Bench-Routes

https://github.com/zairza-cetb/bench-routes

19th August 2019

Language - Go (Why use Go for the project? Link)

Note:

1. All collaborators are requested to go through the newer content added and also give a complete proofread to the proposal and comment in all instances which seem unrelated. I think we are near to complete the documentation for the first release of the product. Hence, would be exporting to the Zairza Whatsapp group.
2. **important** Please go through the newly added Network Congestion approach and give suggestions on its improvement.
3. Please go through the brief description of the High-Level Model.
4. **Important** We would be starting the development process in some moments. Please mention how are we going to divide the work. Like, are we going to make issues and assign them, or in any way. For answers, please comment on this statement.
5. **Important** **New** Development on bench-routes has been started. All efforts would be maintained regarding the use of the latest concepts, in order to provide maximum performance and extract out the max from golang. While implementing any module, please note which one needs to be multithreaded and hence provide the channel support to them for communications with the main thread. Eg. ping operation (for reference) takes significant time, hence if not multithreaded, then it would freeze the application until the process exits. Hence, to make things working synchronously, please make use of goroutines and channels.

This section corresponds to the document and research efforts required to search and implement the logic of some algorithms which are faced with difficulties in finding a feasible solution yet. You are requested to contribute your ideas for implementing the following issues and document the same. The documentation of bench-routes is the section following this.

To be done (need help):

1. **High priority** **A mechanism to auto-search the routes supported in a particular port (or web app):** Currently, the input of routes in an application is decided to be

manual. However, in real-world scenarios, organisations have routes in numbers of hundreds, sometimes thousands. In such scenarios, manual input is not a feasible approach. Hence, an algorithm to automatise this process is highly in need.

Outline

Resources

1. Golang
    a. Beginner level link (excludes goroutines)
    b. Standard link (nearly everything)
    c. Benchmarks of golang vs other languages link
2. Native ping support Linux kernel link
3. Jitter link

Terms

1. **Route**

Routes are paths to which a server listens or responds to, whenever the client pings (or requests) on that address. It is easy to confuse the route with the address (or URL).

Example:

- http://192.168.22.25:8080/**add_events**
- http://192.168.22.25:8080/**remove_events**

Here,
IP: 192.168.22.25
Port: 8080
Routes: [**'add_events'**, **'remove_events'**]

2. Monitoring

Monitoring refers to the act of constantly checking and observing all the routes at particular intervals, in order to check the status of the routes. The status of the routes can be running, stop, error or warning. In case of error or warning, the admin will be notified to check for the error and the log details.

3. tsdb

A time-series database (TSDB) is a database optimized for time-stamped or time-series data. Time series data are simply measurements or events that are tracked time-series and monitored. In these types of models, the data points are saved in stamps of time rather than any ID which is generally the case of SQL and NoSQL databases.

Why do we need a time-series database?

In traditional databases, values are stored simply as rows in tables or as an object, in a document. However, the data values in this project need to be stored in every fraction of a second. Use of traditional data doesn't seem to be convenient when it comes to storing and retrieving large amounts of data. Further, the traditional data models will increase time space and time complexity during the course of data extraction in sorted time order.

In a tsdb, the data is stored at a particular timestamp(at a particular instant of time) in which each record can be represented as a hash block. This block can be stored in the form of a chain of events, thereby forming a blockchain. This chain will be highly useful in making graphs on the UI end, as the time complexity of extraction of data points would be highly optimized (since the extraction of data would follow chain propagation).

4. **Benchmark**

   In computing, a benchmark is an act of running a computer program, a set of programs, or other operations, in order to assess the relative performance of an object, system or entity, by running a series of standard tests which are generally based on computationally expensive algorithms. Benchmarking determines the performance of the software or device.

Overview

Monitoring has been tough and with the increase in the routes used in any sophisticated project, the performance and metrics of an application are seriously affected. With an increase in server computational models, the probability of a complete request-response cycle without any throws is nowhere close to 1.

bench-routes acts as a routes-benchmarking, monitoring, and route-network analysis tool. It monitors the routes of the application and analyses the network pipe between the server-client.

Goals

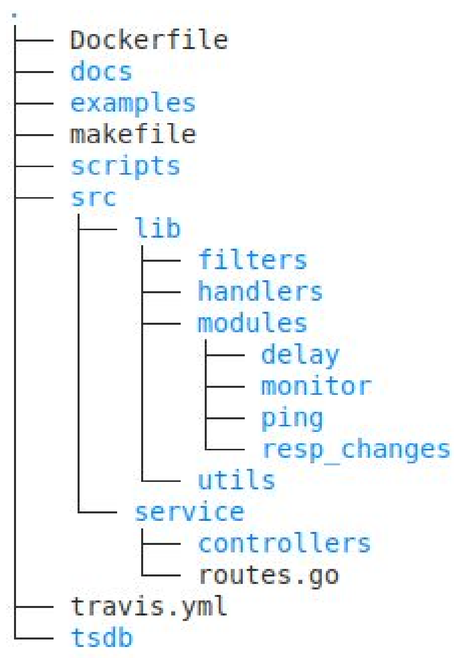
1. Benchmark route
   a. Load-handling of application on the individual route.
   b. Test various possibilities of data in params (Permutate params), like sending an empty param to see how the server response behaves.
2. Analyse network performance of the hosted application irrespectively of containerization
   a. Network ping
   b. Jitter analysis
   c. Packet loss
3. Log error handling capability of the application
4. Maintain a check on server-route output and alert on changes above the threshold
5. Graphical view using ElectronJS

Learning

1. Analysis of network-based calculations
2. Intensive use of channels (golang)
3. Familiarity with monitoring concepts
4. Time-series database(tsdb) [basic as of now]
5. Stronghold on Golang
6. Extensive use of go-subroutines

7. Simple blockchain concepts

High-Level Model



```
.
├── Dockerfile
├── docs
├── examples
├── makefile
├── scripts
├── src
│   ├── lib
│   │   ├── filters
│   │   ├── handlers
│   │   ├── modules
│   │   │   ├── delay
│   │   │   ├── monitor
│   │   │   ├── ping
│   │   │   └── resp_changes
│   │   └── utils
│   └── service
│       ├── controllers
│       └── routes.go
├── travis.yml
└── tsdb
```

Please see the Data Flow for a better understanding of the propagation of data from the above modules

Brief description:

docs: This directory would contain all the related documents to the project. It will also support the project documentation for general users.

examples: This would contain use-case examples of benchroutes. This may also include native CLI examples if we plan to have a cli version as well.

scripts: All shell and bash scripts related to setting up, building, execution or testing would be present in this directory. This would ease the development process.

src: This is the main directory that contains the development code. Majority of the development process would take place here.

lib: This directory will be responsible for the core implementation of ideas in golang.

service: Or in other words, server. This directory would contain all server related stuff like routes, controllers etc., which would run as a daemon.

tsdb: Directory implementing the time-series database specific to bench-routes.

Basic Idea Explanation

bench-routes is a GUI-powered highly scalable testing tool that monitors the performance of the routes in any web/routing - application. Bench-routes would perform a series of networking algorithms and calculations involved to find the real-time state of routes in an application. It monitors the routes at regular intervals and analyzes the response and the time involved in the same. The moment, the delta in response rises above a threshold limit, an alert would be sent to the admin of the respective manager.

Bench-routes (B) follows a basic idea of pinging the server at a respective route.

Consider a server **S** that contains routes **R** respectively.

Let,
r belongs to any route in **R**
t be any time constant
tres be a threshold above which the delta in response would lead to an alert to the admin
rmean be the mean value of responses from **r (**where **r < tres**, since if **r** exceeds **tres**, an alert would be sent as an exceptional behaviour of that particular route**)**

B sends a request to **S** at **r** route in every **t** instant. **B** logs the response to any local storage in t intervals after each response is received.

Consider **tres** be 0.7 or 70% respectively. This means that for every consecutive response if the difference of the response length from the mean response length is >= **tres**, would lead to alerting. **rmean** would update itself for every response < tres so that the **rmean** behaves ideally and updates itself with time, dynamic to changes.

Methods

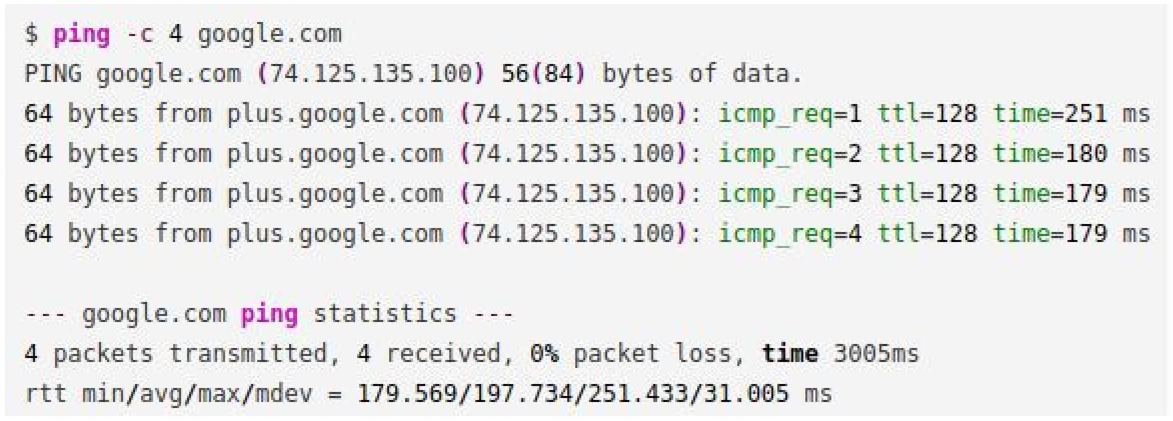
1. Linux ping command
    a. min/avg/max/mdev ping values
    b. Flood ping
    c. Jitter
2. Monitoring
3. Report response changes
4. Req-Res delay
5. Network congestion

Detailed Proposal

1. **Linux ping command***:* All Unix based system supports basic ping command right from their shell. This can be made into use by running it through a subprocess and pipe out the necessary details of our needs and scrap them accordingly. However, it is important to note

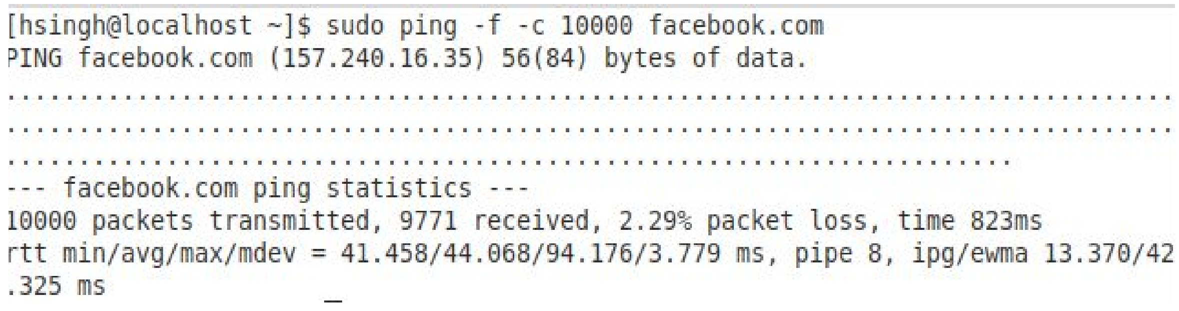
that, ping works on the raw IP as a pong from the hosted service which runs as default in any network card. Hence, this method is just to check the server response time, requests performance and could be helpful in checking bottlenecking states. Following ways can make this process more helpful:

a. **min/avg/max/mdev ping values***:* These are the general ping values after the ping subprocess would end its successful execution. This can be made to run in count mode so that only a particular number of pings are made to the mentioned IP.



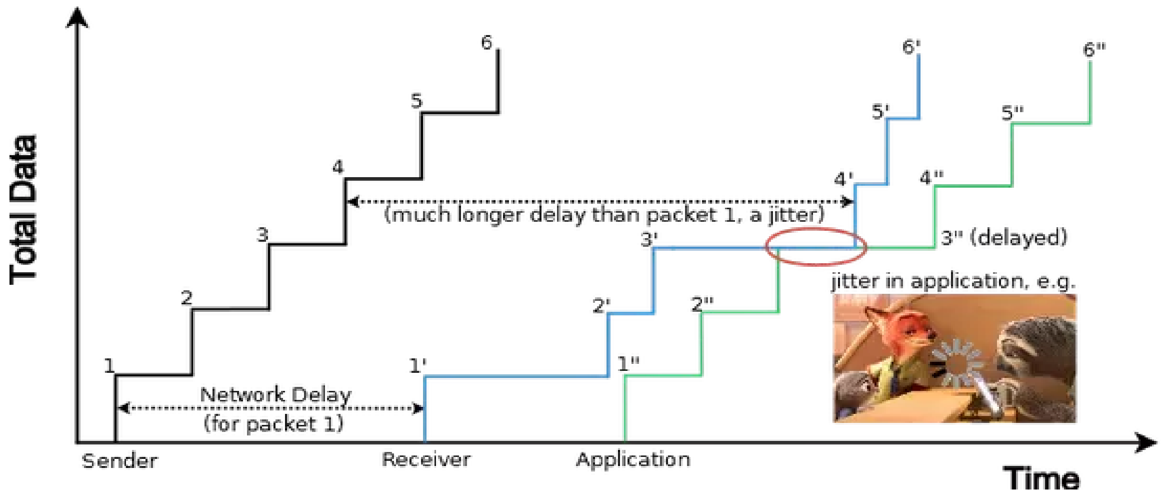
```
$ ping -c 4 google.com
PING google.com (74.125.135.100) 56(84) bytes of data.
64 bytes from plus.google.com (74.125.135.100): icmp_req=1 ttl=128 time=251 ms
64 bytes from plus.google.com (74.125.135.100): icmp_req=2 ttl=128 time=180 ms
64 bytes from plus.google.com (74.125.135.100): icmp_req=3 ttl=128 time=179 ms
64 bytes from plus.google.com (74.125.135.100): icmp_req=4 ttl=128 time=179 ms

--- google.com ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3005ms
rtt min/avg/max/mdev = 179.569/197.734/251.433/31.005 ms
```

b. **Flood ping***:* These would send a high number of requests (often in thousands or lakhs) to the IP route specified. This leads to more accurate results but requires sudo if the minimum interval is set to < 0.2 (default). Consider the example below, the count is set to 10,000 requests with the interval set as 0 (since not mentioned) and hence requires sudo.



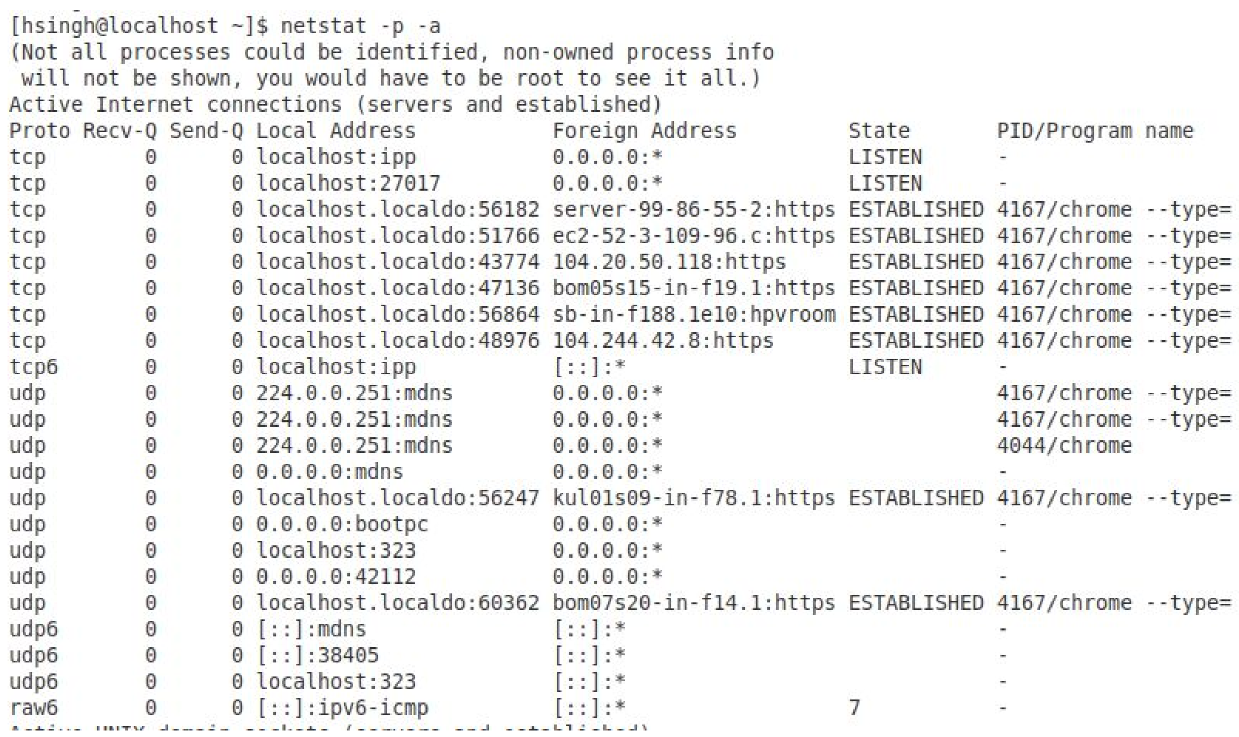
```
[hsingh@localhost ~]$ sudo ping -f -c 10000 facebook.com
PING facebook.com (157.240.16.35) 56(84) bytes of data.
.................................................................................
.................................................................................
......................................................................
--- facebook.com ping statistics ---
10000 packets transmitted, 9771 received, 2.29% packet loss, time 823ms
rtt min/avg/max/mdev = 41.458/44.068/94.176/3.779 ms, pipe 8, ipg/ewma 13.370/42
.325 ms
```

c. **Jitter***:* Jitter is the amount of variation in latency/response time, in milliseconds. Reliable connections consistently report back the same latency over and over again. Lots of variation (or 'jitter') is an indication of problems. Example: consider 5 ping samples: 136, 184, 115, 148, 125. Then, jitter found in the sample can be calculated by averaging consecutive differences in the pong response time as follows: 136 to 184, diff = 48; 184 to 115, diff = 69; 115 to 148, diff = 33; 148 to 125, diff = 23; The total difference is 173 - so the jitter is 173 / 4, or 43.25.



2. **Monitoring**: In a world of complex and sophisticated servers with several concurrent routes, monitoring remains an issue especially with the multi-deployment instance, handled by Kubernetes. This part aims to monitor the specific route and to alert the user in case of downstate.
3. **Report changes**: Extension of the above point, in which the response of the particular route would be saved as a time-series and a mean of the response length would be saved on each incoming request. On each response, the per-cent change in the response length would be compared to the mean length of the response of that route. If the change is beyond a threshold, an alert will be sent to the user, since a change beyond the threshold would indicate some error in the VM.
4. **Req-Res delay**: The delay or increase on the req-res time would lead to more likeliness of a request being a timeout and hence a bad performance functionally and non-functionally as well.
5. **Network Congestion:** Network Congestion is the situation which occurs when the entire bandwidth of the network is occupied. For better understanding, consider a water flow pipe. When the water in the pipe has the highest Reynold number (in the situation of highest pressure output), the pipe can be said to be congested, i.e., any other stream of water cannot be passed through it. A network can be related in the same way. Hence, any request in a congested network will be bound to get slow and sometimes may lead to time-out, resulting in termination of operations.

   Checking congestion in a network can be carried out by using **netstat.** The following is sample output from netstat command.



```
[hsingh@localhost ~]$ netstat -p -a
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 localhost:ipp           0.0.0.0:*               LISTEN      -
tcp        0      0 localhost:27017         0.0.0.0:*               LISTEN      -
tcp        0      0 localhost.localdo:56182 server-99-86-55-2:https ESTABLISHED 4167/chrome --type=
tcp        0      0 localhost.localdo:51766 ec2-52-3-109-96.c:https ESTABLISHED 4167/chrome --type=
tcp        0      0 localhost.localdo:43774 104.20.50.118:https     ESTABLISHED 4167/chrome --type=
tcp        0      0 localhost.localdo:47136 bom05s15-in-f19.1:https ESTABLISHED 4167/chrome --type=
tcp        0      0 localhost.localdo:56864 sb-in-f188.1e10:hpvroom ESTABLISHED 4167/chrome --type=
tcp        0      0 localhost.localdo:48976 104.244.42.8:https      ESTABLISHED 4167/chrome --type=
tcp6       0      0 localhost:ipp           [::]:*                  LISTEN      -
udp        0      0 224.0.0.251:mdns        0.0.0.0:*                           4167/chrome --type=
udp        0      0 224.0.0.251:mdns        0.0.0.0:*                           4167/chrome --type=
udp        0      0 224.0.0.251:mdns        0.0.0.0:*                           4044/chrome
udp        0      0 0.0.0.0:mdns            0.0.0.0:*                           -
udp        0      0 localhost.localdo:56247 kul01s09-in-f78.1:https ESTABLISHED 4167/chrome --type=
udp        0      0 0.0.0.0:bootpc          0.0.0.0:*                           -
udp        0      0 localhost:323           0.0.0.0:*                           -
udp        0      0 0.0.0.0:42112           0.0.0.0:*                           -
udp        0      0 localhost.localdo:60362 bom07s20-in-f14.1:https ESTABLISHED 4167/chrome --type=
udp6       0      0 [::]:mdns               [::]:*                              -
udp6       0      0 [::]:38405              [::]:*                              -
udp6       0      0 localhost:323           [::]:*                              -
raw6       0      0 [::]:ipv6-icmp          [::]:*                  7           -
```

Note the columns Recv-Q and Send-Q. These two are the most important. The numbers in the two columns represent the **leftover packets.** Let's understand what **leftover packets** are.

In any system, **Recv-Q** and **Send-Q** are sockets which are responsible for inbound and outbound connections. Consider them as buffers. Whenever any program has to send information to the external network (or internet), the information is broken down into small transferable packets which are atomic in nature. These packets are then sent to **Send-Q** buffer. In **Send-Q** buffer, the packets are queued and the initial link is established to the corresponding foreign address. Once the link is established, these packets are then transferred one after another (serially) and removed from the buffer. Naturally, those packets which could not be transferred to the foreign address (or external network) would be **leftover** in the buffer. This could be due to various reasons, the most common and important being the congestion of outgoing network bandwidth.

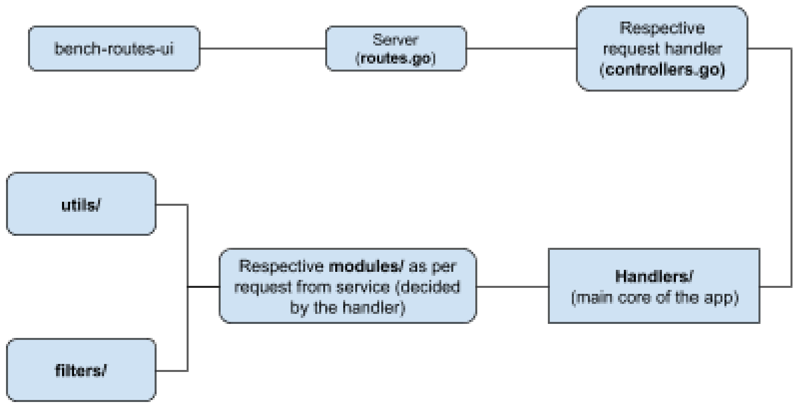
Similarly, when a packet is received by the system, it gets stored in the **Recv-Q** buffer. Once the storing of the batch of incoming packets is done, the requested listening programs are signalled the arrival of the requested packets. These programs then collect the required packets. Ideally, the programs should collect all the packets from the buffer which they have requested while sending from the **Send-Q** buffer unless some internal error has occurred in the program. This leads to **leftover packets** in the **Recv-Q** buffer and hence indicates some issue in the internal network (or local network or localhost).

Hence, it must be clear by now, that for the successful execution of a request-response cycle, the **Recv-Q** and **Send-Q** must show values of **0** (i.e., **0 leftover packets** in the respective buffer). Now about the implementation. This approach can be done on the host system only. Hence, we need to make an independent script which would be deployed on the host server and that would be listening on some port X. Bench Routes would ping X in regular intervals to collect data and analyse whether the server is in congested state of network or not. It is obvious to have doubt that if in case the network is already congested, the ping to X might time out. Yes, that's true. To avoid this situation, the request needs to be highly optimised. Moreover, this timeout will itself indicate the congestion and once the congestion improves, the stats or reason could be studied from the PID or PName (as in the event of failure, the benchroute would still continue to ping until a response is received. Once the response is received, the entire picture will become clear).

Data Flow

Please read this as a continuation from High-Level Model.

The flow of data right from the requesting the bench-routes service by the front UI until the final response to the front UI is as follows. (All lines indicate bi-directional propagation, and **/** indicates a directory as related with the High-Level Model)


bench-routes-ui
Server (routes.go)
Respective request handler (controllers.go)
utils/
Respective modules/ as per request from service (decided by the handler)
Handlers/ (main core of the app)
filters/

Implementation

The implementation of the above-proposed idea would be carried out in the Go programming language. This project will launch itself on boot and work as a daemon service. The benchroutes-UI, an electron app(would be later discussed), will be responsible for the graphical aspect of the project.

Queries

1. How will a user be able to provide the routes in the software to **bench-routes?**

Ans. The user can provide the required details about the software i.e. the IP address and the routes in the GUI of bench-routes.

2. Does the user have to import the package in the software?

   Ans. No, as mentioned in explanation, **bench-routes** is a GUI tool, the user has to provide the IP address and the routes of the server In the GUI interface.

3. Is there any automation for finding out routes, as many sophisticated applications contain millions of routes in a single web-application?

   Ans. We plan a native mini-package of every possible language which would contain a fixed hidden route. The programmer will have to pass an array of strings to the imported function. When bench-routes is initiated, as the IP address is provided, the service will automatically ping the fixed hidden route and get the list of routes available and initiate the testing and monitoring process on it.