AutoPerformance
Network Benchmarking for Power Users

Craig Odell, Cody Hanson
CS 526, May 8th, 2012

Abstract:

A common issue for internet subscribers is verifying that the level of service they are paying their provider for is being met. Free to use websites exist which allow users to benchmark their connections. However, when using these solutions, users lack end to end control of the benchmark, and are provided with only limited data about the performance of their connection. It is also hard to schedule tests to run automatically to collect data on performance trends.

In this paper, we describe the AutoPerformance benchmarking tool that we have developed that alleviates some limitations of free to use network connection benchmarking websites. AutoPerformance software is deployed on two different Linux endpoints to test the IP connection between them. The core library used to take the measurements provides rich throughput and delay information for both UDP and TCP modes of transport. Data is collected according to a user configurable schedule and can be plotted with a convenient to use graphing utility that is included.

Introduction:

The internet connections in residential homes are as important today as the telephone has been for the past century. The increasingly connected culture that we live in has an ever greater demand for bandwidth. Everyone knows what it is like to try and work or play on a slow or underperforming internet connection. It is especially frustrating when an internet subscriber feels that they are not getting the level of service for which they paid. How can one tell if one's ISP is keeping their end of the contract? Sure, a user could see that their applications are running slower than they should be, or their data transfer speeds are sluggish, but a more quantitative approach would be better.

Some prior methods existed before we set out to create a better tool. There are several webpages [1] provided by independent companies, or the service providers themselves, that helps one to benchmark one's Internet connection. Those pages have several disadvantages.

- There is no way to schedule recurring tests, and trend data over time.
- The objectivity of the tests is not guaranteed because the administration and the source code for the tests is unknown.
- Data that is provided may be limited to a simple TCP throughput measurement, or an average round trip time using ICMP Echo requests (ping).

We wanted to create a more powerful tool that would be under complete control of the user, end to end. The result was the AutoPerformance tool. Below is a short summary of the features.

- Simultaneous bi-directional TCP and UDP throughput testing, with rich delay, duplication, and sequencing statistics.
- Compatible with Linux, over wired or wireless network interfaces.
- CRON-like interface allows user to schedule how frequently the tests should be run and at what times they should be run.
- Daemonized Python processes allow for headless, background, operation.
- Graphing utility can be used to easily generate charts from raw output benchmark data.

Design Implementations:

Measurement Engine – Thrulay:

Thrulay [3] is an open source program written in C that provides the core measurement functionality for AutoPerformance. Thrulay is a piece of measurement software that works in a client-server model testing TCP and UDP metrics such as throughput, latency, jitter, loss, sequencing, duplication, etc. The metrics that Thrulay provides and that are consumed by AutoPerformance are discussed in more detail in a later section. The typical use case for Thrulay would be to start the server and connect to it from the client using the compiled binaries.

AutoPerformance Client:

The AutoPerformance Client is the master and initiator of the sequence of steps leading to a full bi-directional performance test. This client is configured using the provided configuration from the user which includes items as follows:

- Duration – how long the performance test will run
- Frame Size – how large of frames to utilize
- DSCP – replacing TOS (could be used to test for differentiated services)
- Number of Streams – the number of data streams the underlying performance test should utilize (to better represent real traffic)
- UDP Rate – the specified rate at which UDP performance test should run
- Host – the targeted far-end server name or ip address
- Port – the port number the test should be run upon (not that port+1 and port+2) will also be used.
- Minute – the minute in an hour to run
- Hour – the hour 0 – 24 to run
- Day – the day of the month to run
- Month – the 1-12 month to run
- Day of the Week – the day of the week 1 – 7 to run

Note: that if one wants to run every hour or every month, etc. they can use the text "allMatch" which is similar functionality to "*" in CRON-TABs

Using this provided configuration, the client can then aggregate a rich set of metrics for network trending. A user would start the server using the following command

Autoperf.py –c start <config-file>

To stop the client and kill the daemonized process:

Autoperf.py –c stop

Note that the usage can be viewed when running autoperf.py without any arguments.

AutoPerformance Server:

The AutoPerformance Server is designed to be passively listening on the desired server. For AutoPerformance to work, a user would start the server providing configuration in a similar manner to that passed to the client:

Autoperf.py –s start <config-file>

A user can stop the server as well using the following command:

Autoperf.py –s stop <config-file>

Note that usage for the server will also be displayed when incorrect arguments are specified. Upon starting and daemonizing the server will listen on the configured port and wait for client connections. The listening TCP server is a threaded TCP server that will start threads for each connection accepted. The thread then follows the commands of the client performing the necessary steps needed to carry out a performance test. The server can be configured by specifying the following items:

- Host – the IP address the server should bind to
- Port – the port on which the server should listen

Upon accepting a client request and starting a servicing thread, the performance tests will run the Thrulay client/server on the specified port+1 and the specified port+2. This allows for servers to bind to and listen on ports that are not being used. Consequently, this means that UDP and TCP ports must be forwarded (in NAT configurations) for the configured port, the configure port+1, and the configured port+2.

Test Scheduling:

In order to provide the user with flexibility and control, the scheduling of a performance test is modeled off of a Linux CRON-TAB. CRON Tabs allow for specification of minutes, hours, days, months, days of week to run some command. Similarly, the configuration of the client allows for a user to specify the same parameters. This is done using via a Cron class that extends the Daemon class. The Cron class contains a list of preconfigured events that run at the specified times. When told to run, the Cron object will check if there are any items to run at the current time and run all the events matching the current date/time. When events are finished running and/or there are no events, the Cron object process will sleep for the remaining time until the next minute. For example if it spent 10 seconds performing operations, the cron process will sleep for 50 seconds until the next minute quantum arrives.

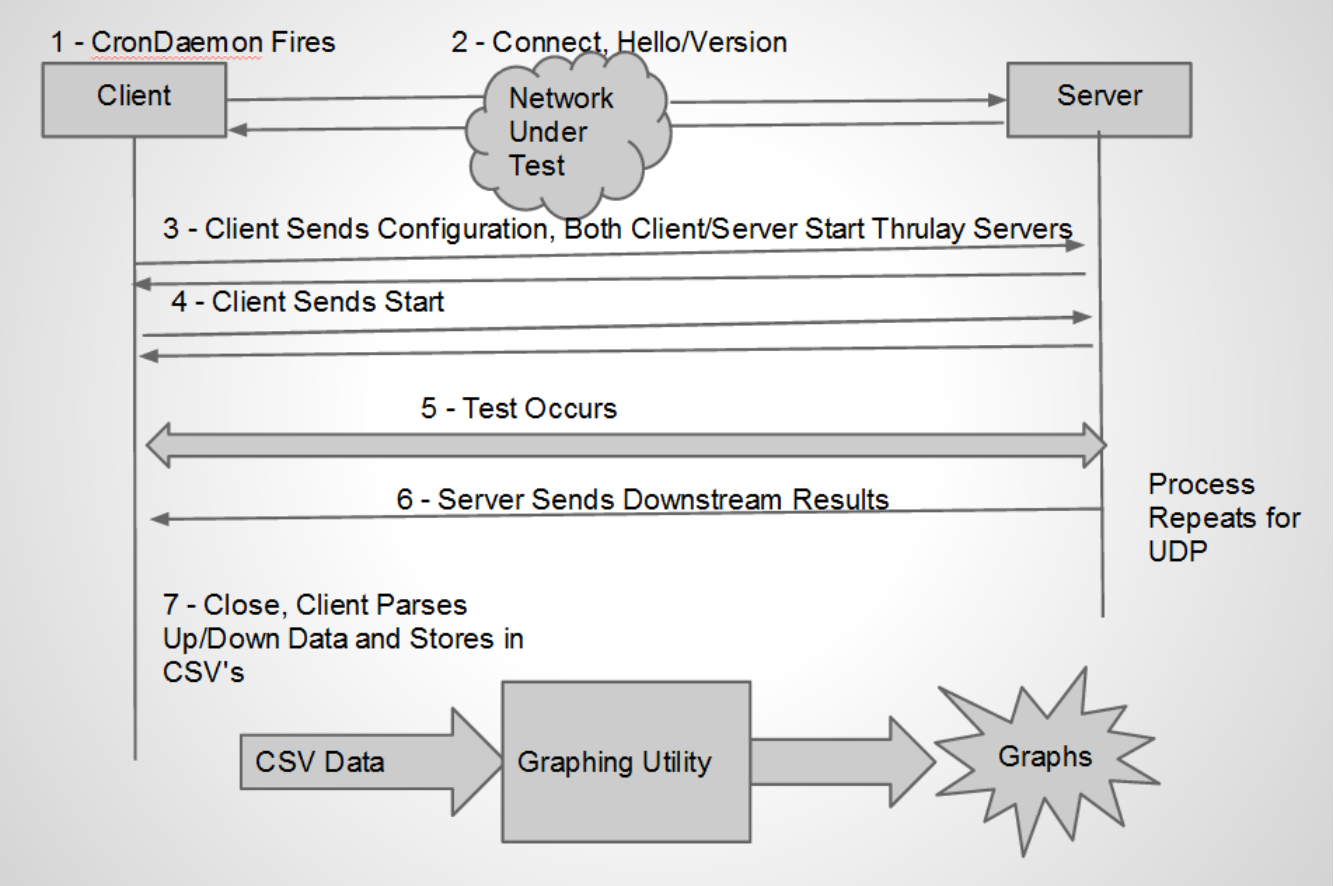
Using the discussed mechanism, we can provide CRON-like functionality to the user and allow them to powerfully specify when they perform their performance tests.

Running AutoPerformance as a Daemon Process:

As previously mentioned, both the client and the server processes run in a daemonized state. This allows for headless background information gathering and allows for users to perform other tasks without being concerned with the foreground processes. The daemonization takes place using the typical double-fork method for creating dameons. This mechanism is implemented in the Daemon class which the client and server inherit from.

AutoPerformance Test Flow:

The AutoPerformance test flow is as follows where sequence of items are numbered:



Graphing Utility - Matplotlib and Python:

Visualizing data from the throughput tests is essential to easily spot trends and gain useful insight. In order to provide this, another part of our AutoPerformance system is the graphing utility. The AutoPerformanceGrapher is a command line utility which is written in Python, and uses the freely available Python graphing library Matplotlib [2]. The utility takes the CSV files that are the output of AutoPerformance and will generate several graphs for the different metrics.

Network Performance Data:

TCP Metrics:

- **Throughput** - Measures in Megabits per second the transfer rate of the TCP stream. TCP will throttle itself to make best use of the available bandwidth between the two endpoints.

- **Round Trip Time** - Measures the network transit time for a communication path from sender to reciever, and back to sender.

- **Jitter** - Also known as delay variation, jitter shows the difference in arrival times for packets in a stream. A low jitter number is important for a connection to carry voice and video streams that are not degraded or choppy.

UDP Metrics:

- **Throughput** - Measured in Megabits per Second, this throughput measurement is a bit different than the TCP variety. For AutoPerformance, one specifies what rate to send UDP packets at, and it will blast a stream at that specified rate. For this reason, graphs of this metric should always show a flat line at the level that it has been configured.

- **Loss Percentage** - While one may be blasting out a UDP stream at 40 Mbps, one's network connection may only be able to handle some percentage of that. Loss Percentage measures what percentage of packets sent to the remote host were actually recieved. Combining the loss percentage with the UDP throughput, one can compute an effective UDP stream throughput.

- **Jitter** - Similar measurement to TCP jitter.

- **Duplicate Percentage** - The Thrulay engine puts unique sequence numbers in the packets of the UDP stream so that the receiving host can determine if packets were duplicated during transit. This metric tells what percentage of packets received were duplicates.

- **Reorder Percentage** - Using the sequence numbers in a different way, this metric tells what percentage of packets arrived out of sequence. This could happen due to a load balancer changing paths, or a routing change mid-stream.

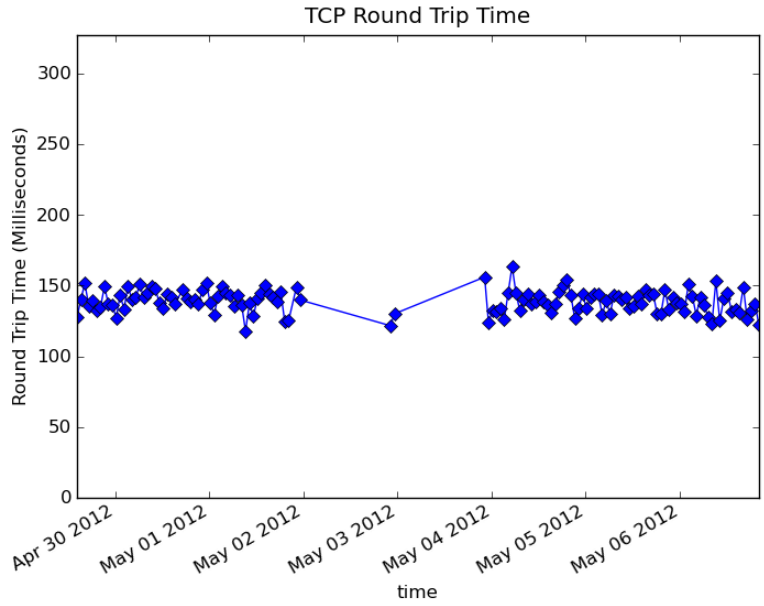
Testing Methodology:

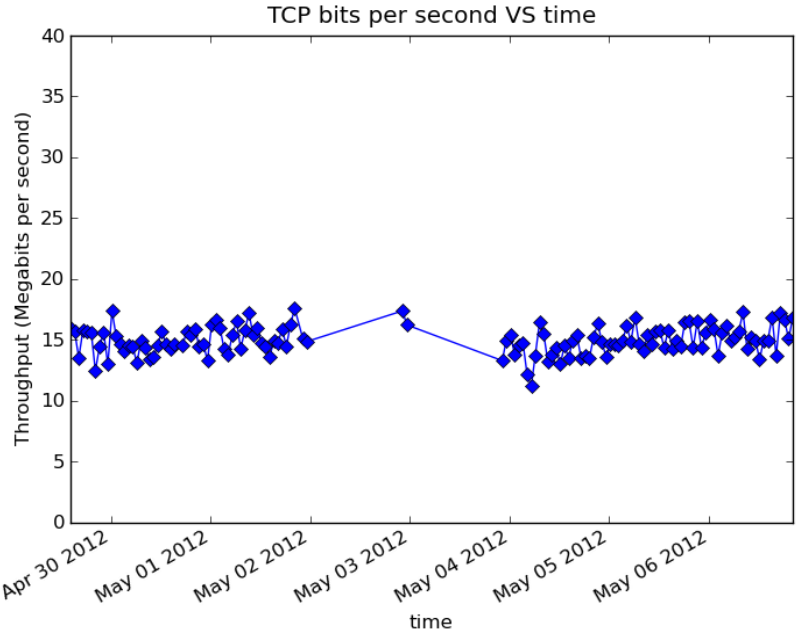
Our chosen test setup consisted of deploying two Linux hosts on modern consumer hardware with the AutoPerformance software. Each endpoint was at a house with the Century Link DSL service at the service level of 40 Mbps download and 20 Mbps upload. Both endpoints were located in the Colorado Springs area. This setup was chosen because the close geographical proximity and the same service provider network and speeds should provide for the most controlled path for the test. These factors should allow for an environment that will allow for the best chance to meet agreed upon service levels.

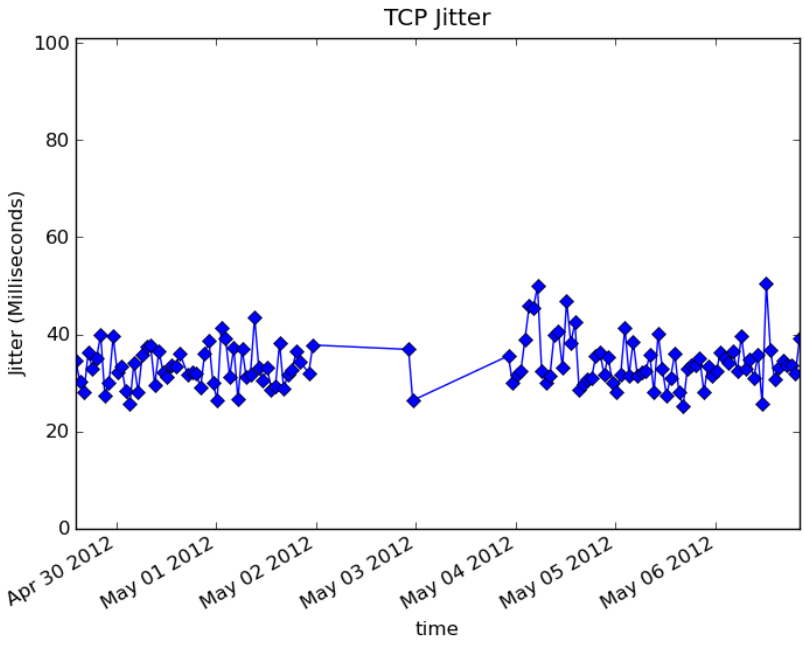
Due to the nature of dynamically assigned publicly routable IPv4 addresses, we used a Dynamic DNS service to map domain names to our dynamic provided addresses. This ensured no interruption of connections if a statically coded IP address changed.

Tests were run once per hour over a course of several days. The results were then run through the graphing utility, and are presented next.

Testing Results:





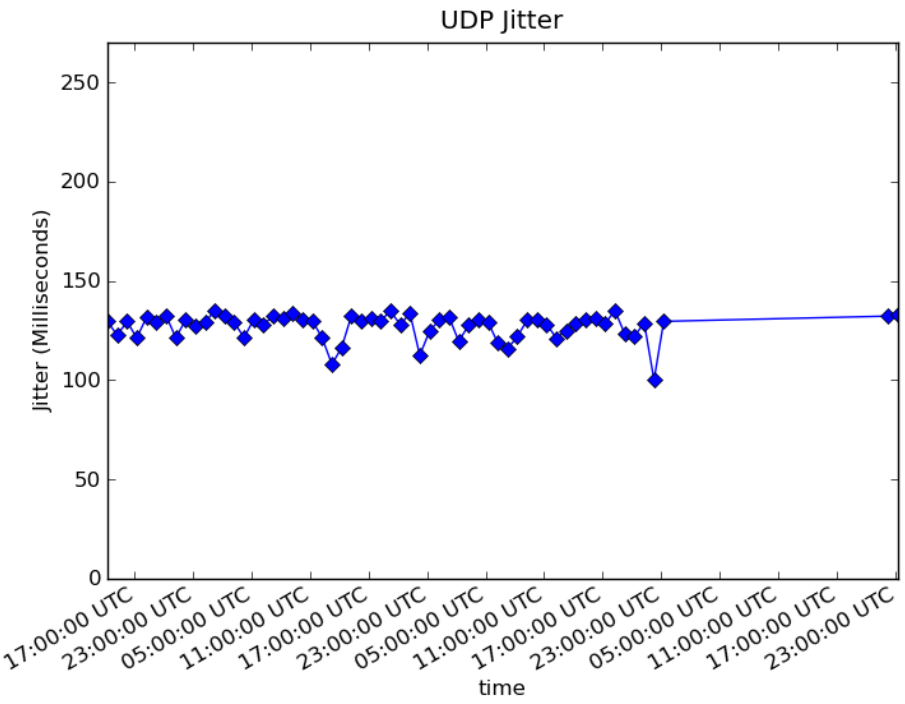


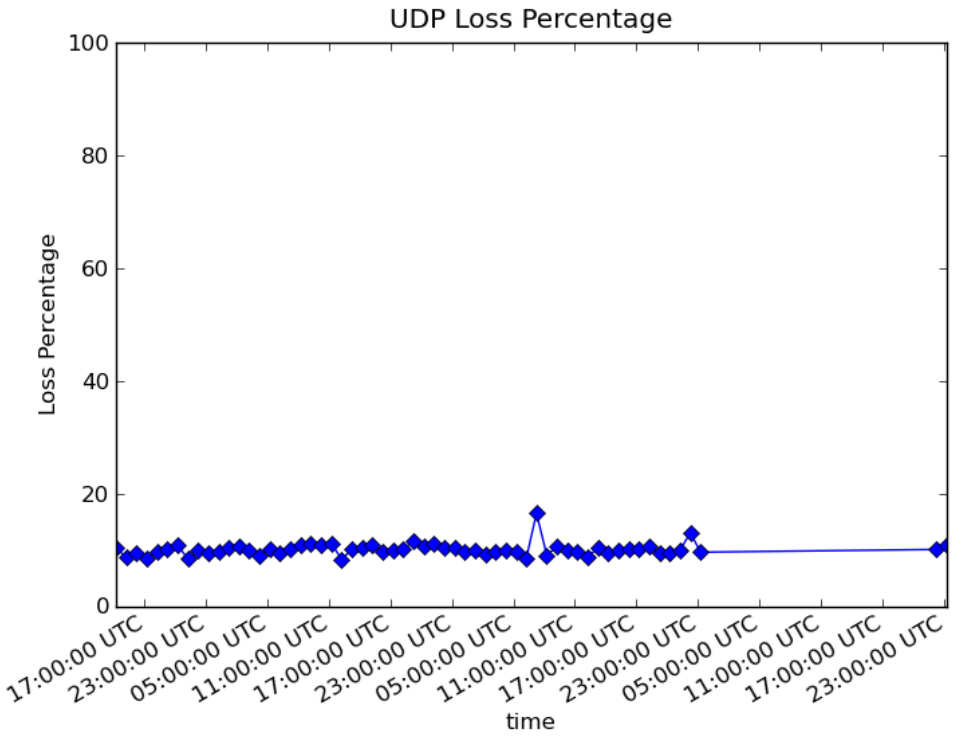
Results Evaluation:

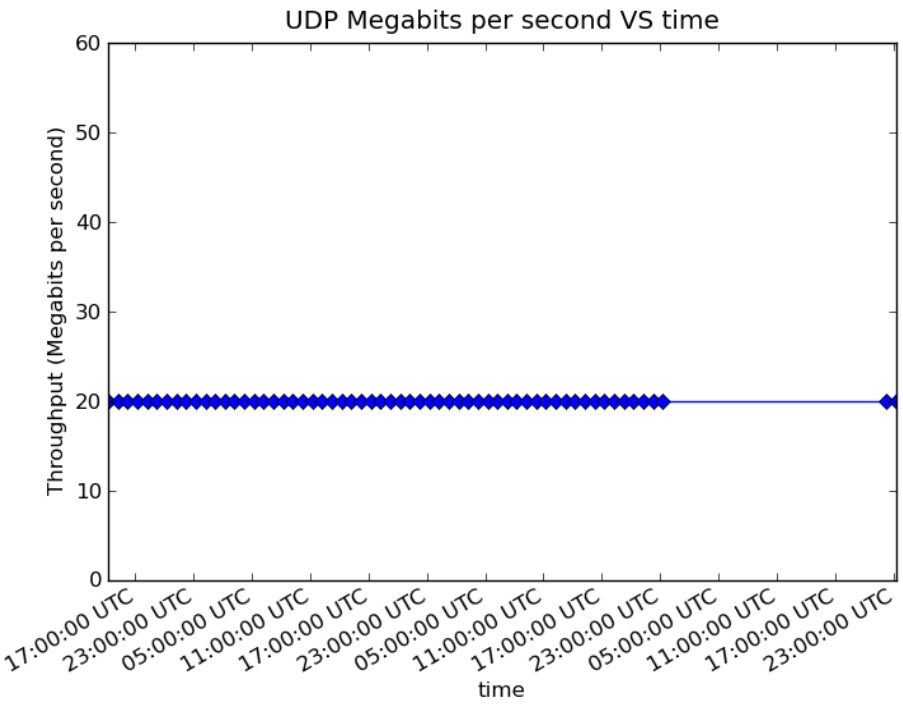
TCP:

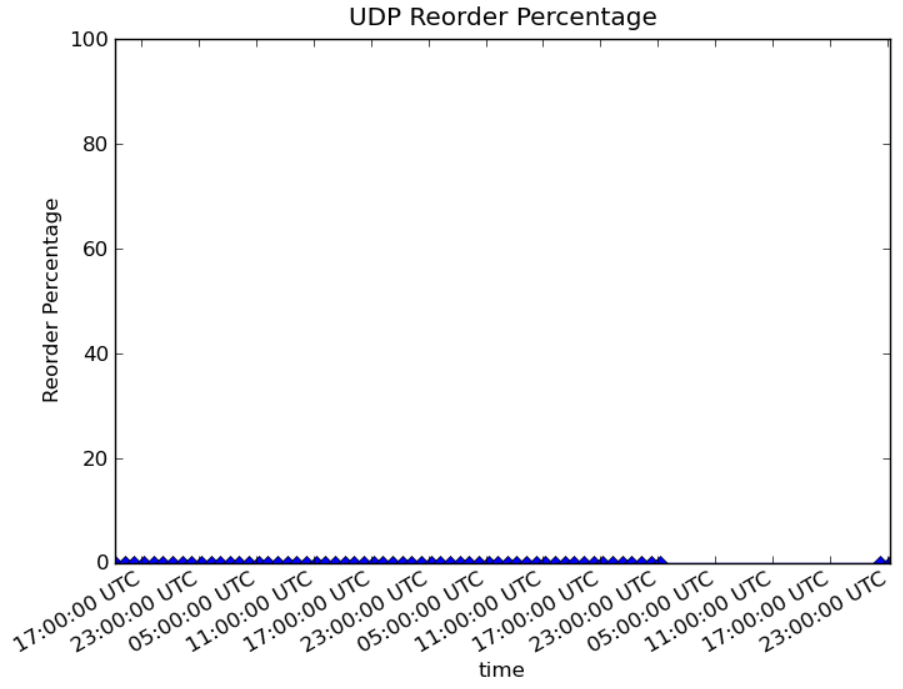
From the time range that we were able to collect, the throughput results were in line with our epectations. The theoretical maximum throughput between our endpoints would have been 20 Mbps, since that is the upload spead of each endpoint. For throughput we were averaging around 15 Mbps. This is a bit lower than the advertised rates, but is respectable enough that iw would not warrant issuing a trouble ticket to Century Link.

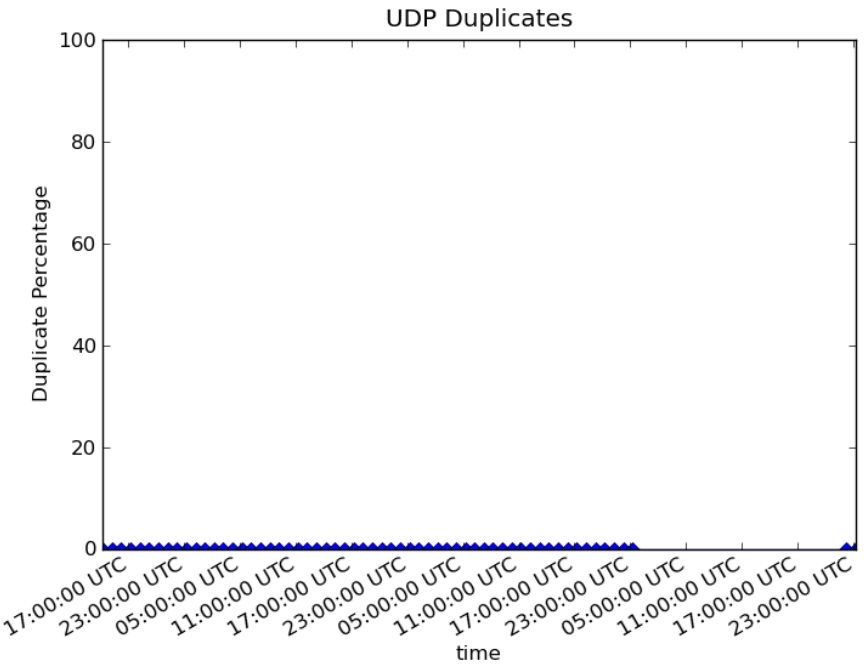
Round Trip Time was also acceptable, with an average of about 75 ms each way, for a total of around 150 ms round trip. This would provide sufficient performance to support latency sensitive applications like working over an SSH terminal connection. Jitter was an order of magnitude smaller than round trip time.


UDP Jitter
Jitter (Milliseconds)
250
200
150
100
50
0
17:00:00 UTC
23:00:00 UTC
05:00:00 UTC
11:00:00 UTC
17:00:00 UTC
23:00:00 UTC
05:00:00 UTC
11:00:00 UTC
17:00:00 UTC
23:00:00 UTC
05:00:00 UTC
11:00:00 UTC
17:00:00 UTC
23:00:00 UTC
time


UDP Loss Percentage
Loss Percentage
100
80
60
40
20
0
17:00:00 UTC
23:00:00 UTC
05:00:00 UTC
11:00:00 UTC
17:00:00 UTC
23:00:00 UTC
05:00:00 UTC
11:00:00 UTC
17:00:00 UTC
23:00:00 UTC
05:00:00 UTC
11:00:00 UTC
17:00:00 UTC
23:00:00 UTC
time


UDP Megabits per second VS time
Throughput (Megabits per second)
0
10
20
30
40
50
60
17:00:00 UTC
23:00:00 UTC
05:00:00 UTC
11:00:00 UTC
17:00:00 UTC
23:00:00 UTC
05:00:00 UTC
11:00:00 UTC
17:00:00 UTC
23:00:00 UTC
05:00:00 UTC
11:00:00 UTC
17:00:00 UTC
23:00:00 UTC
time


UDP Reorder Percentage
Reorder Percentage
0
20
40
60
80
100
17:00:00 UTC
23:00:00 UTC
05:00:00 UTC
11:00:00 UTC
17:00:00 UTC
23:00:00 UTC
05:00:00 UTC
11:00:00 UTC
17:00:00 UTC
23:00:00 UTC
05:00:00 UTC
11:00:00 UTC
17:00:00 UTC
23:00:00 UTC
time



UDP:

The throughput graph shows that we were streaming at a nominal 20 Mbps, with a loss rate of approximately 10%. This computes to roughly an 18 Mbps effective throughput. This is mostly on par with what we expected, and on the same order as TCP throughput.

We had no duplicates or reorders during UDP testing. This could be due to the fact that the routing between our to locations was stable during the test runs, and we did not have a complicated Enterprise level routing and switching LAN to go to, just our home networking equipment. It would be an interesting experiment to see if these metrics would increase if a destination host was behind a network layer load balancer.

Lessons Learned:

It is common knowledge that It is always good practice for software developers to “reinvent the wheel”. When prior work has been done, it is to one’s advantage to make use of it when possible. We heavily leveraged open source software libraries to make our job easier. These include the Thrulay library as well as Matplotlib. The more efficiently developers can make use of prior work, the faster new ideas can be prototyped and made useful.

Future Direction:

A natural next step would be to make the program more easy to use. As it currently stands, the AutoPerformance daemons must be started manually. Providing a

Debian or Red Hat software package which automatically installed and configured the system for the user could help to make the software more accessible.

As one can see from the graphs, when there was a problem with the servers, or a router lost its connection, we would sometimes get gaps in the data. Another piece of work to be done would be to improvie the graphing tools to take this into account, and allow an interactive visualization of the data.

Conclusion:

In the limited time that we were able to work on this tool, we were able to create a useful tool which can give users insight into the level of service from their ISP. This could also have applications in an Enterprise LAN environment, to monitor routing and switching between VLAN's, or geographically seperated data centers. Ultimately when one is trying to benchmark a consumer grade internet connection, it is difficult to determine the root cause of any problems (from minor ones, to major decreases in speed) because the ISP does not provide insight into how they are running their network. There may be deep packet inspection, or traffic shaping going on which can effect one's performance. As the Internet becomes an ever more central technology to communication in the home, effectively understanding performance will become more important.

Appendices
Appendix A: References

[1] - Network Performance http://www.speedtest.net/
http://www.speakeasy.net/speedtest/
[2] - Matplotlib - http://matplotlib.sourceforge.net/
[3] - Thrulay - http://thrulay-hd.sourceforge.net/
[4] - AutoPerformance Source https://github.com/codyhanson50/AutoPerformance