2GB model of the Raspberry Pi 4B. Note that the networking is significantly improved from the Raspberry Pi 4 versus the 3. Specifically, the Raspberry Pi 4 can support gigabit ethernet speeds while earlier models were throttled by a slow USB 2.0 bus connecting the main SoC to the networking chip. This architecture limited the ethernet throughput to about 300 Mbps.

We will not use Raspberry Pis for large scale deployment. For mass deployment, we will incorporate a LagSpy client into Cox managed gateway devices.

2.5. Employee Pilot

We solicited 19 employee volunteers for the pilot from a DOCSIS working group and mailed pre-configured Lag-Pis to them. The volunteers simply plugged in the power adaptor and connected the Lag-Pis to an ethernet port on their LAN. The volunteers are distributed across multiple Cox markets.

Note that while the Raspberry Pi 4 supports Wi-Fi, we have disabled it and are relying exclusively on gigabit ethernet for LAN connectivity. Wi-Fi introduces a significant amount of latency and jitter, and the focus of this project is on the access network. However in a future project, we could easily measure Wi-Fi latency and jitter in this architecture simply by enabling the Wi-Fi on the Lag-Pi, and setting up an IRTT server on the access point.

3. LagSpy Technical Deep Dive

3.1. Software Architecture

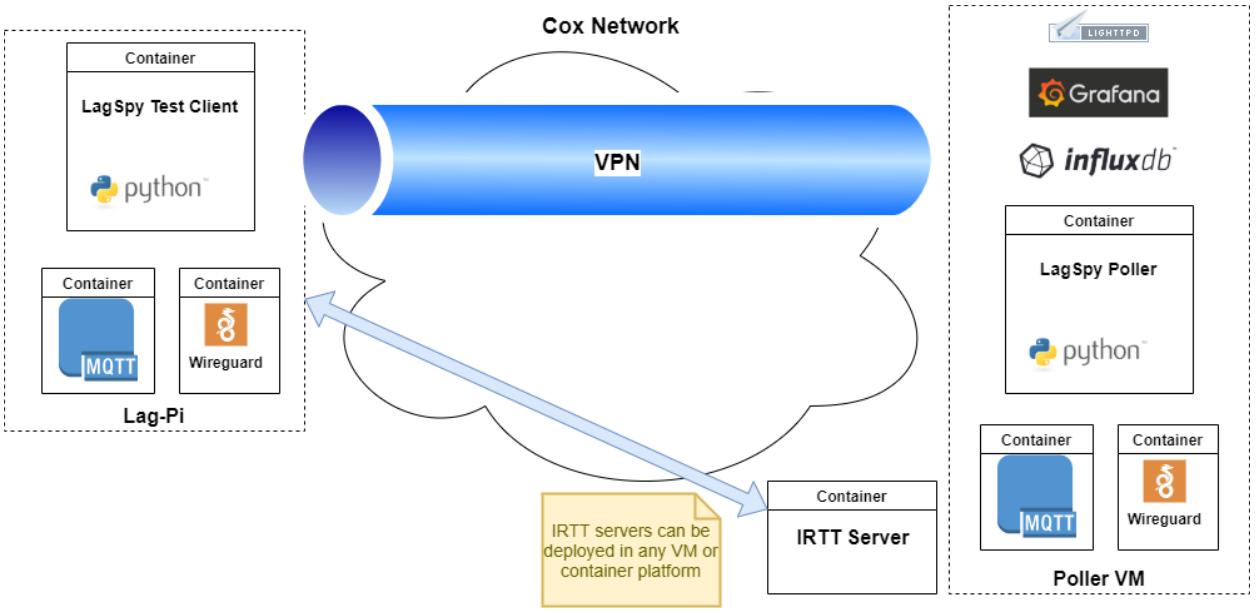


Figure 2 – LagSpy Software Stack

The principal components of the Lag-Pi are:

- The LagSpy Test Client, written in Python 3.8.
- Eclipse Mosquitto to implement an MQTT client.
- Wireguard to establish a VPN connection to the Poller for command and control.

The principal components of the Poller are:

- The Lagspy Poller, written in Python 3.8.
- Eclipse Mosquitto to implement an MQTT broker and localhost client.
- Wireguard for a VPN endpoint.
- InfluxDB to import and aggregate data from the Poller for visualization.
- Grafana for visualization of test results.

- Lighttpd (primarily used to upgrade the Lag-Pi.)

Note that MQTT and Lighttpd are only exposed over the VPN interface over the 10.13.0.0/16 subnet.

The IRTT Servers are implemented in containers, which can be deployed virtually anywhere as IRTT is a simple protocol that only exposes a single UDP port. Docker (and other container frameworks) can strictly limit the resources available to an applications, including memory, CPU, and network sockets, greatly reducing the risk to the platform.

3.1.1. Connectivity Driven by Client

The VPN tunnel is initiated from the subscriber side, meaning that no special configuration is required on the home gateway. The connection is initiated from the LAN side. The Wireguard `PersistentKeepalive` feature prevents the home gateway's NAT state from timing out due to inactivity.

Once the VPN tunnel is established, the Poller can initiate MQTT traffic at will to the Lag-Pi. No port forwarding is required.

Likewise, IRTT traffic is initiated by the Lag-Pi (client) and does not require special forwarding rules.

3.2. Command and Control

MQTT, a popular IOT control protocol, is used to manage the Lag-Pis. MQTT is a Publisher-Subscriber (Pub/Sub) protocol. Clients publish messages to named channels called "topics". Clients can subscribe to any topic of interest. MQTT creates a reliable mechanism to establish both 1:1 communication with individual Lag-Pis, and also to send messages to multiple Lag-Pis at once.

The MQTT Broker coordinates the message forwarding and is implemented on the Poller.

For LagSpy, all message payloads are in JSON format.

Table 1 – LagSpy MQTT Topics

Topic	Arguments	Direction	Description
connect/hello	n/a	Lag-Pi → Poller	Register with poller and keepalive
connect/enroll/<mac>	MAC address of Lag-Pi	Poller → Lag-Pi	Provide VPN credentials to Lag-Pi
connect/link_ok/<mac>	MAC address of Lag-Pi	Poller → Lag-Pi	Keepalive response
irtt/start/<mac>	MAC address of Lag-Pi	Poller → Lag-Pi	Start IRTT test
irtt/results	n/a	Lag-Pi → Poller	Results of IRTT test
iperf/start/<mac>	MAC address of Lag-Pi	Poller → Lag-Pi	Start IPERF3 test
iperf/results	n/a	Lag-Pi → Poller	Results of IPERF3 test

3.2.1. *Wireguard Certificate Enrollment*

The Wireguard VPN requires that each peer have a unique Private Key and a statically assigned IP address in the VPN subnet. As we did not want to manually provision each Lag-Pi with VPN credentials prior to shipment, we implemented a dynamic VPN credential mechanism over MQTT.

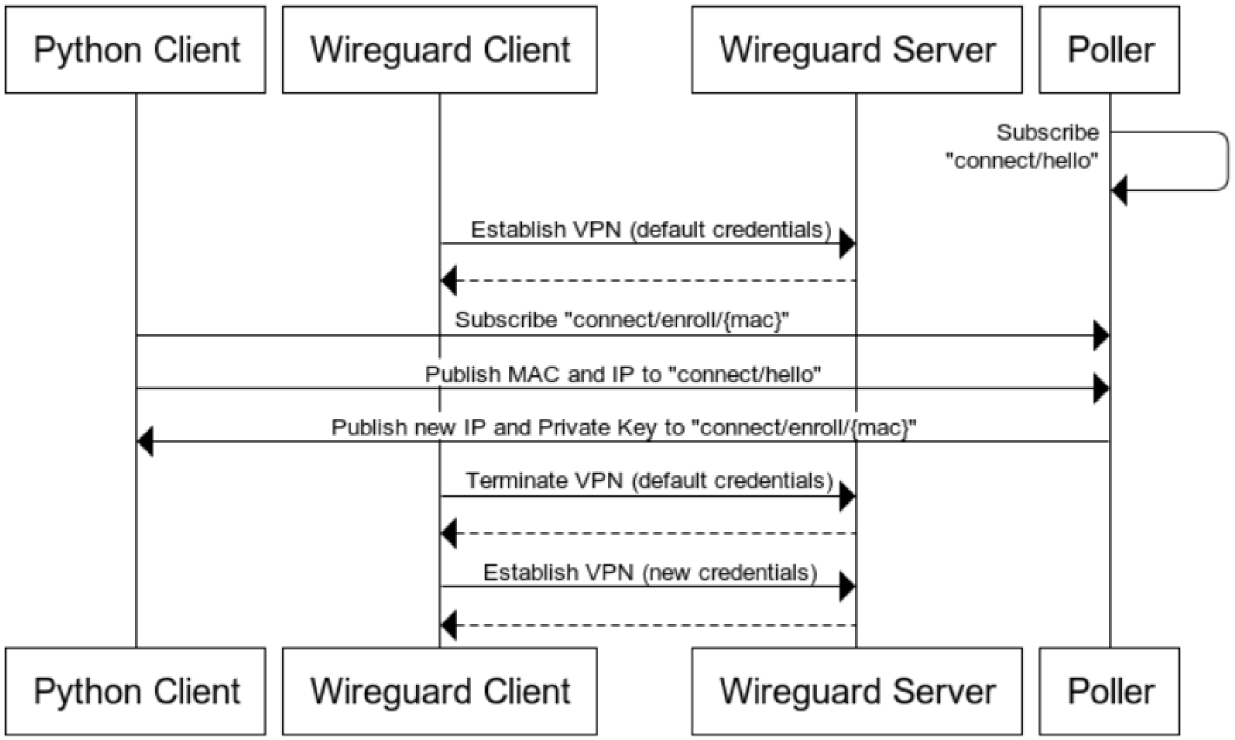

Python Client
Wireguard Client
Wireguard Server
Poller
Subscribe "connect/hello"
Establish VPN (default credentials)
Subscribe "connect/enroll/{mac}"
Publish MAC and IP to "connect/hello"
Publish new IP and Private Key to "connect/enroll/{mac}"
Terminate VPN (default credentials)
Establish VPN (new credentials)
Python Client
Wireguard Client
Wireguard Server
Poller

Figure 3 – Wireguard VPN Enrollment Process

A default set of credentials are used to establish the permanent credentials. Wireguard only allows one endpoint at a time to connect with the default private key. This enrollment process can gracefully handle a small amount of contention for the default credentials, but this algorithm will need to be revisited as we increase scale.

3.2.2. *Network Failsafe Keepalive*

If the Wireguard VPN fails, then the Lag-Pi becomes unmanageable. To mitigate this, each Lag-Pi runs a cron job that verifies the ability to download a small file from the Poller. If the file download fails, the script ensures the Wireguard container is running.

3.2.3. *Debugging with Mosquitto*

MQTT is a pub-sub protocol, meaning that any authenticated party can subscribe to a topic. Therefore a user on the Poller can subscribe to interesting topics for debugging purposes, and sniff the control messages sent over the VPN. This is very useful for troubleshooting.

In the example below, we can observe the “connect/hello” messages received from the Lag-Pis.

```
lagspy@poller-dt1-phx-0:~$ docker exec -it mosquitto mosquitto_sub -t
"connect/hello"

{"publicIpv6": "::", "publicIpv4": "68.109.32.146", "ip": "10.13.0.16", "mac":
"e4:5f:01:3b:18:23", "topic": "connect/hello", "message": "hello", "seqNum":
5393, "timestamp": "2021-07-19 15:27:49.829820"}

{"publicIpv6": "2600:8800:1a1:1a00::b71a", "publicIpv4": "", "ip":
"10.13.0.12", "mac": "e4:5f:01:21:a0:d4", "topic": "connect/hello", "message":
"hello", "seqNum": 2924, "timestamp": "2021-07-19 15:27:54.517621"}
```

3.3. IRTT Deep Dive

3.3.1. Limitations

IRTT attempts to break up the measured Round Trip Time (RTT) into "Send Delay" and "Receive Delay". However this decomposition is based on injecting a timestamp into the test packets, which requires precise time synchronization between the client and server. This doesn't seem to work in any practical environment that we've tested using Raspberry Pis or even Windows PCs. If the RTT is low enough, one of the components will be reported as a negative number. As we are fairly confident that the Cox network does not support time travel, we must conclude that this is due to clock skew between the client and server.

We have attempted to achieve better time sync by using GPS modules without success. We also tried to use NTP, where the NTP peer, the NTP server, and the NTP client were all on the same ethernet switch. This didn't work either.

We suspect the only way to get this to work is to use IEEE 1588 (Precision Time Protocol) with IRTT endpoints that use precision real-time clocks. This is out of scope for the LagSpy program, since our long-term plan is to use consumer grade internet gateways. Integrating LagSpy into a cable modem that implements DOCSIS Time Protocol could provide a way to get this precision.

IRTT sends symmetrical bidirectional test traffic. For example, if IRTT is configured to send 60 kbps upstream, then an identical 60 kbps stream is generated in the downstream direction as well. Unlike iperf3, it is not possible to perform unidirectional testing.

However, due to the nature of the DOCSIS protocol and the use of TDM in the downstream vs. TDMA in the upstream, the major contribution of latency and jitter is in the upstream. Downstream traffic of this nature will simply be forwarded along by the CMTS into the plentiful frames available to each modem's SFID (assuming the modem is below its QoS limit). This generally happens at or very near real time. In the upstream, however, the modem must request a timeslot in a contention-based interval first, then wait for the CMTS scheduler to allocate a timeslot, then wait for the MAP message to arrive, and finally send the data. This process is more sensitive to network congestion and has more possibility of variation due to limited contention intervals. Therefore, we make the assumption that the changes in latency and jitter over time are primarily due to the fluctuating conditions of the upstream.

3.3.2. IRTT Traffic Profiles

IRTT streams are highly customizable, so we can model a number of latency sensitive applications. Currently we are using a profile that simulates an audio stream, but we plan to add more profiles to our testing toward the end of 2021.

CableLabs has studied typical data streams from collaboration apps (see References). Their summary findings are below.

Table 2 – CableLabs Video Confrerencing Bandwidth Summary. Applications were anonymized by CableLabs.

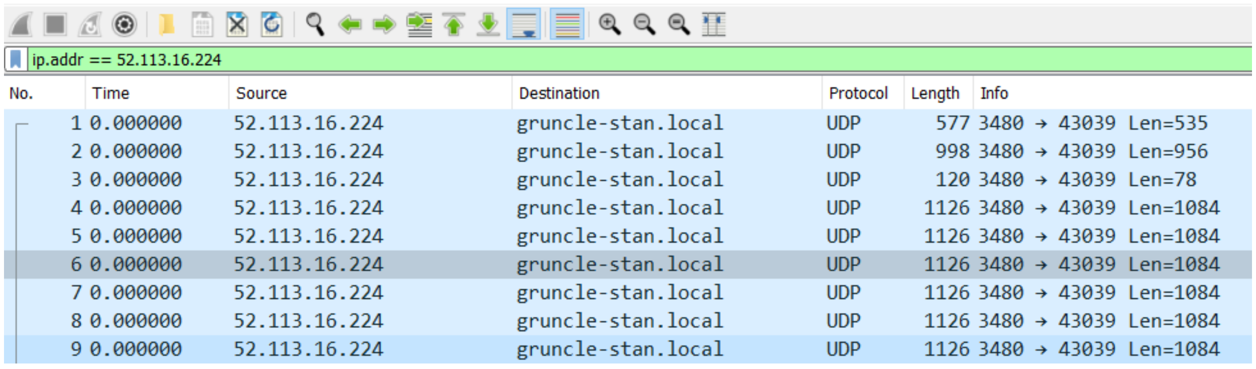
Application	Typical Upstream	Typical Downstream
Application A	< 500 kbps	Between 1 and 2 Mbps
Application B	200 kbps typical; one outlier at 2 Mbps	Approx. 1 Mbps
Application C	Approx. 350 kbps	Approx. 2 Mbps
Application D	Between 200 kbps and about 1.8 Mbps	Cluster at 500 kbps and cluster at 3 Mbps

While video conference traffic is highly asymmetrical, IRTT can only model symmetric streams. We recommend modeling the upstream data rate since most latency and jitter should be introduced in the upstream.

3.3.2.1. *Characterizing Application Streams*

Characterizing application streams is straightforward with a packet sniffer application such as Wireshark.

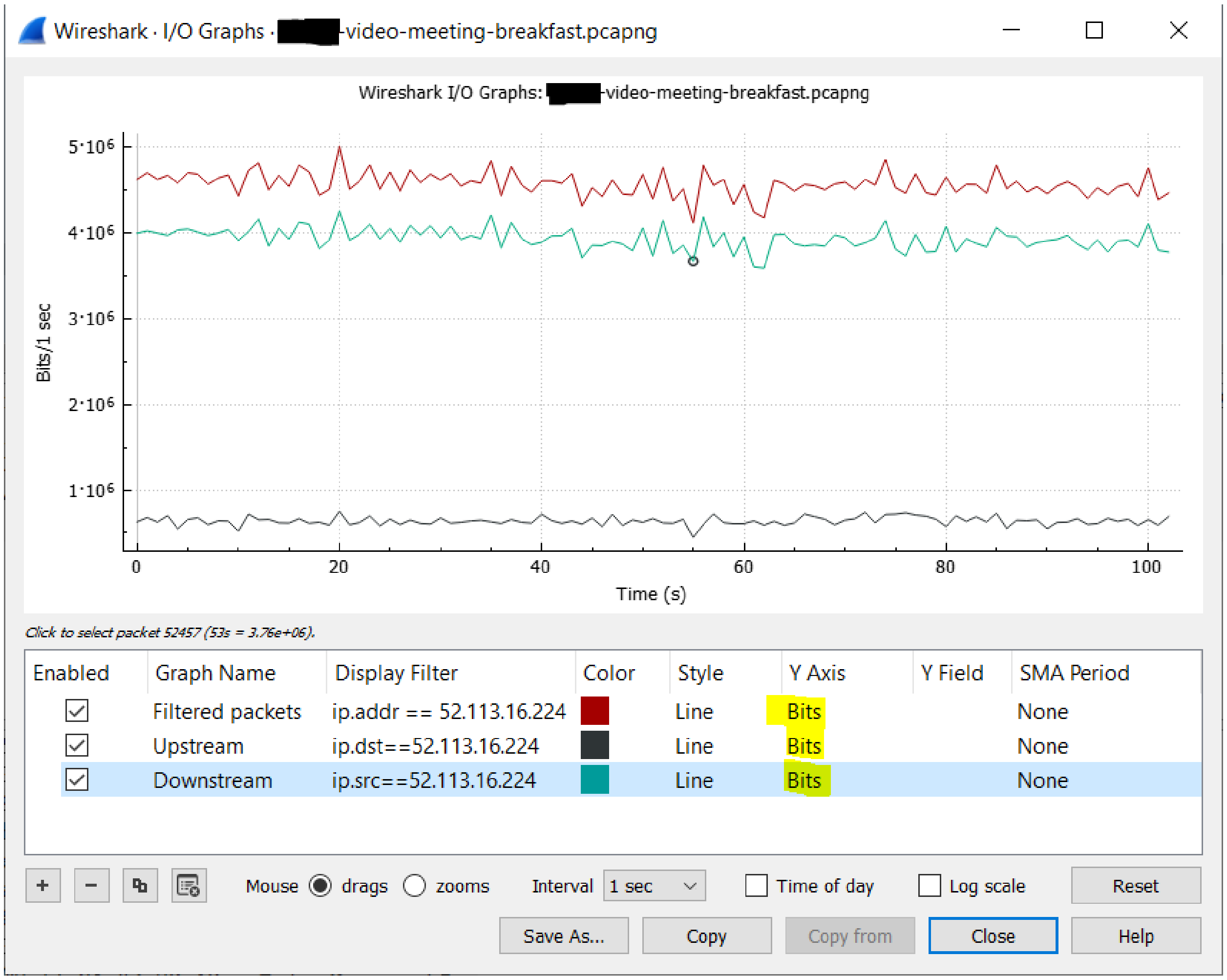
- Optional: place an ethernet switch with a mirror port between the device under test (DUT) and the LAN. These switches are widely available for under $50.
- Launch the app generating the traffic on the DUT.
- Start a Wireshark capture. For apps that run on a computer, you can run Wireshark on the same machine. For something like a game console, the optional ethernet switch must be used. Or, many prosumer/commercial home routers have a sniffer capability.
- Collect data for several minutes, then end the capture.
- Identify the traffic of interest and apply a display filter. The filter can be applied in the upstream, downstream, or bidirectional as desired. In this example:
    - Use `ip.addr==52.113.16.224` for bidirectional.
    - Use `ip.src==52.113.16.224` for downstream.
    - Use `ip.dst==52.113.16.224` for upstream.
    - Take care to filter based on the remote application server (public IP address) rather than the LAN device.



```
ip.addr == 52.113.16.224

No.  Time      Source         Destination        Protocol Length Info
  1 0.000000  52.113.16.224  gruncle-stan.local  UDP       577 3480 → 43039 Len=535
  2 0.000000  52.113.16.224  gruncle-stan.local  UDP       998 3480 → 43039 Len=956
  3 0.000000  52.113.16.224  gruncle-stan.local  UDP       120 3480 → 43039 Len=78
  4 0.000000  52.113.16.224  gruncle-stan.local  UDP      1126 3480 → 43039 Len=1084
  5 0.000000  52.113.16.224  gruncle-stan.local  UDP      1126 3480 → 43039 Len=1084
  6 0.000000  52.113.16.224  gruncle-stan.local  UDP      1126 3480 → 43039 Len=1084
  7 0.000000  52.113.16.224  gruncle-stan.local  UDP      1126 3480 → 43039 Len=1084
  8 0.000000  52.113.16.224  gruncle-stan.local  UDP      1126 3480 → 43039 Len=1084
  9 0.000000  52.113.16.224  gruncle-stan.local  UDP      1126 3480 → 43039 Len=1084
```

- Go to Statistics→IO Graphs in Wireshark. Set up Display Filters for upstream, downstream, and bidirectional. You can inspect both bitrate and packet rate by selecting the appropriate option for the Y Axis.


Wireshark I/O Graphs: [redacted]-video-meeting-breakfast.pcapng
5·10⁶
4·10⁶
3·10⁶
2·10⁶
1·10⁶
Bits/1 sec
0
20
40
60
80
100
Time (s)

Wireshark · I/O Graphs · [redacted]-video-meeting-breakfast.pcapng

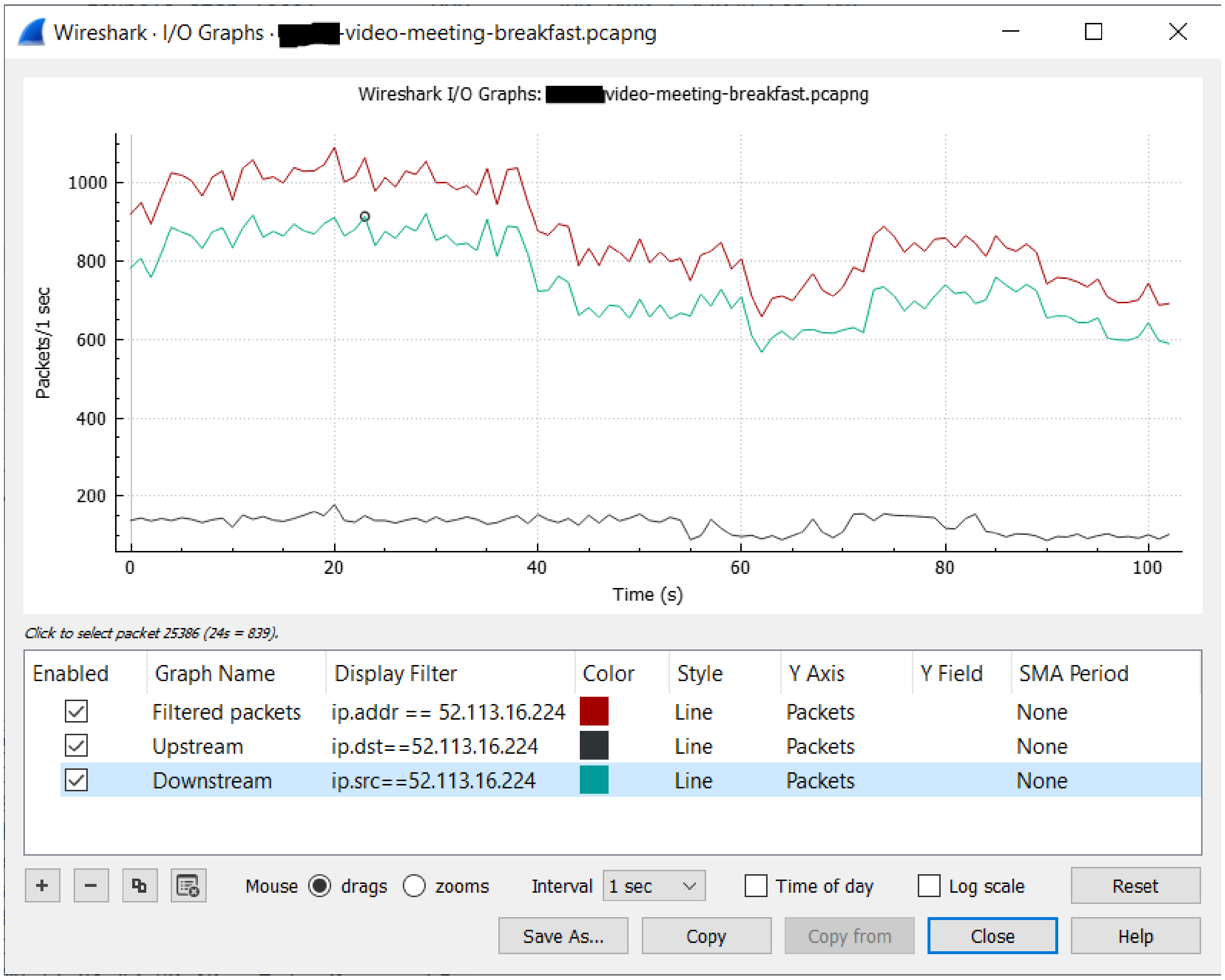
Click to select packet 52457 (53s = 3.76e+06).

Enabled	Graph Name	Display Filter	Color	Style	Y Axis	Y Field	SMA Period
✓	Filtered packets	ip.addr == 52.113.16.224	🟥	Line	Bits		None
✓	Upstream	ip.dst==52.113.16.224	⬛	Line	Bits		None
✓	Downstream	ip.src==52.113.16.224	🟩	Line	Bits		None

Mouse [✓ drags] zooms Interval 1 sec Time of day Log scale Reset

Save As... Copy Copy from Close Help

Figure 4 – Example video conference data rate. Note that "Bits" is selected on the Y Axis, and the Interval is set as 1 second. This is an asymmetric stream, with a typical upstream rate of 500 kbps and a typical downstream rate of 4 Mbps.


Wireshark · I/O Graphs · [redacted]-video-meeting-breakfast.pcapng
Wireshark I/O Graphs: [redacted]video-meeting-breakfast.pcapng
1000
800
600
400
200
Packets/1 sec
0
20
40
60
80
100
Time (s)
+
−
Save As...
Copy
Copy from
Close
Help
Reset

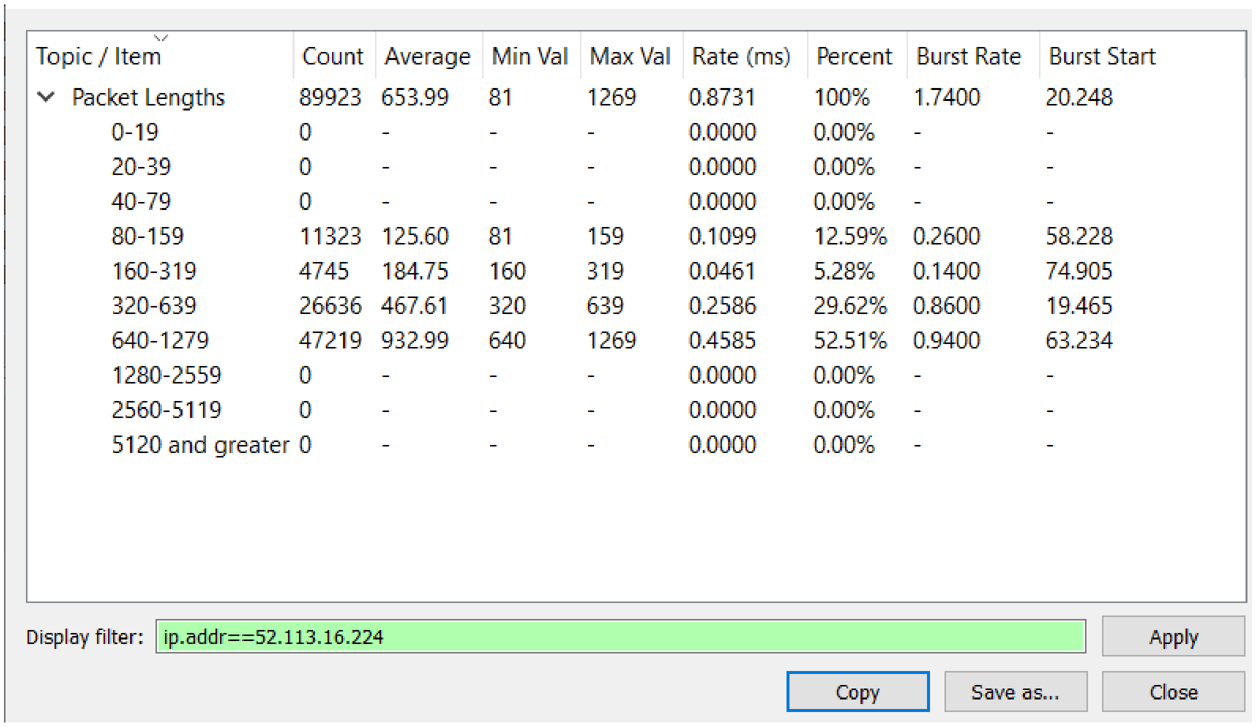
Click to select packet 25386 (24s = 839).

Enabled	Graph Name	Display Filter	Color	Style	Y Axis	Y Field	SMA Period
✓	Filtered packets	ip.addr == 52.113.16.224	🟥	Line	Packets		None
✓	Upstream	ip.dst==52.113.16.224	⬛	Line	Packets		None
✓	Downstream	ip.src==52.113.16.224	🟩	Line	Packets		None

Mouse [✓ drags] zooms   Interval 1 sec   ☐ Time of day   ☐ Log scale

Figure 5 – Example video conference packet rate. Typical upstream rate is 100 pps, and typical downstream rate is 800 pps.

- Inspect the packet length in Wireshark using Statistics→Packet Lengths. Apply the desired Display Filter.



Topic / Item	Count	Average	Min Val	Max Val	Rate (ms)	Percent	Burst Rate	Burst Start
Packet Lengths	89923	653.99	81	1269	0.8731	100%	1.7400	20.248
0-19	0	-	-	-	0.0000	0.00%	-	-
20-39	0	-	-	-	0.0000	0.00%	-	-
40-79	0	-	-	-	0.0000	0.00%	-	-
80-159	11323	125.60	81	159	0.1099	12.59%	0.2600	58.228
160-319	4745	184.75	160	319	0.0461	5.28%	0.1400	74.905
320-639	26636	467.61	320	639	0.2586	29.62%	0.8600	19.465
640-1279	47219	932.99	640	1269	0.4585	52.51%	0.9400	63.234
1280-2559	0	-	-	-	0.0000	0.00%	-	-
2560-5119	0	-	-	-	0.0000	0.00%	-	-
5120 and greater	0	-	-	-	0.0000	0.00%	-	-

Display filter: ip.addr==52.113.16.224

Figure 6 – Example video conference packet size. Typical packet size is 933 bytes. Since upstream and downstream histograms were very similar, only the bidirectional analysis is shown here.

From this analysis and focusing on the upstream, this application can be modeled in IRTT using the following parameters:

- Interval (`-i` parameter): 1 / 100 pps = 10 ms
- Length (`-l` parameter): 933 bytes – 28 bytes = 908 bytes. Note that we subtract the IP and UDP header size to calculate this parameter.

As a sanity check, IRTT reports a throughput of 733.5 Kbps, which is reasonably close to the observed value of approximately 500 Kbps.

3.4. Security Considerations

3.4.1. Lag-Pi (IRTT Client)

The Raspberry Pi is not a hardened hardware platform (e.g. it does not have secure boot or secure storage.) We are only using Raspberry Pis for a limited employee trial and will migrate to managed and secured hardware in the next phase.

The LagSpy application does not listen on any open ports. All command and control traffic is secured over a Wireguard VPN tunnel.

3.4.2. Poller

The Poller is listening on a Wireguard server port. All other services (MQTT, HTTP) are restricted to the VPN interface. In other respects, the same hardening considerations apply to the Poller as any standard server with an open port.

3.4.3. IRTT

The IRTT application is written in Golang and is not subject to the buffer overflow-based attacks that plague C/C++ applications.

By default, the IRTT server listens on UDP port 2112 and is discoverable using standard network scanning techniques (including Shodan.io). At a minimum, this allows an attacker to waste network resources by sending test traffic to the server, and could be the basis for a reflection attack.

To mitigate this risk, we recommend the following:

- Override the default port number
- If possible, disable IPv4 on the IRTT server and use IPv6 for IRTT testing. The IPv6 address space is sparse and more difficult to scan.
- Use the IRTT `-hmac` option to enable the HMAC feature. When enabled, the server will not establish a connection or otherwise respond to client traffic that doesn't use the same HMAC value. This means that the port will appear to be blocked by a firewall (filtered) in a standard nmap scan.

3.4.4. SLR

Cox utilizes routers at each hub site that are dedicated to hosting ancillary (non-data path) services. These routers are known as Services Layer Routers (SLR) and are deployed in pairs, directly connected to metro hub routers. The SLR is an appealing place to host an IRTT server because:

- The Docker implementation on the SLR imposes resource limits to mitigate against DoS and other resource-based attacks.
- The underlying router platform supports access controls that block traffic arriving on the IRTT address from impacting the other router functions.

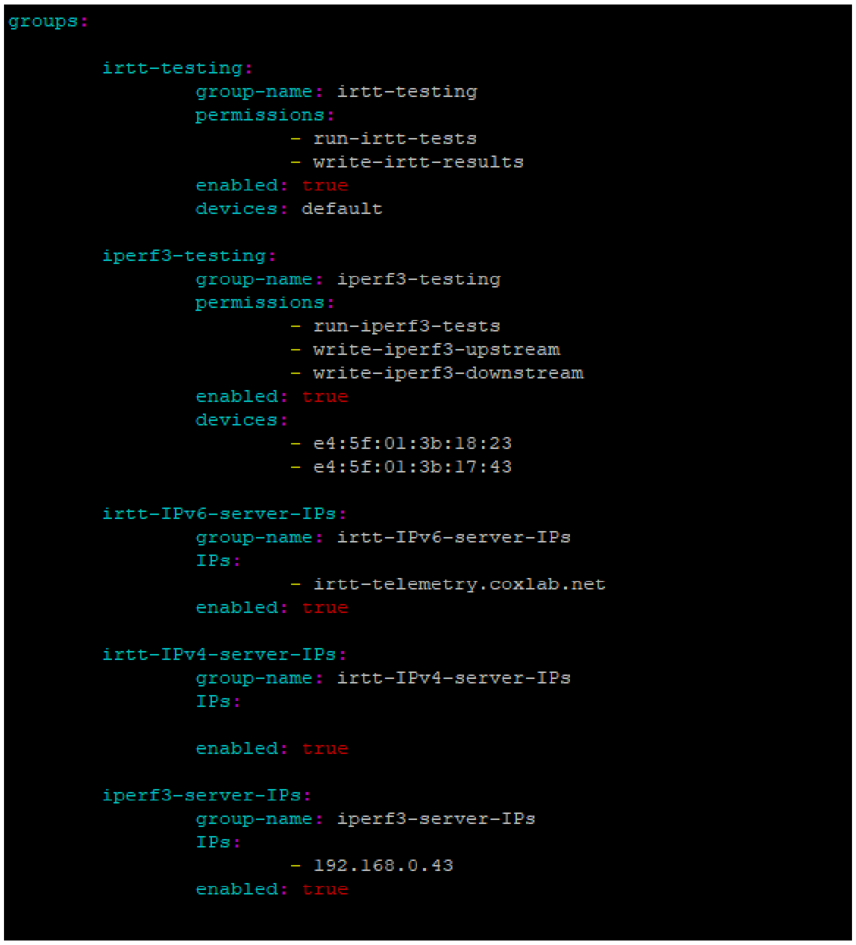
3.5. Test Policy Configuration

Different Lag-Pis operate in different environments and some populations need to execute differents tests. Thus there is a need to implement a flexible policy framework to control how and when tests are executed on a given Lag-Pi.

Our current policy file subdivides testing into 5 different groups, as shown in Figure 7. Our two main group categories are testing groups and server groups. Testing groups specify tests for a member device to run and the server groups provide addresses of available servers.

For testing groups, each group has a list of devices with defined mac addresses. The devices with those mac addresses will run specific tests and write those results to a file in accordance with the permissions of the group. The irtt-testing group sets the mac address as "default", assigning all devices to the `irtt-testing` group automatically; this will require any connected devices to perform irtt tests. In contrast, the `iperf3-testing` group requires that devices with mac addresses defined in the group, such as "ef:5f:01:3b:18:23", execute iperf3[1] tests in addition to irtt tests.

For server groups, each group contains a list of IP addresses running case-specific servers. These groups allow devices within the testing groups to obtain a list of Ips to run their tests against. For example, the `irtt-IPv4-server-Ips` group contains IP addresses running an IPv4 irtt server. As a result, when a device using IPv4 in the irtt testing group is looking for a server to run a test against, it would get the necessary IP from the `irtt-IPv4-server-Ips` group. Likewise, the `iperf3-server-Ips` group contains a list of Ips running an iperf3 server and the `irtt-IPv6-server-Ips` group contains Ips running an IPv6 irtt server.



```
groups:

        irtt-testing:
                group-name: irtt-testing
                permissions:
                        - run-irtt-tests
                        - write-irtt-results
                enabled: true
                devices: default

        iperf3-testing:
                group-name: iperf3-testing
                permissions:
                        - run-iperf3-tests
                        - write-iperf3-upstream
                        - write-iperf3-downstream
                enabled: true
                devices:
                        - e4:5f:01:3b:18:23
                        - e4:5f:01:3b:17:43

        irtt-IPv6-server-IPs:
                group-name: irtt-IPv6-server-IPs
                IPs:
                        - irtt-telemetry.coxlab.net
                enabled: true

        irtt-IPv4-server-IPs:
                group-name: irtt-IPv4-server-IPs
                IPs:

                enabled: true

        iperf3-server-IPs:
                group-name: iperf3-server-IPs
                IPs:
                        - 192.168.0.43
                enabled: true
```

Figure 7 – Policy file containing different testing and server groups.

[1] Iperf3 testing is not part of the core LagSpy functionality, but we are leveraging the LagSpy command and control framework to automate some iperf3 testing we are doing for product acceptance.

3.5.1. Server Autodiscovery

To avoid the need to manually assign each Lag-Pi to the closest server, we are implementing a simple autodiscovery algorithm to find the nearest server. Once per boot, the Lag-Pi will perform a traceroute to each server in the pool and will select the closest (by hops) for IRTT tests.

3.6. Data Visualization

To gather and present the metrics received from MQTT we have a three-part architecture. First, the metrics sent over the MQTT `irtt/results` topic activate a Python listener that decodes them, normalizes the data, then sends each measurement to an InfluxDB instance. Second, InfluxDB collects the data and applies any active retention policies, then makes the data available for consumers.

Lastly, we assembled a Grafana Dashboard to display a subset of the measurements returned by IRTT. Thanks to both Grafana and the InfluxDB query language (Flux), we're able to not only display the metrics as returned but also partition and display them however we wish. This includes building composite metrics, windowing, and aggregation across the whole metric population to make trends clearer or weed out noise that may be present in the data set.

Our current visualization tools plot results for individual Lag-Pi test points. As we increase scale and deploy more test points, we will need to develop tools to analyze the data in aggregate.

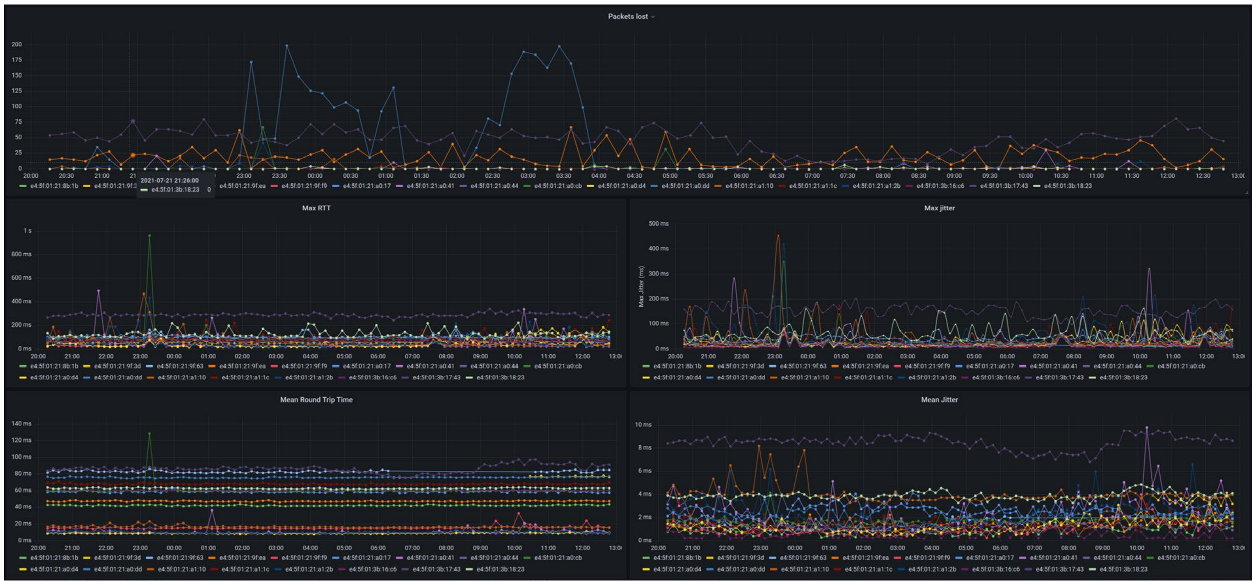


Figure 8 – Grafana Dashboard illustrating: Packet Loss, Max Round Trip Time, Max Jitter, Mean RTT, and Mean Jitter

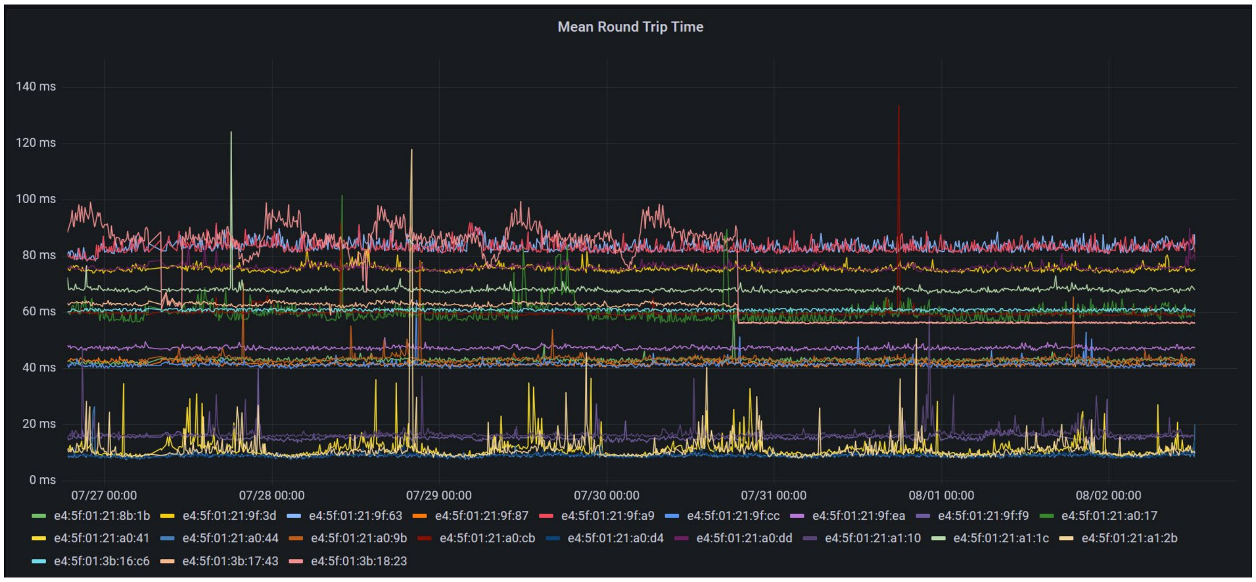


Figure 9 – Mean round trip time over several days of data. Note the cyclic increases for some devices (e.g. the yellow line near the bottom.)

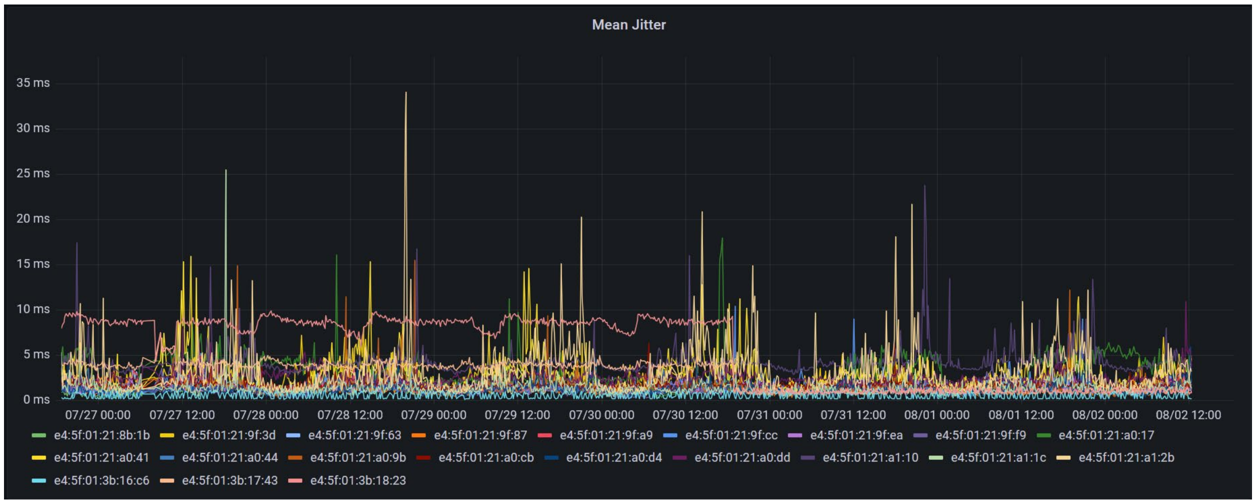


Figure 10 – Mean jitter over several days of data. Again cyclic behavior is evident for a few Lag-Pis.

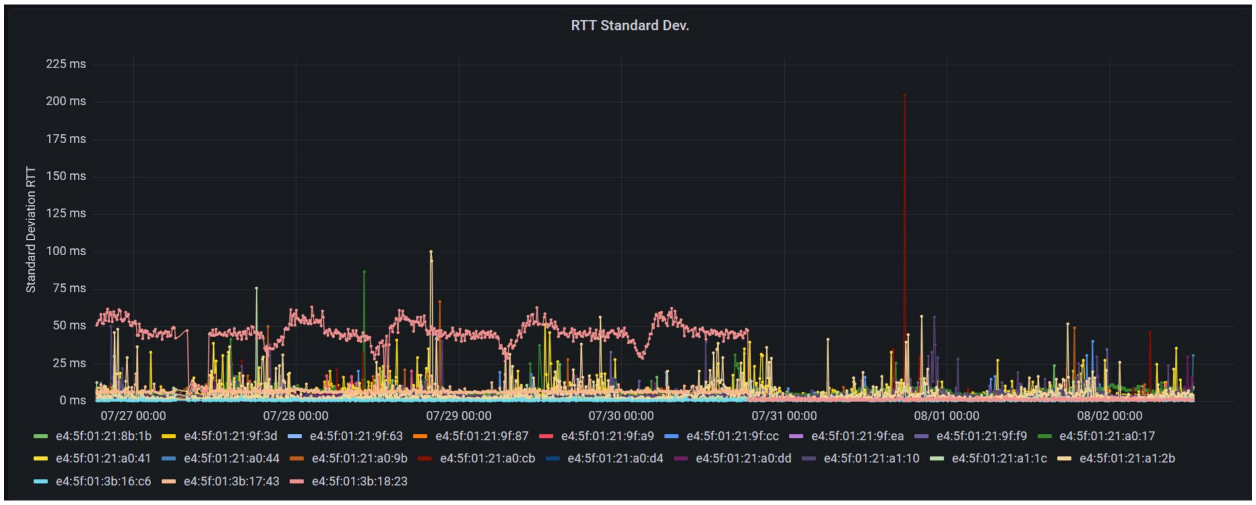


Figure 11 – Round trip time standard deviation across several days of data.

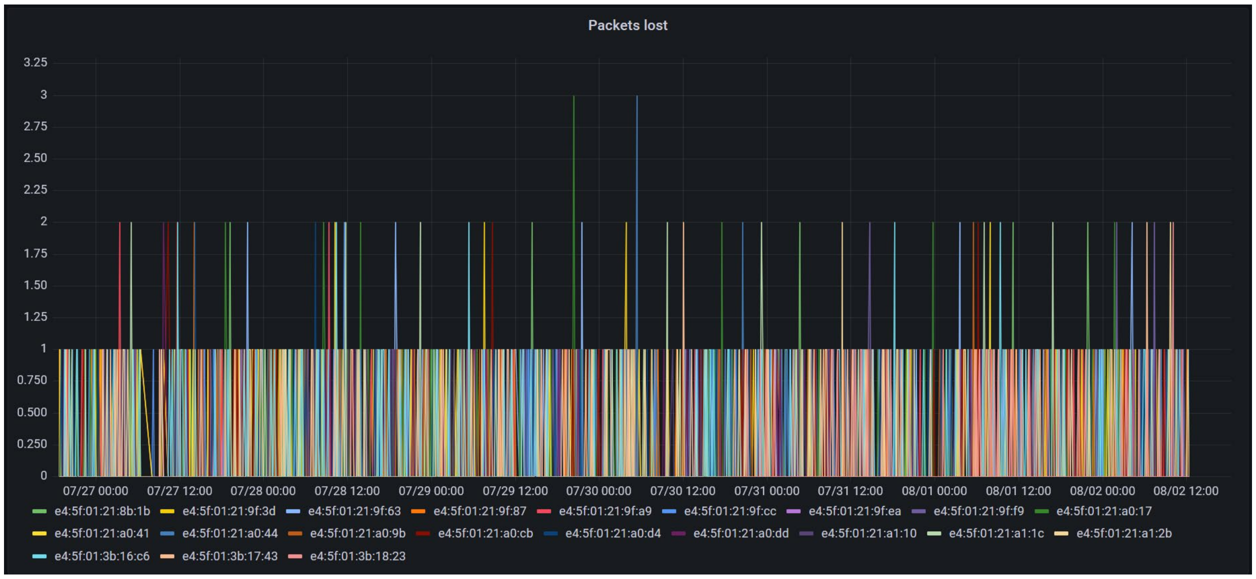
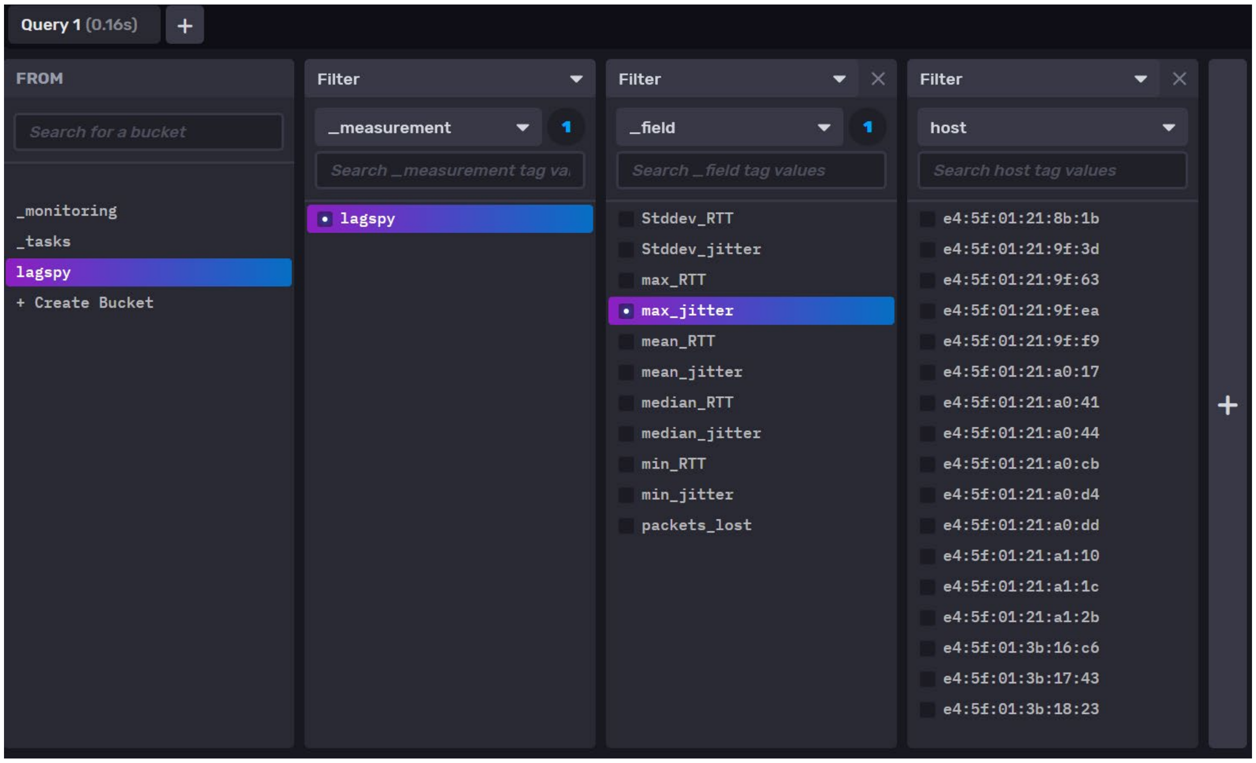


Figure 12 – UDP packet lost during test. Over this test interval, at most 3 packets were lost (out of 1500 sent during the test.)



Query 1 (0.16s)

FROM
Search for a bucket
_monitoring
_tasks
lagspy
+ Create Bucket

Filter
_measurement 1
Search _measurement tag va
lagspy

Filter
_field 1
Search _field tag values
Stddev_RTT
Stddev_jitter
max_RTT
max_jitter
mean_RTT
mean_jitter
median_RTT
median_jitter
min_RTT
min_jitter
packets_lost

Filter
host
Search host tag values
e4:5f:01:21:8b:1b
e4:5f:01:21:9f:3d
e4:5f:01:21:9f:63
e4:5f:01:21:9f:ea
e4:5f:01:21:9f:f9
e4:5f:01:21:a0:17
e4:5f:01:21:a0:41
e4:5f:01:21:a0:44
e4:5f:01:21:a0:cb
e4:5f:01:21:a0:d4
e4:5f:01:21:a0:dd
e4:5f:01:21:a1:10
e4:5f:01:21:a1:1c
e4:5f:01:21:a1:2b
e4:5f:01:3b:16:c6
e4:5f:01:3b:17:43
e4:5f:01:3b:18:23

Figure 13 – InfluxDB Fields and Query Generator

3.7. Resource Usage

Application CPU and memory usage is minimal on the Lag-Pi. When idle, the CPU is 99.7% idle and there are 1.372 GB RAM free (on a 2 GB device). During an active IRTT test, 4.3% CPU is consumed by the IRTT process.

Table 3 – Application Resource Usage

Component	% Memory	Virtual Memory
Docker overhead (containerd, dockerd)	6.2%	3.4 MB
Python Test Client	0.6%	52 KB
IRTT Client	0.1%	879 KB

LagSpy is a lightweight application, especially on a platform that already has Docker running. This supports the long term goal of contributing LagSpy to a Linux-based gateway stack such as RDK-B or DD-WRT.

Note that Wireguard resource usage is primarily in kernel space and can't be easily measured.

4. Conclusion

This has been a project of discovery for our team, and we have learned the following lessons thus far:

- Approximately 50% of our Lag-Pi are deployed in an IPv4-only environment. Since Cox has enabled IPv6 for years, this was surprising to us. Many home routers still disable IPv6 by default.

- The Wireguard VPN implements an effective NAT keepalive using the `PersistentKeepalive` keyword. However this keepalive is disabled by default.
- There is a lot of variation in home network configuration, especially among our enthusiastic engineering volunteers. The sooner this functionality can be integrated into the home gateway, the better.
- The ability to remotely upgrade the Lag-Pi software has been critical even at this very early phase.
- Never underestimate the number of hurdles to jump through (e.g. security reviews) when setting up a server with a public IP address, no matter how trivial the service.
- There are still routers out there with outbound firewall policies.
- There is a software compatibility issue with NOOBS 3.5 and the Raspberry Pi 4B impacting about 10% of devices. Avoid this issue by installing Raspberry Pi OS directly onto the SD card.

Abbreviations

CCAP	Converged Cable Access Platform
CM	Cable modem
CMTS	Cable modem termination system
CSV	Comma Separate Value
DD-WRT	DresDeren-Wireless Router
DOCSIS	Data-Over-Cable System Interface Specification
DoS	Denial of service
DUT	Device under test
GB	Gigabyte
GPS	Global Positioning System
HMAC	Hash-based Message Authentication Code
IRTT	Isochronous Round-Trip Tester
IP	Internet Protocol
IPv6	Internet Protocol Version 6
IRTT	Isochronous Round-Trip Tester (open source application)
KB	Kilobyte
LAN	Local area network
LLD	Low Latency DOCSIS
Mbps	Megabit per second
MB	Megabyte
MQTT	Message Queuing Telemetry Transport
NOOBS	New out of the box software
NTP	Network Time Protocol
OS	Operating System
RAM	Random Access Memory
RDK	Reference Design Kit
RTT	Round Trip Time
SD card	Secure Digital card
SLR	Services Layer Router
SoC	System on a chip

UDP	User datagram protocol
USB	Universal serial bus
VM	Virtual Machine
VPN	Virtual Private Network
WAN	Wide area network

Bibliography & References

IRTT (Isochronous Round-Trip Tester) README.md, GitHub, https://github.com/heistp/irtt

Expanded Testing of Video Conferencing Bandwidth Usage Over 50/5 Mbps Broadband Service, CableLabs Inform[ed] blog, February 19, 2021, https://www.cablelabs.com/expanded-testing-of-video-conferencing-bandwidth-usage-over-50-5-mbps-broadband-service