Title: What Impacts Network Latency
Sanket Makkar (sxm1626)

1. Introduction

As we study networks one of the measurements we repeatedly have considered for optimization is network latency. Network latency has a huge impact on the actual usefulness of a network. Consider that high latency networks make information transfer totally inefficient, defeating the purpose of the network, while low latency networks approach the vision of information traveling between two locations instantly and securely. Therefore, it is only intuitive that many in the field networks would seek to understand what influences network latency. In particular, an understanding of network latency and how it relates to other variables involved in network communication enables one to better understand what to do to optimize network performance, or at least predict it over various scenarios using simple off hand knowledge. Such knowledge enables us to identify which variable elements in a network truly impact network performance, and to what degree. In short, a deeper understanding of network latency and how it is impacted by other environmental variables - if at all - enables a better understanding of what truly impacts network performance and in turn what to do to improve network performance broadly speaking.

It is for these reasons that I want to understand network latency and its relation to various variable conditions of network information transport. Specifically, I want to find out which conditions that perhaps many assume to impact network performance actually do, and to what extent. To answer this I need to consider both what questions, as well as the who, how, where, why, and when questions of network performance. I need to consider who, as a country, has better or worse network performance. I need to consider how network latency is impacted by the path it takes. I need to consider where in the path may be the harshest network's latency performance. I need to consider to what extent network latency is impacted by the amount of data requested. I also need to consider to what extent popularity of a request impacts latency broadly speaking. Ultimately, answering these questions will shed light on what factors really do impact network performance, and to what extent.

Answering these questions will enable a better understanding of what truly impacts network latency, and will give insights as to what factors are perhaps most influential in determining a network's effectiveness in propagating information quickly. In short, network latency is something that seems almost random in some cases to me, and determining which factors truly impact latency - as well as which do not - will give incredible insight into the intuition required to understand network communication and speed, which is essential to understanding and optimizing networks in today's world.

2. Procedure

2.1 Gathering Data:

There are really three steps to make clear how I gathered data. First we need to consider which network measurements are needed broadly speaking. Next we have to consider the list of hosts we will use to analyze network latency. Finally, we need to make clear a general framework for gathering data and storing data, as this will eventually inform reasons why this data will be useful when read later on. Ultimately, after learning these things, I can dive into implementation details that give a fuller understanding of how I gathered data.

Identifying the variables required allows us to determine which network measurements are required, so we start there. The first two variables we need to consider are network latency to a host in general and network latency along the route taken. "Ping" is the obvious tool choice for a measurement here, as it by definition informs us of the latency to a host. However, we also need information on latency along a network route. To do this we can use the "traceroute" tool, as it informs us of round trip time to various points along the network path. A simple calculation taking a difference between these round-trip-times will inform us of latency at each stage of the path. So we can use a combination of "ping" and "traceroute" for the purposes of gathering data for these two variables.

The next variable to consider is the route taken to get to a particular host. For our purposes, we only care about the size of this route, but we need it nonetheless. The clear tool of choice for this is again "traceroute" as this tells us the route, and indirectly the route length as a result.

Another variable to consider is the bytes downloaded from a host upon reaching them. As stated before, we want to measure how download size from a host impacts latency - and the clear choice here for this measurement is "curl". We can actually request particular variables from a curl to a host - and bytes downloaded is one of them. This means when we gather the data, we can have a portion dedicated to just the bit we want from curl, making analysis easier later on.

The final variables to consider are country of origin for a given host, and popularity of that host. Because of where we gather data from - i.e. our list of hosts - as discussed in a later paragraph, we basically get popularity information as a ranking for free. To get information for a host's country of origin, we can actually use the host name - again provided by the list of hosts we consider - and feed that into a curl request to a database lookup from "ip-api.com". After decoding the json response, we will have a strong representative of this country.

So in brief summary of the variable question, to gather information about popularity, host country of origin, network latency along the route and in general to a host, and also the number of bytes downloaded from a host via a simple "GET" request, we only need to use a few simple tools. Really, all we need to do is run a "ping", "traceroute", and "curl -w" command over each host given to us in our host list. We can then, still for each host, use information provided by our host-list to gather country of origin data, while popularity data comes for free from that list.

The next question to answer is in regards to our host list, as mentioned previously. Basically, we need to identify where we are going to get our list of hosts. Luckily there is a “tranco” ranked list of hosts by recent popularity available as a csv for download on the project five page on the class website. So we simply download this csv, and open it in code to observe popularity data and host name. For this assignment we need on the order of thousands of data-points, so we need to balance our hosts and measurement count so that we can collect a reasonable but not overwhelming amount of data to consider. Ultimately, I ran these measurements on the top 2500 websites from the “tranco” ranked list of hosts by popularity, as it gave me a wide enough spread of popularities that I could consider the popularity vs latency data I wanted to collect, and it also collected a reasonably large number of data points. From this choice we get, as desired, on the order of thousands of measurements and five times that many actual data points to observe - one for each measurement we take.

The framework for gathering data is simple and follows fairly clearly from what we have just learned. Basically, we want to open our “tranco” host list csv. We want to then run a method that - for an arbitrarily large number of points - will safely and reasonably request a “ping”, “traceroute”, and “curl -w” on that host and store this information as a string. We then want to request country of origin data for the host, and popularity information - again all stored as strings. Finally, we want to format these strings such that, if we dump these in a text file called “data.txt” as requested in this assignment, we can easily read back the data for each corresponding host.

We start by writing ourselves a simple script, datagathering.py, which will contain all code required to gather data. Within datagathering.py we define a simple class, dataCollector, whose job is to iteratively collect all relevant data points for the number of measurements, i.e. data points, we specified.

To collect each measurement, we define a series of functions belonging to the dataCollector class named ping, traceroute, and curl respectively. Each of these functions performs basic request formatting given a host input, and then calls an underlying helper method that makes a system call to the relevant “ping”, “traceroute”, “curl” measurement as appropriate, all before doing some simple parsing through the response and returning a response string.

For the ping method, this means accepting a host name string and then forming a command ['ping', '-c', str(pings), '-W', str(PING_TIMEOUT), host] before sending this formatted command to a simple helper method - which for future reference will be named runSubproc - that uses suprocess.run(...) on this command. The ping method will then get some returned output string as passed by the helper method described earlier. The ping method then parses the first line of returned output, thereby grabbing the ip address we are pinging, and parses for the summary line of output which provides the raw “ping” summary info as a string - this includes latency. We then return this information as a tuple of the format (ip, pingLine).

The traceroute method is a little simpler, and just calls the runSubproc helper and returns the result, only specifying the command 'traceroute' and the host given to the method. The reason we don't do much formatting here is that it can be useful to have all of traceroute output available to parse later - as it includes information about route length and latencies, both of which are valuable. We could do this parsing here, but doing this during data collection is over the top for the task of just gathering data, and is probably best done in parsing the data later on.

The curl method is also a lot simpler, taking a hostname as well and basically calling and returning the output of runSubproc for a curl on the host with the -s argument, which should make the output of curl not include any progress bars that may complicate parsing the collected data. The curl method also runs curl with a -o argument to /dev/null, throwing away any actual content downloaded since we don't care about content here, and the -w '%{size_download}\n', which forces the output of curl just to be the part we care about - the size in bytes of downloaded content.

In addition to these simple data collecting methods, we have another simple helper method named "getCountry" that takes in some ip address and returns the country of origin for that ip address. We accomplish this by starting a http.client.HTTPConnection to "ip-api.com", a database online that allows us to convert IP addresses into data regarding country of origin for the requested host. The code for this method is simple, and just involves setting up the connection, sending a simple get request by providing an ip address, and then parsing the json response for the country.

Finally we get to the most important method in dataCollector - collectDataInMass. This method takes in an argument only of the number of data points you wish to collect, and will run all the measurements we have described so far. We start the method by setting up a counter of how many datapoints we have collected so far, and we then open two files. The first file is opened in write mode, and is basically the output file named 'data.txt' in our case, while the second file is opened in read mode and is the "tranco-20241115.csv" file provided by the class website. We then open a reader to the csv file, and for each line we first check if we have collected enough data - if so then we stop, otherwise continue - and then move onto the data collection stage of the "for row in fileReader" loop. Now we grab popularity straight from the csv file, increment our collected data points, for debugging purposes print our popularity so that when running the code we know which website we are on, and then make the first ping measurement. If the ping measurement fails, we see that in the output of its tuple response, and we just skip this particular website - as traceroute only works when ping works, and curl only returns a useful response for bytes downloaded when ping works. If ping did work, we now run traceroute on the host, then we run curl on the host, and finally we pass the ip address we got from running ping to our getCountry method. We store all these data points as strings, and then write the data to our data.txt file. At the end of the for loop we time.sleep for precisely ⅕ of a second, as this ensures we are limited to running each data point measurement exactly 5 times every second. Note that these measurements are not run in parallel, but instead run in sequence.

There is some complication to writing to the file data.txt that is best described in a separate paragraph. We want to write in a way such that we can easily parse through our measurements later on, and so that we can easily parse through sets of measurements as well. This can be done by having a measurement separator, in our case we use “|--|” between measurement strings, and a host data separator after we dump the measurements above in the file, in our case this is “\n------------------\n”. This way, when we go to read our data, we can read the data.txt file as a large string, then split it by the host data separator to look at each measurement set, and to look at each individual measurement string we can just split by the measurement separator.

Finally, to execute the functionality described in our class, we simply record a particular time using pythons time module, initialize our dataCollector with csv file path information, define the points we want to collect as NUM_WEBSITES, a constant that is equal to 2500 in our case, and then finally we call the collectDataInMass method. Note that after this method completes, we take a second time, print the difference in times (in ms) to observe how long our method took, and then perform and print a simple calculation on the number of hours it might take for the total number of datapoints we wish to collect to complete. We do this so that, if you want to collect fewer data points and estimate the time to get an output on the whole set of points, you can.

To run the script, simply use the command “python3 datagathering.py”.

2.2 Analyzing Data

Data analysis is probably done best in a jupyter notebook, as a notebook allows you to separate analysis into cells for a degree of separation from other bits of analysis - basically allowing you to see whatever analysis cell you really need to at any one time instead of combing through one large output. Jupyter notebooks also work well with libraries like matplotlib for plotting and visualizing information - which will be very handy in this assignment. For these reasons we do all our data analysis in a jupyter notebook called “dataparsing.ipynb”.

Data analysis really takes two steps in my notebook, data parsing and data analysis. Data parsing and analysis are done by separate classes, named dataParser and analyzeData respectively, because parsing data is really a task that is best separated from considering the data’s meaning.

The class dataParser basically contains an orchestrator method called to grab all relevant metrics, as well as a set of helpers to this orchestrator method that modularize some of the more complex string manipulation required. To make clear how the dataParser class works, we first describe the helper methods used to look through the data file and break it up appropriately, then move onto the string manipulation methods, and finally explain the orchestrating method.

The first helper method to consider is named “fileAsText”. This method takes no parameters, and just uses a “data.txt” file path that we provide at construction to open the data.txt file and then read the whole thing into one giant string and return it all.

A helper method that builds on this is “getFileChunks”. This method calls fileAsText, and then splits the resulting string along LINE_SEPARATOR- the separator between measurement sets. getFileChunks then returns an array of strings corresponding to each measurement set taken on each website. This method again requires no input.

Another helper method that we have is “chunkComponents” which only takes an input chunk string and returns that string separated into an array of substrings using the split method along MEASURE_SEPARATOR - the “|--|” we use to separate each measurement. This method basically allows us to look at a chunk string and see the individual measurement strings within.

Now we can move on to brief descriptions of the string manipulation methods used to extract various measurements from their measurement strings.

The first method is perhaps one of the simplest string manipulation methods, named “extractLatency”. Given a latency measurement string input, it greps for the characters between specific indexes, found by grepping for keywords, which are known to contain the latency in milliseconds from the output of ping. The method then casts the relevant ping timing string to a float, making sure to avoid casting any units of “ms” and just looking at the numerical components. The method then returns the resulting float, informing us of a chunks ping latency.

Another string manipulation method is extractRouteLength, which is the simplest string manipulation method we have, which just takes an input of routeLengthString - i.e. the “traceroute” output raw string - and returns the number of newlines minus an offset of one for the last newline given by the “traceroute” output. This basically tells us the traceroute length.

Our final, and more complex string manipulation method is “extractRouteTimes” which basically iterates through the same routeLengthString input, but greps each line for a timing unit at its end, and if the line has the timing unit it grabs the unit in a similar way to how we did in “extractLatency”. If the line does not have a timing unit, or if the traceroute did not get a response, we will find “*” at the end of that line - in which case we simply append the previous latency time, assuming spatial locality holds for routers in this case.

Finally we move on to a description of the orchestrator method. We name our orchestrator method getMetrics, and it takes no arguments. It starts by defining arrays named popularity, websites, countries, latencies, routeLengths, routeLatencies, and bytesGrabbed - all of which hold precisely what they describe. Just to be clear, popularity holds the popularity ratings, websites holds the website name - in case we want to query it during debugging, countries holds countries of origin for hosts, latencies holds ping output as a float, routeLengths holds the route length to each website, routeLatencies holds subarrays of floats which describe the latencies in order along the network route to the host, and bytesGrabbed holds the curl output of bytes downloaded from each host.

The getMetrics method calls a for loop to iterate through the getFileChunks output, and then within the for loop we call chunkComponents to separate the measurement set string string into measurement strings to pass to the relevant string manipulation methods. We then directly grep and append popularity to its array, and do the same for the websites array, countries array, and bytesGrabbed array. We can avoid string manipulation for these arrays because the position of the data is known and not complicated to find outright. We then finally go on to call each extractLatency, extractRouteLength, and extractRouteTimes method on the relevant chunk component before appending the result to the latencies, routeLengths, and routLatencies arrays respectively. The method ultimately returns a large tuple of arrays corresponding to each host we measure from (of the 2500), each of which indicates a data point measured.

Now comes the more complex class we use to analyze all that data we parsed. This class inits itself by taking in some metrics tuple input, and grabbing a local copy of the arrays in that tuple - so the metrics from earlier are passed in and we have a local copy of them for that instance of the class. The init method also modifies the latencies array (self.latencies). To make our latencies array reflect latency to a host more powerfully, we measure latency two ways and average them out - one comes from the round trip time to the host reflected by the last line of a traceroute output, and the other from the present self.latency[...] output. So we call a helper method defined in the class named "computeLatencyTwoWays" which looks through each index of the latency array, and averages that amount with the corresponding routeLatency to the host - which is self.routeLatency[idx][-1] in our case because of the parsing work done earlier. This forces the latency array, self.latencies, to contain a more robust and multi-source consideration of latency to a particular host.

The remainder of the class analyzeData is composed of five analysis functions, one corresponding to each finding and data observation, as well as a single helper method named statsHelper, which basically takes as input two arrays, and two array names - thus defining x-y pairs for statistical analysis while also defining the x-y axis names - and then performs statistical measurements on each using the python statistics module, printing the type of statistic, what it was performed on, and the result. This method computes standard deviation, mean, median correlation, covariance, mode along the X axis, and mode along the Y axis. Note that a nice feature of this method is that if you select one of the optional variables corresponding to a particular statistical measurement as false - then you won't see that output printed.

Note that no analysis function takes arguments, as all analysis functions just use data stored locally to the class instance. Also, each analysis function is intended to plot data and provide useful statistical information in regards to that data.

The first analysis function, “analyzeNetworkLatencyToCountries” seeks to statistically analyze and plot a series of bar graphs with error bars for data regarding the network latency to a host for various countries. We accomplish this by simultaneously recording network latency data points for each country in a separate dictionary, and also taking a simple sum of network latencies for any particular country. We iterate through the zip of self.latencies and self.countries, so that we can build three dictionaries organized by country. One dictionary contains the number of data points collected for a particular country, one dictionary contains a list of latency data points for that country, and the last dictionary contains the sum result. We then iterate through each dictionary again, and convert our sum to an average using our information for each country about the sum of latencies and the number of data points contributing to that sum. As we iterate through each dictionary for a second time, we also compute a standard deviation via the statistics module in python for each set of data points and append that to an array of standard deviations corresponding to each country. For our own purposes of knowing precise measurements, we print the average latency for each country and standard deviation explicitly, before utilizing matplotlib.pyplot’s plt.bar functionality - basically passing in a country-latency dictionary key set, value set, and separate array of standard deviations. There is some simple formatting for making the country names fit - namely rotating them by 90 degrees and enlarging the size of the graph appropriately - however much of the formatting aside from this is trivially naming the plot and setting labels.

The next analysis function is “analyzeNetworkLatencyByRouteLength”, and this function seeks to plot and analyze latency vs network route length - basically how do longer routes in traceroute impact network latency overall. To do this we make a simple scatterplot of x=self.routeLengths, y=self.latencies, and after some trivial formatting we aim to produce a line of best fit. To do this, we use numpy’s np.polyfit functionality, as it allows me to pass our route lengths and latencies and in return grab coefficients for a linear function that acts as a line of best fit for the data. After grabbing the coefficients, we simply compute an line using numpy again - multiplying (slope * np.array(self.routeLengths)) + intercept, and then pass this into a plt.plot function. This allows us to, again with some trivial formatting in the plot function, observe a line of best fit. We then call statsHelper on this data set, allowing all statistics to be printed and computed.

The third analysis function, “analyzeLatenciesAlongRoutes” allows us to observe where latency is most impacted - in terms of quartiles in the data set - along a network route. Note that this is a lot different from our earlier efforts to measure network latency, as previously we were observing total route length’s impact on latency - basically how does going through more routers impact the latency - but now we are observing how latency is impacted in terms of spread and median latency along the network. The problem here is that our traceroute is giving us RTT’s (round trip times) to us and not relative to the last router - so to fix that we can simply construct a new array that consists of the RTT difference between hops, meaning for each time in the RTT array for a traceroute output we simply make a new array that sets itself at that index equal to the current RTT - the previous RTT. By doing this, we basically figure out the time between hops, which is what we are after. So we implement this with a simple for loop that constructs a new array of these times between hops - noting that we ignore the first time since there is no difference for it to take.

Next, within a for loop iterating through all data RTT timing arrays provided by the traceroute output, we want to separate the in these arrays data into quartiles - which is simply a matter of setting the first quartile end marker to "int(math.floor(len(latencyList)/(FOUR_QUARTILES)))" and then the next two quartile end markers to a multiple (2, 3 respectively) of that marker. We can then define some quartile arrays to consist of everything from [0:quartile_1_marker], [quartile_1_Marker:quartile_2_marker], [quartile_2_marker:quartile_3_marker], and [quartile_3_marker: end]. Note that variable names in this explanation are not quite the same as what I used in code, but are named this way for clarity of explanation here.

Still within that for loop, after we have separated the data into quartiles we iteratively append to four arrays (beginning, firstSection, secondSection, end) and append the statistics.mean result of each quartile to the corresponding array.

Finally, after the for-loop we print and label in our print message the median and variance reported by each quartile, and use matplotlib to construct a graph consisting of each box and whisker plot for the data set.

Our fourth analysis function, "analyzeLatencyAlongPopularity", looks at how the latency of a host is impacted by that host's popularity. Here we start by using matplotlib to generate a scatterplot for x=self.popularities and y=self.latencies, effectively plotting popularity and its impact on latency for a large number of hosts. Because this data is all over the place, as it does not involve averages or medians or quartile cutoffs, one technique I use to make observation of this data a little easier on the human eye is to shift the y-axis into logspace using plt.yscale('log'). This enables us to more clearly see relationships in the data. For the statistical aspect of this analysis, I call statsHelper again on this data (x=popularity, y=latency) and explicitly exclude an output of the mode for the x input here - as it doesn't really make sense to look at the mode for popularity data since the popularity data is just a set of unique rankings and the mode has no meaning in that context. Additionally, mode makes much more sense to consider in the y axis, as observing which latency is most common might be more beneficial than observing that all popularities are equally present in the data set.

The fifth and last analysis function, "analyzeDownloadSizeAndLatency" observes the impact of download size requested from a host on the hosts' latency. To do this we first need to take an average for each "bytes downloaded" quantity of all the latencies observed in our data set. While this does not reduce the data set greatly, it helps make sure we get a more robust picture of the typical latency observed for a variety of bytes downloaded variable data points. We take this average as we have in previous methods, by iterating through a zip of (self.bytesGrabbed, self.latencies) and constructing as well as updating a dictionary of sums where each key is a byte quantity from bytesGrabbed, and each value is the value we wish to use to update the correct sum. Note that we simultaneously keep track of the number of times a key's sum is updated in a separate dictionary. After having made this dictionary of sums, we take the average by simply dividing each key's sum by the number of times it was incremented. Again, this is similar to how we take averages over measurements in previous analysis functions.

Finally, after having the averages taken, we construct a scatterplot of x=bytes, y=latencies based on the keys and values in the dictionary we just constructed. To model the relationship aptly, I decided to use an exponential decay function as the curve of best fit for the data. This meant converting latency values to log form using np.log, computing a simple linear fit on the log data as we did previously using np.polyfit, defining a domain, and finally setting each coefficient as e^(coefficient) in our equation: np.exp(slope * domain) * np.exp(intercept). After drafting out this curve of best fit, I plotted the curve against our scatterplot using plt.plot. Because outliers were making the data impossible to see, I also put a hard limit using plt.ylim on the amount of y-axis that can take up the plot.

For the statistical analysis here, I decided to use statsHelper on three different subsets of this data. Note that in all three cases I ruled out covariance as a measurement, as it was unneeded. Firstly I used statsHelper on the whole scatterplot data set, then I used it on the first ¾ of (x,y) pairs in the data set, and finally I looked at the last ¼ (x,y) pairs. The reason I did this was to show how the parameters, particularly correlation, changed over the data set and how the exponential decay modeled this.

Finally, to execute this analysis, in a cell below the definitions of these classes I instantiated a dataParser, grabbed some metrics from our data set using the dataParser class, and passed that metric data to our analyzeData class. In the cell below that I ran the first insight, in the cell below that the second insight function, and so on until I had plotted and displayed statistical data for all five insights.

3. Results

3.1 Network Latency Over Different Countries

Our first analysis is in regards to how network latency is impacted in different countries. In particular, I am interested in the "spread" of possible network latencies in different countries, as well as the median network latency in each country. The question we ultimately aim to answer in this analysis is, how does requesting information from another country with differing wealth and actual distance from us impact latency in the network and in what way? We in essence aim to answer the "who" question of network latency.

After parsing through two measurements each from 2500 datapoints, each one being a unique host read from the top 2500 websites in the "tranco" popular host list, I computed mean latency over various countries in milliseconds, as well as the standard deviation in latency values for each country. The table below, *Table 1*, truncated for brevity, shows a shortened list of data points indicative of observations I made in the data. A full view of the data is provided in a bar-graph provided later on after this table.

Table 1:

Country	Mean Latency (ms)	Standard Deviation In Latency (ms)
United States	10.50519040697675	14.869798309235971
Russia	57.14585714285715	28.36392674051973
Canada	5.388832995951421	8.167871609343427
Australia	19.2	13.56240903195299
France	44.765283333333336	23.105690057781533
China	109.0832076923077	16.24610167973293
Netherlands	51.98303571428571	23.68070186331938
Belgium	32.8805	33.811661631011276
South Korea	88.75835714285715	20.05216378256066

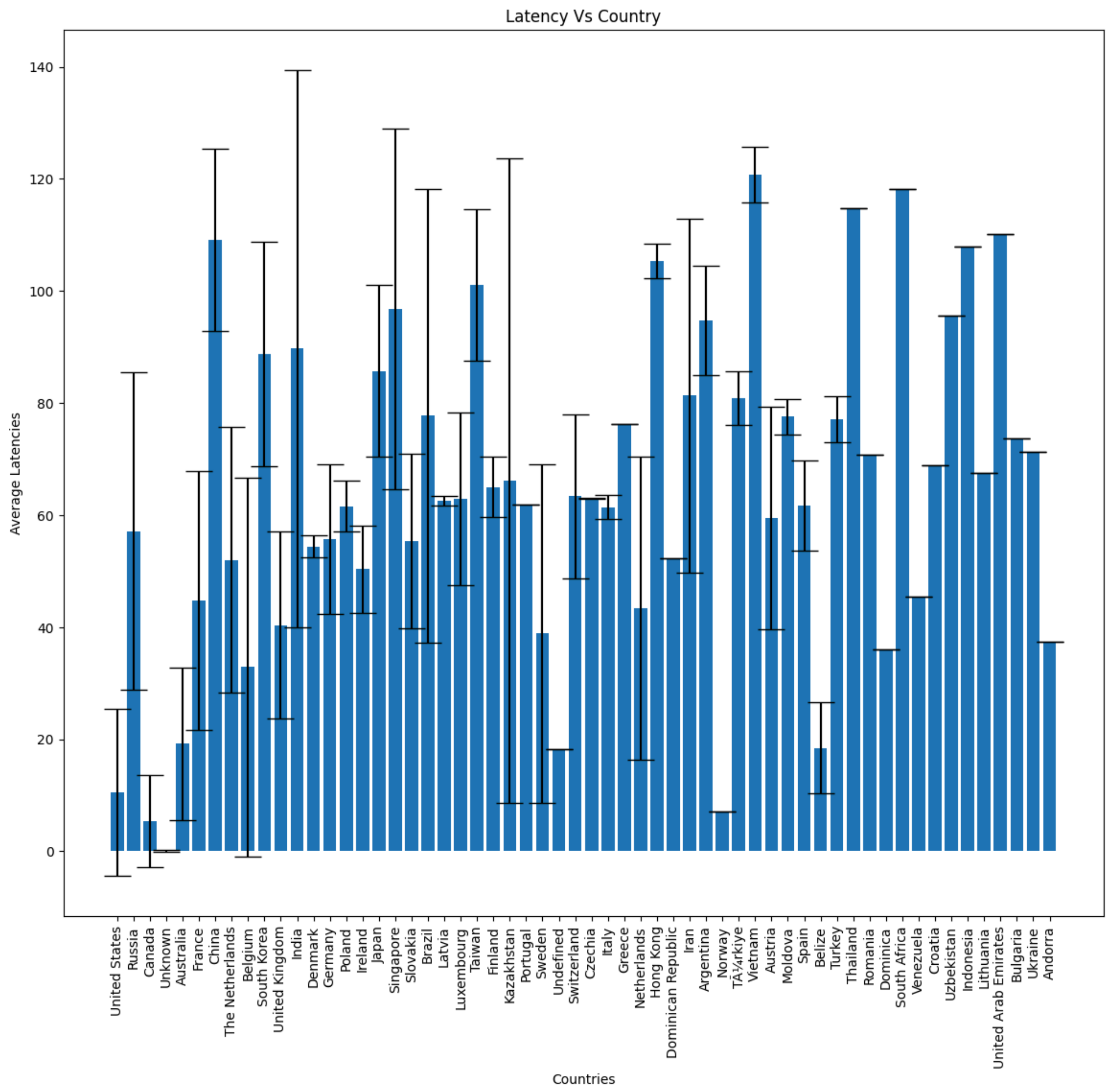
Even anecdotally from these few data points, we can make a few predictions of observations regarding mean latency and standard deviation in latency.

With regard to the mean latencies, these tend to be substantially larger for countries where the internet activity is further away from the United States, and also lower for countries that are closer to the United States. Consider the case of Russia, where most internet activity is in western Russia and therefore extremely far away from the United States. Russia's mean latency is about 58 ms compared to latency within the United States of merely 11 ms. Further consider China, a country across the world from the United States. China's mean latency is about 109 ms, nearly 9.9 times that of mean latency within the United States! Now in stark contrast consider Canada, a country so close it has lower mean latency than the United States broadly speaking, with a mean latency of only 5ms which is about half that of the United States'.

With regard to standard deviation in latencies, it seems that perhaps a clear line between countries with larger wealth - and consequently improved network infrastructure - and lesser wealth - with consequently less investment in improved network infrastructure - can be drawn. Countries like the United States, Canada, Australia, France, and China - who each have relatively larger wealth and consequently investment in network infrastructure - have extremely low standard deviation in latencies, each below 25 ms. Meanwhile countries like Russia and Belgium, with relatively lower GDP's, have substantially greater standard deviations in network latency - each nearing 30 ms.

These predicted observations, even though only anecdotal in nature up until now, hold when observing the larger data set. We do indeed find that a country's distance is a weakly accurate indicator of mean latency, while a country's wealth is a relatively stronger indicator of standard deviation in latency. To see this over the full dataset, I took all - untruncated - means and standard deviations for each country and produced a bar-graph, *Figure 1*, where each bar represents a country as described by the label beneath it on the x-axis, and each bar's height represents its corresponding mean latency over the data set. The x-axis has country labels, the right axis has latency times, and the error bars drawn indicate the standard deviation in latency. The latencies are in milliseconds. Observe the full data in the figure below, *Figure 1*.

Figure 1.



In *Figure 1* we observe that the number of countries geographically closer to the United States with lower mean latency bar lengths, which means these bar lengths are consequently closer to that of the United States', is 6 - including Canada, Belize, Romania, Dominica, Norway, and Andorra - while the number of countries geographically further from the united states with similar mean latency bar lengths is 4- including Australia, Belgium, Hong Kong, Sweden. This shows a distribution, where only a slight majority of countries with mean latencies lower, and thus closer to that of the United States' mean latency, are geographically closer to the United States than, for example, Western Russia. While extreme examples like Russia, South Africa, and Singapore exist - where countries that are very far from the United States have higher mean latencies - the distribution overall tends to lean only slightly in favor of distance driving mean latency differences.

Interestingly however, in *Figure 1*, we see that countries with greater wealth - namely the United States, Canada, Denmark, Germany, Italy, France, Poland, and many other Western European countries - have far smaller error bars than countries with relatively smaller GDP per capita - namely Russia, India, Kazakhstan, Brazil, and Belgium - have dramatically larger error bars. So we can observe that countries with greater wealth among citizens tend to have far smaller error bars - hence lower standard deviation in network performance.

Consequently we find that there is only a slight positive correlation between network latency and geographical distance between countries, but a strong negative correlation between standard deviation in network latency and the typical wealth citizens of any particular country have.

Thus our first finding is that the countries with better network latency consistency are those with greater typical citizen wealth. Through speculation one may consider such wealth indicative of a country's investment in network infrastructure, and so this kind of makes sense.

Furthermore, we find that the countries that are far away, more often than not, tend to have greater network latencies. Though this impact is more limited than the impact of a country's typical citizen wealth on network latency consistency.

3.2 Network Latency For Varying Route Lengths

The next question we seek to answer is in regards to how route length over networks impacts network latency. This is the "how" question of network latency - how does network route length impact network latency?

To measure this we observe two variables over 2500 points of data gathered from the "tranco" ranking list of most popular hosts, namely the network latency to the host and the route length. We perform statistical analysis on this output as shown in *Equations 1*, and graph a scatter plot with a line of best fit as shown in *Figure 2*.

Equations 1

Correlation Between Route Lengths and Latencies : 0.3671251722343327
Standard Deviation for Route Lengths vs Latencies : 7.928572853339499
Most Typical Route Lengths : 30
Most Typical Latencies : 3.669
Covariance of Variables : 97.11795492837688
Median Latencies : 6.349

Here we observe that correlation between route lengths and latencies over the network is a mere 0.36, which is incredibly low. Standard deviation nears a high 8, however the data we have is biased towards route lengths of 30 with typical latencies of 3.669 ms. So far these statistics do not seem to lean in favor of the assumption that longer network route lengths mean higher latencies over the network - they aren't totally unrelated, but not strongly tied together from the looks of it. Fortunately we do find a high covariance measure of around 97, indicating that increasing one variable, the length of the route, has a strong tendency to increase the other variable, the typical network latency. Median latency as well as mean latency add little value here, but are interesting to note nonetheless.

Figure 2

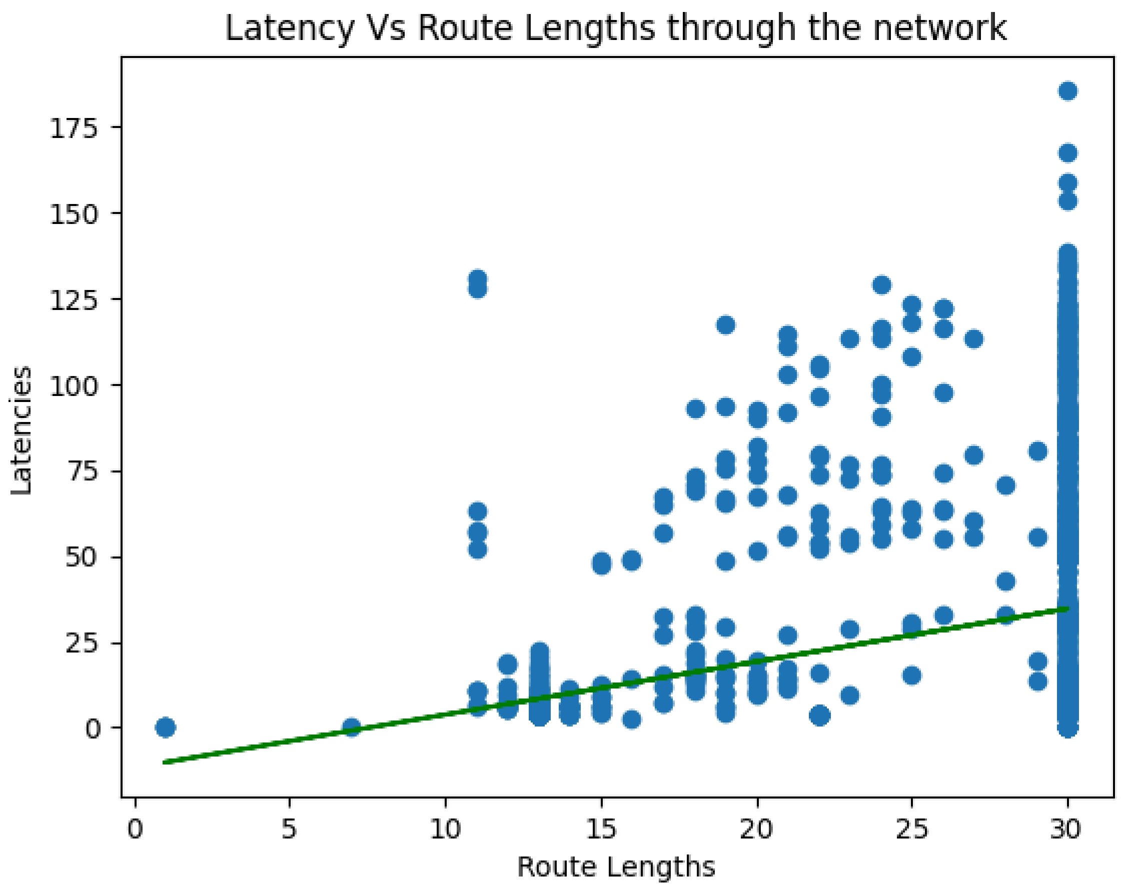


Figure 2 sheds greater light on the latency-route length relationship. We see a clear trend, demonstrated by the green line of best fit, where increasing route lengths does tend to increase latency - however another striking relationship is clear. As we increase route-lengths, the variability in network latency increases dramatically. On top of that the data is heavily biased towards route-lengths of thirty, which can be observed to have the highest variability of all.

These conclusions on visually observing the data match strongly with conclusions observed in our statistical analysis of the data. In summary, we have strong reason to believe that increasing route lengths has two clear impacts on network latency, one in the realm of latency variability and one in the realm of general covariance.

So our findings here are that there is a strong positive correlation between variability in network latency and network route lengths to a host, and a weak positive correlation but strong covariance between network latency and network route lengths to another host. So increasing the route length in a network impacts network latency by generally increasing latency for many runs, but also increasing variability in the network latency as well.

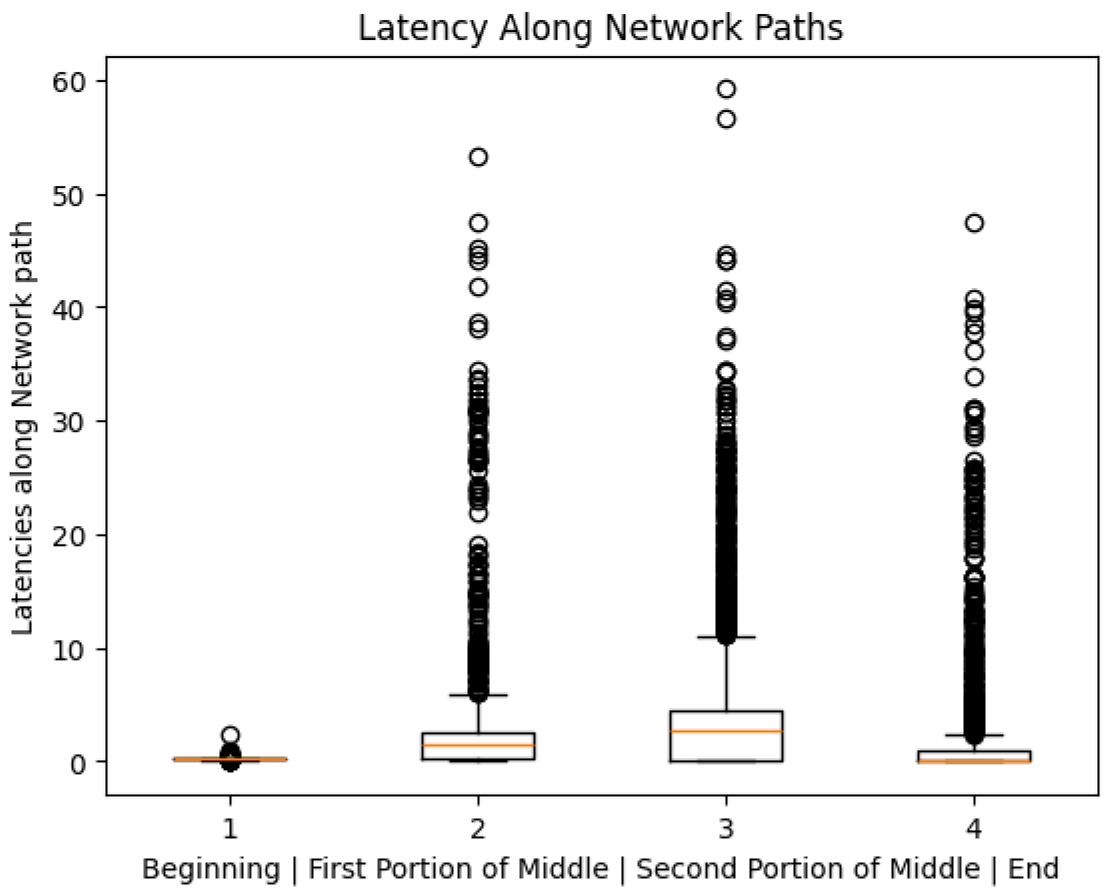
3.3 Network Latency Along the Route

The next question to consider is the "where" question of network latency - where is network latency impacted most along its path? To answer this question I took 2500 traceroute measurements and broke them down into quartile measurements before drafting a simple box-and-whiskers plot, followed by statistical measurements of each quartile's covariance and standard deviation. Observe the latency boxplot in *Figure 3* below, and the measurements along quartiles in *Equations 2* below. Note that axes along *Figure 3* represent millisecond latency in the case of the y-axis, and labels for each of the varying quartiles for the x-axis, in the order of first quartile (Beginning), second quartile (First Portion of Middle), third quartile (Second Portion of Middle), and fourth quartile (End).

Equations 2

Beginning (Quartile one) : Median = 0.21414285714285713 : Variance = 0.0769223112387344
FirstMiddle (Quartile two) : Median = 1.4417142857142857 : Variance = 5.488583854911591
SecondMiddle (Quartile three) : Median = 2.686 : Variance = 7.380808942308265
End (Quartile four) : Median = 0 : Variance = 4.919027981714869

Figure 3



Notice in *Equations 2* a general trend that continues even into boxplot data. Median latency tends to spike aggressively in the third quartile, going from 0.21 in the first quartile to 2.686 in the third quartile - a nearly 12x increase in value. The increase from quartile one to quartile two was a simple roughly 6x increase in value. Further observe that standard deviation spikes most aggressively in the third quartile, again demonstrating a nearly 10x increase in value. A bell-curve shape starts to form, however, when you observe the drastic decrease as in latency we approach quartile four. From the statistics, it is clear that network latency is impacted in a bell-curve like shape, where a significant increase occurs in the second and third quartiles while a decrease occurs following the third quartile. Additionally, we find that standard deviation follows a similar trend, approaching a steady but slowing growth up until the

Observing *Figure 3* demonstrates this bell-curve phenomenon clearly. One can easily trace a bell-curve like shape from quartile one, increasing to quartile two, increasing more to quartile three where it peaks, and decreasing back down to quartile four. Error bars further demonstrate previously observed behavior, where quartile three has by far the largest error bars. Even the error bars seem to follow a bell-curve shape.

One additional observation from *Figure 3* is that the number of outliers is drastically higher as you approach quartile three, and steadies as you approach quartile four.

In summary, we find that network latency, both in terms of variability in its value and in terms of median latency value, is greatest in the third-quartile - i.e. 75% complete region - along its path to another host. Furthermore, for the first three quartiles of measurement it seems that network latency increases in a bell-curve like manner - indicating that broadly speaking the closer you are to the third quartile the greater the network latency and variability.

This indicates that the true area where network latency performs poorest is indeed closer to the other host. This could be for a number of reasons, but perhaps the most likely is that networks nearing hosts are subject to greater traffic for very popular hosts - which indeed all of these hosts were given that they were grabbed from the top 2500 most visited hosts.

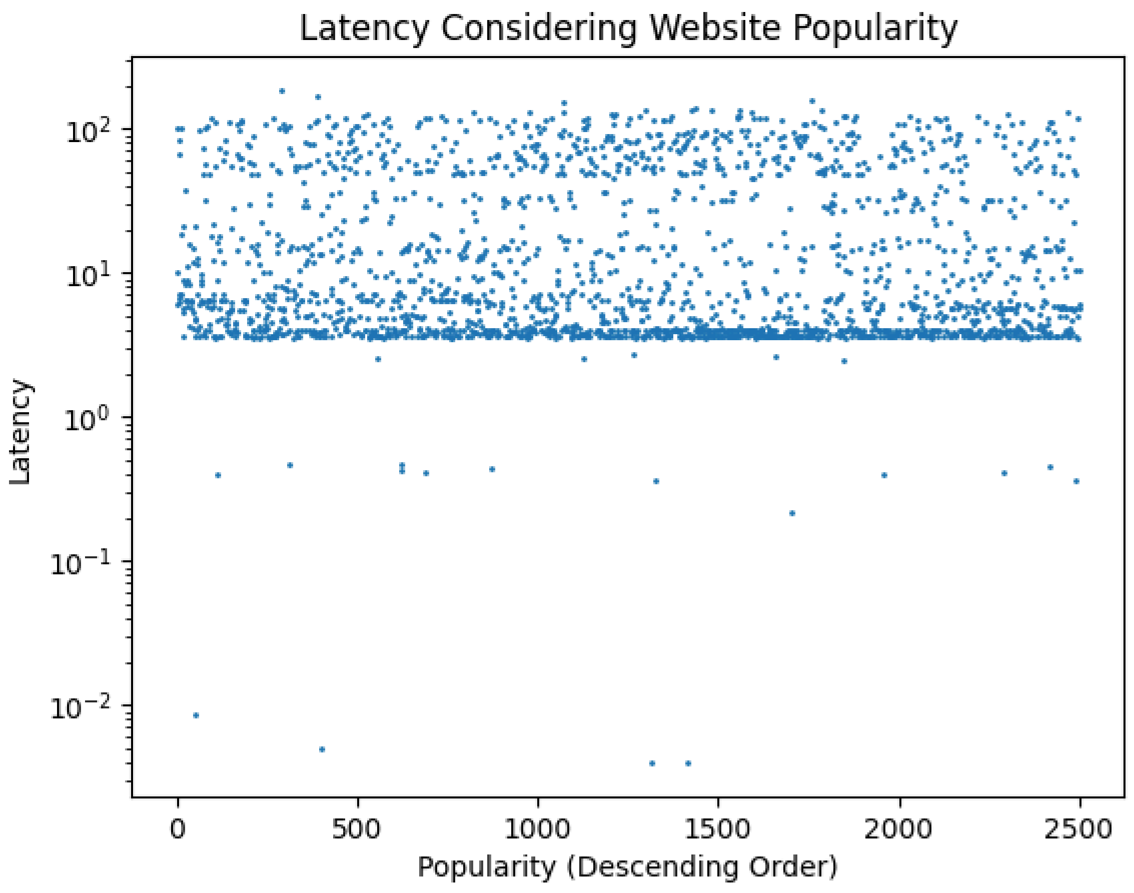
3.4 Network Latency For Popular and Unpopular Hosts

The next question I seek to answer is the first "what" question regarding network latency - what impact does the popularity of a host have on network latency if at all?

To understand this I observed the behavior of a the popularity and latency data for the top 2500 hosts in the "tranco" popular website ranking list, and performed statistical analysis on the data set in addition to producing a scatterplot to give some visual intuition as to what is happening in this data set. The resulting statistics, for a domain of popularity rankings and a range of latencies to a corresponding host, are given in the *Equations 3* below. The resulting scatterplot with latency along the y-axis in milliseconds, a domain of popularity rankings, and a logarithmic y-axis scale as compared to a linear x-axis scale is given in *Figure 4* below. Note that axis scaling changes were made for increased clarity of data viewing, otherwise the data tended to have such a great range that viewing it was difficult.

Equations 3
Correlation Between Popularity Rating and Latencies : -0.020834609672394047
Standard Deviation for Popularity Rating vs Latencies : 700.6345198819756
Most Typical Latencies : 3.669
Covariance of Variables : -487.04293030452413
Median Latencies : 6.349

Figure 4



Among the statistical results of *Equations 3*, the most striking is the near zero correlation between popularity rankings and true latency. The correlation is roughly -0.02, which is nearer to zero than any other correlation observed up until now for our data.

Additionally, the slight negativity to the correlation is indeed strong - with a covariance of roughly -487 indicating that when the outliers cancel each other out we do observe a clear negative trend - however this means very little when the correlation is so low. This just indicates that the data reliability shows a very slight, but basically unreliable negative relationship over time between a hosts popularity ranking and typical latency.

Furthermore, most typical and median latencies vary greatly - with most typical latencies being closer to 3.7 ms and the median latency being closer to 6.4 ms. While this may seem only slightly odd, it makes even more sense when you realize that the standard deviation is 700.

These statistical results all point to effectively no relationship between popularity of a website and network latency.

This belief is solidified, but also refined by the results of the scatterplot shown in *Figure 4*. Clearly there are aggressive outliers, as we saw with the statistics - but most data tends to fall within a range of 10^0.5 ms and 10^2 ms. There is also an obvious lack of correlation between the data points, as not only are they all over the place, but even for regions like those around the y = 10^0.5 mark we are hardly able to observe a noticeable change in latency due to popularity.

In effect, the plot demonstrates that, yes the popularity of a host has little to do with the latency to reach that host, and yes there is horrible standard deviation and poor correlation between the two metrics - however we do also notice that latency rarely improves relative to the y=10^0.5 ms mark, but often - though not incredibly frequently as we can see from the density of points at each mark - can spike to near 10^2 in terms of latency. This tells me that perhaps there are circumstances aside from website popularity, which lead to spikes in network latency.

Notably, the lack of impact that host popularity has on network latency may be due in part to a sample size of the top 2500 most popular hosts - and perhaps for an even greater range of hosts one might observe extreme shifts. That being said, 2500 most popular hosts is a relatively large range of hosts, and practically no correlation or relationship was visible.

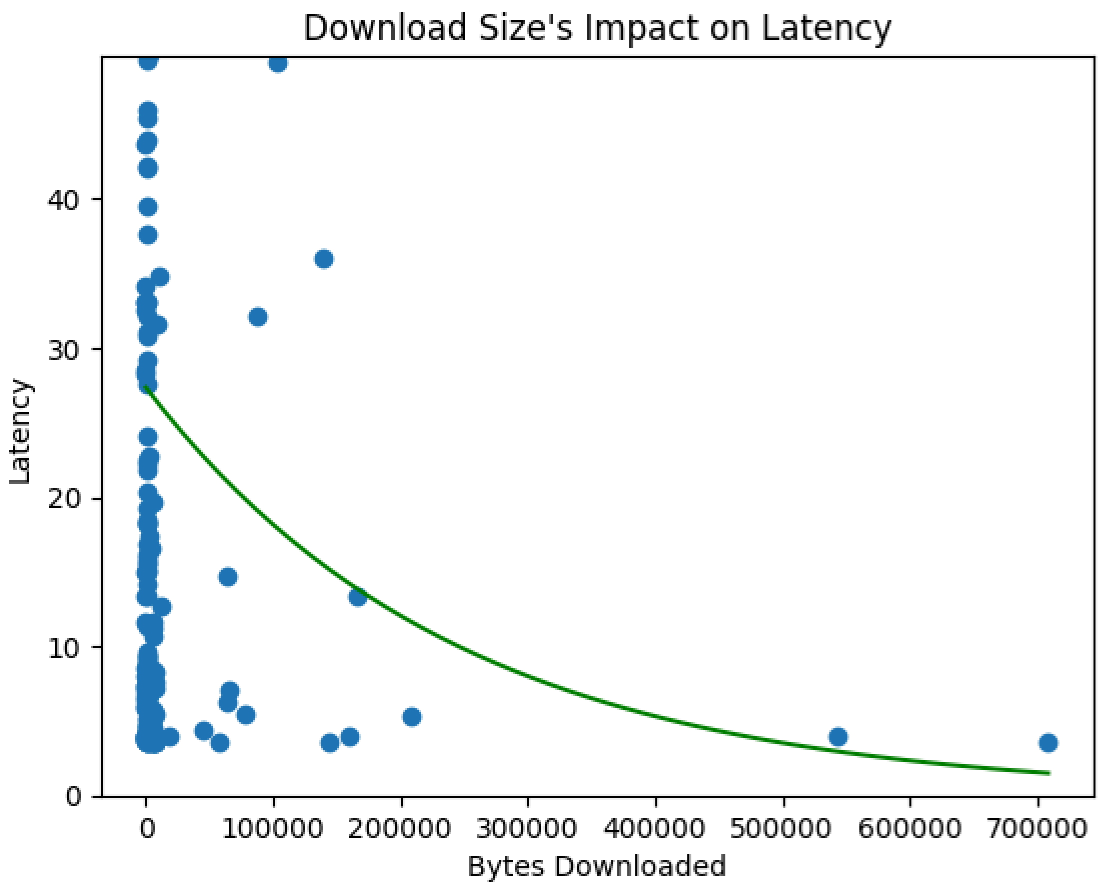
In summary, we find that there is nearly no correlation between network latency and the popularity of a website. This means the answer to "what impact does host popularity have on latency?", is that host popularity has nearly no obvious impact on website popularity.

3.5 Network Latency For Differing Download Sizes

The last question we seek to answer is the second and final "what" question of network latency - what impact does download size have on network latency.

To answer this we observe the curl result of download size in bytes across a range of the 2500 most popular hosts, and the corresponding network latency of each transaction. We first plot this data and apply a curve of best fit, as shown in *Figure 5* below. Note that *Figure 5* has a y-axis depicting latency in ms, an x-axis depicting quantity of bytes downloaded. Further note that the curve of best fit is exponentially decaying.

Figure 5



The first and most obvious thing to notice is the tremendous degree of deviation from a common latency value near zero bytes downloaded, and the increasing trend towards stability in terms of deviation from a common latency value as we near 700000 bytes downloaded. There is also a sheer quantity difference between the upper and lower bounds of the domain - which is likely due to far more data being available for lower byte-download requests than for large byte downloads given the list of popular hosts. This makes sense, as it is unlikely a popular host would provide a large file to download immediately upon entry.

Regardless of this variation, we observe that a curve of best fit decays exponentially, and seems to roughly approach capturing the behavior of the data - given the variation that is. Broadly speaking, we observe a strange trend - when we download more bytes from a host the latency tends to lower over all data points!

To further understand this trend we observe statistical calculations performed on this data set, given below in *Equations 4*. The statistical calculations are performed first for all data points, then for the first three quartiles of the domain of the data, and finally for the last quartile of the domain of the data.

Equations 4

```
ALL BYTES
Correlation Between Downloaded Bytes and Latencies : -0.02075019073526586
Standard Deviation for Downloaded Bytes vs Latencies : 21845.352486042102
Most Typical Downloaded Bytes : 0
Most Typical Latencies : 3.669
Covariance of Variables : -15124.168037404312
Median Latencies : 6.349

First 3/4 BYTES
Correlation Between First 3/4 Downloaded Bytes and First 3/4 Latencies :
-0.024537991068451247
Standard Deviation for First 3/4 Downloaded Bytes vs First 3/4 Latencies :
24475.653357753585
Most Typical First 3/4 Downloaded Bytes : 0
Most Typical First 3/4 Latencies : 3.669
Covariance of Variables : -20573.681552997965
Median First 3/4 Latencies : 6.433

Last 1/4 BYTES
Correlation Between Last 1/4 Downloaded Bytes and Last 1/4 Latencies :
-0.004882246106454414
Standard Deviation for Last 1/4 Downloaded Bytes vs Last 1/4 Latencies : 10575.007596602818
Most Typical Last 1/4 Downloaded Bytes : 0
Most Typical Last 1/4 Latencies : 3.6435
Covariance of Variables : -1566.5343409998447
Median Last 1/4 Latencies : 5.7307500000000005
```

Addressing the whole domain first, we find that correlation is again hopelessly low as was the case in the previous result. Standard deviation is absurdly high, nearing 22000 - and this is a strong indicator that perhaps this poor correlation is only due to the great variance in data nearing the zero mark in the domain. The good news is that covariance is still absurdly negative - meaning that when you just consider the relationship over the domain of the average data point there is a clear and strong negative relationship between the domain and range.

The results above are promising only with regards to covariance, however an analysis of the remaining statistics may provide some help. For the first three quartiles of domain, there is still a poor correlation that still nears -0.02, and standard deviation remains high and nears 24000, but strangely enough we see an increased covariance of variables. For the last quartile we observe even more significantly reduced correlation, but a strikingly reduced standard deviation of nearly 10,000. We also observe for the last quartile a less aggressive but still strongly negative covariance.

Though the statistical results are not nearly as promising as the graph was, we can at the very least observe that there is a generally negative trend between the number bytes requested from a website and latency - though it is obvious that this trend is so slight that correlation is negligible. The first three quartiles show this trend with the greatest strength, indicating that this trend exists where the bulk of the data is as per our plot. Due to standard deviation remaining absurdly high throughout our data, we have to admit that the trend is minute at best - however absurdly strong negative covariance indicates that the trend does exist. While the trend may suggest that hosts which provide more data have reduced network latency - this is likely influenced by an absurd degree of variance

In summary, we find that there is no trend between the bytes downloaded from a host and the network latency to that host. The answer to the last “what” question regarding network latency is that the bytes sent by a host is only one factor of many, and has virtually no impact on network latency.