Measurements of Connection Speed and Latency

Alexander Nilsson, Filip Västlund, Sebastian Salomonsson
Examiner: Karl-Johan Grinnemo
Measurements of Connection Speed and Latency

Alexander Nilsson, Filip Västlund, Sebastian Salomonsson
Examiner: Karl-Johan Grinnemo
Abstract
This report covers a project done at Karlstad University during the engineering project course DVAE08. The project was aimed to measure response time under load for "Bredbandskollen". Functionality was added to measure response time during download and upload. The response time under load was used to be able to explore if excess network queuing occurred in the connection.

To be able to measure response time under load over a variety of networks, support was added for MONROE, which is an experimental platform for mobile networks. This would enable individual pushing of clients and servers to different Docker containers located around Europe, and subsequently test from them. During upload tests, excessive latency was experienced, so-called bufferbloat.
## Contents

1 Introduction 4

2 Background 4
   2.1 Bufferbloat 4
   2.2 MONROE 5
   2.3 Docker 5
   2.4 Design of Bredbandskollen 5
       2.4.1 Server 5
       2.4.2 Client 6

3 Design of Extensions 7
   3.1 Modifications on the Client Side 7
   3.2 Server Modifications 9
   3.3 MONROE 9

4 Perform a Test 10
   4.1 Local Client Test 10
   4.2 Remote Client Test 11

5 Results 11

6 Conclusion 12

References 13

Appendix 14
1 Introduction

In the last few decades, the Internet has quickly evolved from being a thing only accessible for bigger corporations and governments into a personal service many individuals today can enjoy. It is, however, not a free service, and one has to pay for the quality one gets. Therefore, it is of utmost importance that the service given is indeed the service promised.

Measuring the capacity of their Internet connection is however not something that everyone knows how to do manually. Thus, there is a clear market for such services online, with many competing services, e.g., speedtest.net [11]. This means that every measurement service desires to give the consumers the best service to get an edge over their competitors.

There are several ways to measure the performance of an Internet connection, for example, latency and bandwidth. Some measurements are easier to collect than others, but all give useful information about a connection’s quality. The two most common metrics are latency and bandwidth. Failure to meet the demands in terms of either of these two metrics will unconditionally lead to poorly perceived service quality. With a high latency, the Internet response time will be slow, and cause a lot of waiting, and with low transfer speed the uploads and/or downloads will be unreasonably slow to complete. These measurements are basic services that most measurement services will provide. A more detailed measurement can however be used to give more in depth insights, such as finding bufferbloat in the connection. Bufferbloat occurs as consequence of oversized buffers with long standing queues.

To be able to measure and get a clearer view of the quality of mobile broadband networks across Europe, there needs to be some sort of platform in which to distribute these measuring tools; MONROE provides this platform. The MONROE-project enables end users to measure and experiment over mobile networks across several European countries [12]. MONROE utilizes the technology of Docker containers

In an attempt to demonstrate possible improvements in “Bredbandskollens” [5] service, the ability to measure response time under load and to detect bufferbloat were explored. This would provide more detailed information about the quality of the network. Furthermore, an independent GUI was requested for presentation of the newly acquired data, along with support for MONROE nodes to perform measurements from different client locations.

2 Background

This section will give the needed background information regarding the different parts of the project. This include:

- Bufferbloat, that is high latency in TCP connections, where network buffers are too large [8].
- MONROE: an experimental platform for mobile networks, [12]
- Docker: a software containerization platform, [7]
- The basic functionality of the provided source code.

2.1 Bufferbloat

As of today, most of the people around the connected world, have at one time or another experienced a drastic increase in their Internet response time. This is a problem mostly related to having a TCP connection working inadequately because of too large buffers in the network that are frequently full, hence the name bufferbloat. The way TCP is intended to work is to initially probe for more bandwidth during each transmission round, and once packet loss occur, quickly slowing down the amount of packets sent. This is because the connection can be considered at its peak once packets start to drop; the connection fills the buffer faster than it can process the data.

The issue with the excessive buffers is that the buffers will frequently remain full as the result of bad queuing. The packets transferred to a full buffer will be dropped according to tail drop, which drops packets until there is enough room from them in the buffer. Since the packets destination does not know that the packets have been dropped until the bloated packets in the buffer have been transmitted, TCP can’t properly adapt to slowdown the transfer speed. This causes a long

1 each container consists of a layered filesystem
queue of data packets to build up, which have to be processed, something that often takes quite some time. By the time TCP finally gets the signal that it should slow down, it will already be too late. In the worst-case scenario, an equilibrium has been found despite a non-empty buffer causing the connection to be permanently slower than it should.

Despite the fact that the problem with large buffers was noted as early as the late 20th century [8], it has yet to be solved. The reason for this is mostly due to the large amount of effort needed to change the different routers in the Internet, and general unawareness of the problem. Fortunately the awareness as well as the simplicity of router reconfiguration are increasing.

It is possible to see if a network connection has a problem with bufferbloat by doing a test. The first step is to see what the Round-Trip Time (RTT) is, when the network is unloaded. This can be done through a simple ping test without any latency to a network destination, and calculate the average of the RTTs.

Once the uncontested RTT is obtained, the same test will be conducted but with load applied. By monitoring how much data that is in flight during the connection, and calculating the average of the new RTT, this can give an indication if there exists bufferbloat. If the latency from the test without load differs a lot from the test under load, it may suggest that the buffer is too large. The routers can not handle the speed at which it receives packets compared to how fast it can process and redirect them, eventually this will result in long delays [1]. This is further demonstrated by calculating and comparing the Bandwidth Delay Product (BDP). BDP is calculated by multiplying the data links capacity with the RTT, this will result in a product that describes the maximum amount of data in flight that the link is able to accommodate. The actual amount of data in flight should not be larger then the BDP product. If these two values differ by several kilobytes, it means that the buffer is too large, and thus that the network path is bloated [8]. It is possible to capture how much data that is in flight by using a network analyzer program such as Wireshark [6].

2.2 MONROE

The MONROE project strives to build and openly provide a large-scale platform to measure and test mobile broadband and WiFi networks across European countries [12]. Programs are uploaded onto MONROE nodes, which are then executed inside a Docker container [7]. MONROE nodes have no restriction on resources (provided the node itself has enough resources), however, only permits one test at a time to be run. This requires scheduling of the tests, which is handled by their own web-based API. The measuring programs are built inside Docker containers, with any dependencies (e.g., Python, compilers etc.) installed from scratch with the help of a prepared build script [13]. Once, a test are to be run, the Docker container is uploaded and scheduled to execute via the web-based API.

2.3 Docker

Docker containers enable individual and isolated software to execute, complete with it’s own filesystem, inside of another environment [7]. Compared to a virtual machine (VM), each piece of software does not run inside their own operating system (OS), something which makes containers much more light weight than virtual machines. Due to this, multiple docker containers can be run inside a single OS at the same time, wherein using VM’s to produce several testing software would require a complete OS accompanied with each software.

2.4 Design of Bredbandskollen

The source code distributed from ”Bredbandskollen” contained a server and a client application. In the following, we will further explain how these two program worked before being modified.

2.4.1 Server

The server code is made in C++ and comprises a main process that manages several child processes. During it’s execution, it listens for a client connection request. The server creates a so-called ticket that contains information, which enables the server to distinguish the clients from each other. Each client needs to fetch the ticket before an connection can be issued. The client has to use the ticket for each of its threads. Because the client uses the same ticket for each thread the server knows all the threads that are used during the measurement. During the measurements, the server will
log the statistics for each measurement and store it in a text file. Furthermore, during the whole test, the server will store the client’s different requests and measured data in log files. The files are by default created in the /var/tmp directory, however, the default location of the files can be changed by modifying the source code.

Figure 2.1: Schematic class diagram over the server.

### 2.4.2 Client

The client code is made in Python 3.5 [4], and is relatively small compared to the server code. It consist of three parts:

- A measuring class, whose main purpose is to run tests and to log the resulting information.

  (stored inside measurement.py)

- A progress class which keep track on how far a test has gone.

  (stored inside measurement.py)

- A basic configuration setup which makes sure that the measuring class executes as intended.

  (stored inside bbk_cli.py)

The measurement class is initiated through the use of predefined configurations stored inside a dictionary. These values represent information about the connection, such as IP version (IPv4 or IPv6) and which port number to use. With the use of these values, it tries to discover the available test servers through a remote PHP file at their web page, and additional information by loading physical devices. All of this is done in preparation of the tests themselves.

To make the tests work properly, the client attempts to connect to the given server and wait for it to properly establish it’s link. Once the link is setup, the measurement itself will start. As a first step, the client carries out a pair of simple ping tests to give a rough estimate of how much delay there is in a system without load. Both of these tests entail executing a pre-set amount of 15 pings and then take the lowest five as the average ping without load.

Once the ping measurements have been carried out, the client will start with download tests. This is done by starting an already specified number of threads which carries out the download. If
the speed seems adequate, the number of threads that download is increased until it’s considered to have reached its limit. Upload works in a similar manner: it initiates a fixed amount of threads that will upload a pre-specified amount of data to the server. This data is defined as a byte stream of random characters. The client will wait for an answer from the server in order to start uploading. Once the server sending time exceed the maximum desired sending time, the server will conclude the uploading and return the final result. When the client has concluded all types of measurement requests, it will request the server’s log of the tests as well as a JSON-object [2]. The server will create the logs based on its internal TCP settings. The client program will write the received data in the terminal which started the client.

3 Design of Extensions

This section will more closely explain how the client and server was modified to encompass measurement of latency under load.

3.1 Modifications on the Client Side

The following extensions to the client side were implemented by us:

- A basic GUI for an easier summary of the results. The GUI is able to display statistical results from the tests and draw graphs live in regard to upload and download ping.
- Ability of the client to measure pings during an arbitrary time interval, not just 15 pings.
- The extended client configuration can be done both via the command-line and configuration files, in addition to the already existing dictionary.
- The duration of tests are no more limited to 10 s, instead they can run for a duration specified by the client.
- Added ping test for both upload and download.
- Complemented the ping measurements with statistics: median, standard deviation, 25th-percentile and 75th-percentile in addition to the average.
Everything regarding the measurements themselves are done in `Measurement.py`, while configurations and GUI are done in `bbk_cli.py`. The function that pings the server is called `ping_until_cancel` and works exactly like the previous ping measurements, except that it runs until it gets a cancellation instead of a pre-set number of pings. In that way, it enables ping series of arbitrary duration. It can be executed for any duration, until a prerequisite condition ends and cancels it. When carrying out the unloaded ping test, the procedure `ping_until_cancel`, which is asynchronously defined (allows it to run simultaneously with download and upload), puts the execution to sleep for a pre-specified amount of time. This allows it to be executed on its own, and is canceled when the sleep expires. In cases where the connection is under load, the ping is canceled when the data transfer completes.

Once the ping function is canceled, it will calculate several different metrics and return the results in a list. An example of this is shown in Figure 3.1. The calculations are done with the use of already existing libraries (NumPy) [3].

The duration of the download and upload tests is simply extended by replacing the previously hard-coded timeout value with the duration value defined in the configurations. As the test now also start pinging during transfer as well, it will, in addition of returning the speed, also send the result returned from the ping measurement.

The GUI is made completely from scratch using the PyQt5 GUI library [10]. There are two major reasons for us to building the GUI from scratch. It was deemed easier than trying to modify the GUI provided by "Bredbandskollen". Most of the existing GUI was downloaded through a Java object on the "Bredbandskollen" server. If changes were made to the original GUI, it could affect the one used in the project. Furthermore, the GUI did not contain any functionality to calculate the different metrics and draw graphs, something which was necessary to visualize the results of the tests. As more functionality was needed to be added, a new GUI would provide more design freedom. A simple measurement example of the GUI can be seen in Figure 3.1. The start button starts a new test, and once a test is done, all its data is stored in lists. They are saved so they can be displayed again with the Next/Previous buttons in case several tests are done in the same session.

![GUI](image)

**Figure 3.1:** A simple example of a measurement using the GUI
3.2 Server Modifications

When it comes to the server, the only change made by the project group was to add functionality to store a string sent from the client at the server. The extended log features included the addition of a class variable called `std::string client_data` inside the `ticketconnection.h` file, and a new valid address path called `/logThis`. The new server address path is processed inside of `Connection::HandlerState TicketConnection::new_post_request(size_t header_len)`. (This function previously only handled the random data uploaded during the measurement of the clients upload speed at the path `/cgi/upload.cgi`.) The log statement simply copies the data inside of the packets from the client and store them inside of the `client_data` variable. It only stores the last sent data string.

```cpp
if(get_url() == "/logThis") {
    // This should work 100% unless client sends weird logs
    log() << "Enter client log";
    char tmp[buffer_pos];
    std::copy(buffer+header_len,buffer+header_len+buffer_pos,tmp);
    client_data = tmp;
    log() << client_data;
    log() << "Exit client log";
    send_http_msg("Ok\n\n");
    return HandlerState::REQUEST_DONE;
}
```

Figure 3.2: logThis source code.

Once the client tells the server that all the measurements are done, the server will take whatever string stored in the `client_data` variable and put it inside of the the already existing JSON object on the server at the end of a test. The string from the client is already formatted, so the server will not need to do any changes to it to fit inside its JSON.

3.3 MONROE

Support for MONROE prompted a complete overhaul of the server and client structure. Previously, all server and client files were stored in the same directory. However, due to the fact that it is unnecessary to push the server along with the client and vice versa to the MONROE node, they are now separated into their own subsections, each with their own Docker script.

In addition to restructuring the project, some dependency problems occurred during the installation of Matplotlib and PyQt5 extensions of Python. Due to these problems, it was decided to split up the original `bbk_cli.py` into two separate files, one supporting the GUI, and one which were to be run inside the MONROE node, `monroe_cli.py`.

```bash
FROM monroe-base
MAINTAINER goglittegami.com
RUN apt-get update && apt-get install -y --force-yes -no-install-recommends -no-install-suggests \\
    python3 python3-dev python3-pip python3-dev-docs \\
    && apt-get install build-essential -y \\
    && python3-config \\
    && make \\
    && make install \\
    && apt-get update -y \\
    && rm -rf /var/lib/apt/lists/*
RUN pip3 install -d pip
RUN pip3 install matplotlib
RUN pip3 install numpy
RUN pip3 install matplotlib
COPY files/* /opt/monroe/
ENTRYPOINT ["python3", "bbk_cli.py"]
```

Figure 3.3: Docker script for the client.
4 Perform a Test

The client application supports two ways of doing measurements. The test is performed through a local copy of the client or a client stored on a MONROE node.

4.1 Local Client Test

To perform a test, all the dependencies from Appendix A have to be successfully installed. When those steps have been completed, the project has to be downloaded from Git. Once the project has been downloaded, the user has to upload the server code that is located in the server folder to their server location. The server code has to be compiled by issuing the command `make` at the command prompt. Once the server code has been compiled, it is started by writing `./bbkmaster` at the command prompt. To check that the server is running, the user should write `ps aux` at the command prompt.

In order to start a test, enter the TestScripts folder. This folder contains three scripts where `clean_logs.sh` removes every temporary object that was created during the test, `clean_docker.sh` removes all redundant objects created after a MONROE test, and `run_tests.sh` is used to start a basic test. In the three scripts, the specified file path inside the scripts has to be changed to the file path where the user have stored the files. The script `run_tests.sh` will start a test where the client will issue ping requests to the server during a pre-specified duration of time. The user is able to decide for how long the test should be executed. Usually it is enough to set the time variable to no more than 120 seconds. When `run_tests.sh` is started, the user can choose to remove the objects from previous tests or to run a new test. When starting a new test, the settings for the test will be read from the `conf.txt` file located in the `client-files` folder. These settings can be changed by opening this folder and edit `conf.txt` manually, or by setting them when the `run_tests.sh` script has been started in the command prompt during the Any configuration input? question.

The available flags a user can choose from is:

- Perform the test with a GUI by issuing the `-G` flag.
- Set time in second with the `-t` flag.
- Set port with the `-p` flag.
- Set server with the `-s` flag.
- Open a specific file to run configurations from with `-F <Filename>`.
- Specify if IP is IPv6 or not by writing `-v6 <True/False>`.

```
1 -G
2 -t 120.0
3 -s 127.0.0.1
4 -p 8080
5 # Any configuration input?
```

Figure 3.4: Docker script for the server.

Figure 4.1: Example of `conf.txt` file.
A test starts with the client pinging the server without any load and then pings both during download and upload. Each instance of the test will be performed for the set duration. After one part of the test is completed, the calculations of the statistic measures, e.g. median, average, standard deviation, 75th and 25th-percentile, will be visible on the GUI. During download and upload, graphs illustrate the latency for the performance test. Once the whole test has been completed, it is possible to view the results in a log file that has been stored on the server, it can be accessed by visiting the var/tmp folder. The results have been logged on the bbnmeasure.log files. The graphs during the tests are stored as PNG images, and is located in the client-files folder.

4.2 Remote Client Test

At the time of writing, neither the server nor the client has been tested on an actual MONROE node connected to their network. However, they have been deployed on a locally created MONROE node, and successfully executed. To run our test from a MONROE node, follow the steps taken in Section 2 through 5 in the MONROE user manual [13].

5 Results

The key results of this project are the added functionality of measuring latency under both upload and download, as well as an extended unloaded latency measurement. All of these three measurements runtime depend on a time variable, which can be set in either the conf.txt file, or by user input via the flag -t. The results of each measurement is sent back to the server and logged in the file bbnmeasure.log, which is by default located in /var/tmp/. Both server and client has support for MONROE containers, although not yet tested on a live MONROE node.

The implemented GUI shows a live updating graph of the latency during download and upload, as well as all the gathered and calculated latency measurements. It is able to display the download and upload speed as well as the latency corresponding to each phase. Statistical measurements results on median, standard deviation, 75th and 25th-percentile are also visible in the GUI.

During certain test, we could observe an indication of bufferbloat. Notably, we observed that the latency was continuously getting higher until the end of the execution. As seen in Figure 5.1, the upload latency is really high with an average 628 ms as compared to the ping test without load which gave an average unloaded response time of 19 ms. The download speed is a bit high compared to the flat latency, but nothing when compared to the upload latency, which is higher then it should be. This could be an indication of bufferbloat in the connection: the buffers are so large that they will be slow to handle all the requests which will result in a tail drop when new packets arrive to the buffer. Without the destination knowing that the packets have been dropped, TCP is unable to adjust and slow down the transfer speed.
6 Conclusion

In this project, the purpose was to improve the information obtained by "Bredbandskollen" during their network connection test. Originally, one could only obtain the unloaded latency during the download speed and upload speed, but it was desired to obtain the latency under load. We were able to add functionality in order to perform latency during download and upload test. During the download and upload measurement test’s, particularly high latency was detected for upload. Moments after measurements of response time under load was done, the latency returned back to normal values. Thus, there is plausibility that bufferbloat exists somewhere in the connection. We were also able to create a GUI that is able to create graphs to visualize the latency during download and upload tests. The GUI is also able to visualize statistical results from the tests.

While support for MONROE is added in both the server and the client, we have not yet managed to install all dependencies which are required for the client GUI, namely, Matplotlib [9] and PyQt5 [10]. Due to these problems, we have split the client into two parts, one which is executed inside the MONROE container but lacks the GUI, and one which includes the GUI but can not be run within MONROE.

For future improvements, we suggest trying to successfully install these dependencies. As a result, the client which lacks the GUI would be rendered redundant. Additional future work includes finding ways to improve the stability during tests, as they sometimes can disconnect during long tests, about 120 seconds long. The test scripts can be improved to add more options for the users. Functionality to calculate the BDP and monitor the amount of data that is in flight are also improvements that could be made. This makes it possible to monitor if the connection experience a reasonable amount of queuing.

![Figure 5.1: A 120 seconds ping test.](image)
References


Appendix A

This appendix include a step by step guide how to set up the working environment.

1. VMware
2. Debian 8+
3. Install sudo
   - Open the Terminal
     - Click Activities
     - Click in the Type to search... box
     - Type in Terminal and press the [enter] key
   - Switch to root user
     - Type in the Terminal the following command su
     - Press [enter]
     - Type in the root password and press [enter]
     - The command prompt should now look like this:
       root@debian:/home/<Username>
   - Install
     - Type in the following command:
       apt-get install sudo
     - Press [enter]
   - Add <username> to the sudo group
     - Type in the following command:
       adduser <username> sudo
     - Press [enter]
   - Add <username> to /etc/sudoers file
     - Type in the following command:
       nano /etc/sudoers
     - Press [enter]
     - Scroll down and look for the line:
       %sudo ALL=(ALL:ALL) ALL
     - Below that line type in the following:
       username ALL=(ALL:ALL) ALL
     - Press Ctrl+x then press y and then press [enter] to exit and save the file.
   - Exit out of the Terminal completely
     - Type in the following command
       exit
     - press [enter]
     - Type exit
     - Press [enter]
     - The Terminal application should now be closed
   - Open a new Terminal to test if sudo works with the added user
     - Click Activities
     - Click in the Type to search... box
     - Type in Terminal and press the [enter] key
     - Test sudo by typing the following command:
       sudo ls
     - Press [enter]
     - Type in your password and press [enter]
     - If the output looks like the following:
       <username> is not in the sudoers file. This incident will be reported.
       Start from the beginning of the instructions and do it again.
4. Install VMware-tools
   1 player-> manage-> install
   2 go to downloads and extract it
   3 Go to the extracted file through terminal
   4 sudo ./vm—.pl
   5 Restart

5. sudo apt-get update
6. sudo apt-get dist-upgrade
7. sudo apt-get install build-essential
8. sudo apt-get install make
9. sudo apt-get install linux-headers-$(uname -r)
10. sudo apt-get install git
11. sudo apt-get -y install build-essential debhelper devscripts
12. sudo apt-get install python3-pip
13. sudo pip3 install aiohttp
14. downloading server:
    cd Desktop
15. (Assumed 07/09/2016) Install python
    1 sudo apt-get install python3-pip
    2 python3 -m pip install aiohttp
    3 pip install –upgrade pip
    4 sudo apt-get install libssl-dev openssl
    5 wget https://www.python.org/ftp/python/3.5.0/Python-3.5.0.tgz
    6 tar xzvf Python-3.5.0.tgz
    7 cd Python-3.5.0
    8 ./configure
    9 make
10. sudo make install
11. If this is done correctly "python3.5" should now work
12. Test if aiohttp work
    1 su root
    2 python3.5
    3 import aiohttp
    4 print(aiohttp.__version__)
    5 If done correctly, it should show a version
13. If There is no aiohttp, try the following:
    1 sudo apt-get install python3-pip
    2 You may need sudo apt-get install python3-pip3 instead.
    3 su root
    4 python3 -m pip install aiohttp
    5 Pip may need to be updated (you may have to change pip to pip3)
    6 pip install –upgrade pip
    7 sudo apt-get install python3-pip
16. Installing netbeans

1. Download JDK at:
   

2. Go to download location
   
   `sudo chmod +x jdk-8u101-nb-8_1-linux-x64.sh`

3. Go to the same location as before, but the update tab instead of plugins-tab

4. Inside Netbeans, click on My Netbeans on the start page

5. Install all python and c++ plugins, then restart

6. Inside Netbeans, click on My Netbeans on the start page

7. Go to the same location as before, but the update tab instead of plugins-tab

17. Configure git in Netbeans

1. Go to team>git>clone in Netbeans taskbar

2. Enter:

   https://git.cse.kau.se/DVÆ08/bredbandskollen.git

   in URL field

3. Configure it correctly with login and branch, then click OK/finish

4. Make a comment somewhere and then commit it through Team>Commit from the taskbar

5. Compare to repository head to see if there is a difference Team>Diff>diff to repository HEAD from the taskbar

6. Push changes with Team>Remote>Push from the taskbar

7. Compare to repository head to see if there is a difference Team>Diff>diff to repository HEAD from the taskbar

8. If everything is working, they should now be the same.

18. Install pyqt5 libraries

1. `sudo apt-get install qt5-default`

2. `sudo apt-get install libqt5webkit5-dev`

3. Download SIP (latest verified working version: 4.18.1)

   1. Get the linux version at:

      https://www.riverbankcomputing.com/software/sip/download

   2. Extract anywhere you want

   3. `tar xzvf sip-4.18.1.tar.gz`

   4. Go to the extracted file path and do the following commands

      5. `sudo python3.5 configure.py`

      6. `sudo make`

      7. `sudo make install`

4. `sudo pip3 install pyqt5` (if this did not work/give errors when running, uninstall and try specifically PyQt5 version 5.5)

5. Download PyQt5 version 5.5

   1. Get the linux version at:

      https://sourceforge.net/projects/pyqt/files/PyQt5/

   2. Extract anywhere you want

   3. `tar xzvf PyQt5-gpl-5.5.tar.gz`

   4. Go to the extracted file path and do the following commands

      5. `sudo python3.5 configure.py`

      6. `sudo make`

      7. `sudo make install`
6 sudo apt-get install libcanberra-gtk-module

19. Install matplotlib library
   1 sudo pip3 install matplotlib
   2 sudo apt-get install python3-tk
   3 Test if it's done correctly
      1 python3.5
      2 import matplotlib as mpl
      3 mpl.use('Agg')
      4 import matplotlib.pyplot as plt
      5 repeat once in case render/load error and stuff.