Project Report

GTR: Global Web-based Visual Traceroute

Kelong Wang
Computer Science
Stony Brook University
kelong.wang@stonybrook.edu

1. INTRODUCTION

Traceroute is a network diagnostic tool for displaying route (path) between two IP address[1]. Typically, this tool is shipped as a command line utility on various OS distributions. To extend the tool as practical as possible, the Global Web-based Visual Traceroute (GTR) project is proposed to build a web platform that supplies extra information of the path than the traditional traceroute does. As the name suggests, this platform will enable users to issue traceroute queries from specific source probes and do the path plotting on maps (visualization).

This report is structured as follows. Section 2 describes the goals, design and several challenges we are about to solve. Section 3 describes the detailed implementation of GTR components. Section 4 summarizes this project.

2. GOALS & DESIGN

The primitive goals of GTR is clear:

1. User can submit a traceroute query on GTR website;
2. GTR will do the traceroute for the user;
3. GTR will visualize the traceroute result;
4. User may interact with the visualization;
5. GTR may provide more information based on the result.

2.1 RIPE Atlas Integration

For this project, we want to achieve the globalized traceroute, which means user can decide which country (or location) to start the traceroute. One approach is to implement and deploy the probe agent on partners' hosts. But in the short term, it is hard to find such resource. In this project, we are granted the access to the third-party network measurement service – RIPE Atlas[2] in which we can submit traceroute request and fetch the result later via their restful HTTP API[3]. This service reduces our work and we can focus more on the visualization.

Based on the features of RIPE Atlas service, we can start to design GTR's work flow, since GTR are more or less acted as a wrapper of Atlas besides the plotting component. Imagining GTR receives user's request, forwards the request to RIPE Atlas and waits until the result is available and finally returns the result to user.

In this flow, the very first step is to generate the request that is made up of two elements – source and destination. The source probe selected by user is from a probe list. There are two ways to get this list. First, we can simply give a country list. Second, we can query the RIPE API to fetch several active (online) probes. For the first approach, RIPE will recognize the country code and randomly pick one active probe to issue the traceroute. GTR does not adopt this way because we want to give user more choices in certain large countries (instead of randomly picking by RIPE). For the latter approach, the drawback is that it takes some time because we have to query multiple times (one country at a time) to get the full list. Moreover, we have to keep the list up to date to decrease the probability that users select an inactive probe. Actually combination of two approaches maybe a practical implementation but we should take care that some small countries only have 1 or 2 probes which could be inactive at the same time or not that responsive. The destination of the request can be IP address or domain name. Both forms can be easily entered incorrectly such as a non-existing domain. The invalidation is expected to be examined and returned by RIPE Atlas, which we should handle properly.

Follow the flow, we get to the important step in which we forward the request and then fetch the result when it is available. Definitely, there will be several considerable time gaps comparing with the short time when users submit a request to GTR:

1. GTR submit the request with HTTP API;
2. RIPE dispatch the request to specific probe;

[1] http://en.wikipedia.org/wiki/Traceroute

[2] https://atlas.ripe.net/

[3] https://atlas.ripe.net/docs/measurement-creation-api/

3. Probe conducts the actual traceroute;

4. RIPE generate the final result in JSON;

5. GTR check and fetch the available result.

By the nature of this asynchronous processing, we have to design our system to adapt itself smoothly, which will essentially improve the user experience. The whole procedure may take several seconds. We do not want user's request to be blocked until the result is available. Instead, a practical way is to make the original request immediately success and let the user agent (browser) check the result periodically with a friendly interface. In addition, we can also provide an address from which user can retrieve the result in the future, especially for the users who are not willing to wait several seconds for the incoming result after submitting. When it comes to the forwarding and fetching via RIPE Atlas API, certain programs should be responsible to do this job on the background. All these requirements suggest that we can leverage a permanent central database to link things up. We will describe the detailed implementation in later section.

A minor but necessary step is to prevent the abuse of this service. E.g., user may submit the request repeatedly. Since each traceroute by RIPE Atlas will cost some credits[4], we should introduce a sort of quota mechanism. At least, we can simply deny the submission if the number of requests from certain IP excesses the limitation for a configured interval.

2.2 Plotting on Map Service

Visualization is the main feature of GTR. For a given traceroute result, we have the IP address of each hop. If we have the geography database which maps from IP to location, we can directly draw the path connecting each hop. In addition, we may want to put more information such as autonomous system and round trip time (RTT) on the plotting. To keep the plotting clean, we could make it interactive and display the related information as needed.

In practice, it is possible that the traceroute result is not well formed for the plotting. We have to deal with these special cases. Below are possible situations:

1. Some hops (usually the first several hops) may be private addresses such that we do not know their locations.

2. Some hops have no addresses. This may happen when the route in that hop drops the ICMP request or the response packets get lost.

3. For some hops, we have no location information of their addresses or the information is not accurate. As the hops are usually the ISP routes, we may only know the location in country or region level.

We may experience with more unexpected situation through the feedback of users.

[4]https://atlas.ripe.net/user/credits/

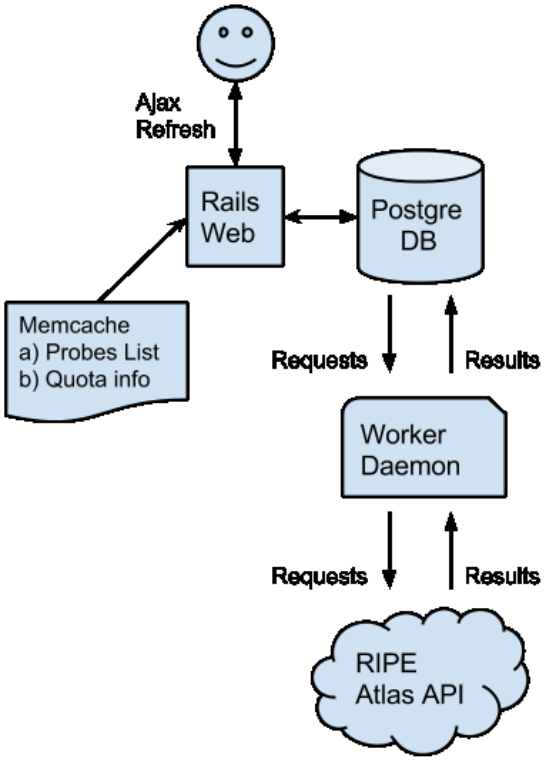

Figure 1: Flow of the Asynchronous Processing

3. IMPLEMENTATION

GTR is a typical web project (client-server mode). Figure 1 presents the overview of the whole procedure (except the plotting). In this flow, users' requests will be forwarded to RIPE Atlas. And the results will finally be returned to users (browsers).

In next several subsections, we will look at each component in detail.

3.1 Geo IP Database

Geo IP database is the most important component that determines the accuracy of the visualization. But it is also the component which we cannot control much. Currently, we will use a relatively complete and popular database. This database is combined with a free geo-location IP database and an IP to ASN (Autonomous System Number) database. Both are lite version from MaxMind[5]. For maintainability, the database can be periodically updated with a background program. Actually they update the lite version every month. In plotting section we will describe how we deal with the inaccuracy of the database.

3.2 Asynchronous Processing

The web part is implemented with Ruby on Rails[6](or Rails), a MVC (Model-View-Controller) web framework. For fast front-end development, we adopt several techniques by which we could write less code and concentrate on the business logic:

[5]http://www.maxmind.com/en/opensource
[6]http://rubyonrails.org/

1. Slim[7] is a template engine to generate the HTML document. We replace Rails' default engine with Slim to make the view's code clear and more readable.

2. Bootstrap[8] is CSS framework which enables developer to use pre-defined styles to organize the web object without writing from scratch. It also provides a dozen of JavaScript (jQuery) plugins to enrich the forms of display.

Rails supports various relational databases to store the instances of models. In this project, we choose PostgreSQL[9] as its backend database. Around the Rails, we also run several daemons to do background jobs. These daemons shares the same environment with Rails and can fetch the data with Rails ORM interface so that we can write Ruby codes instead of SQL.

3.2.1 Probe List Preloading

The probe list is used in submission form where user can select the source of the traceroute on the portal page of GTR.

As mentioned before, loading probe list with RIPE Atlas API consumes considerable time depending on how many countries we are going to fetch. We do not have to fetch this list each time a user comes. So we preload this list in memory (Memcached[10]) and keep it for 30 minutes. After the list expires, we will load it again to get the recent active probes.

3.2.2 Ajax Refreshing

On the portal page, users will select the source probe and enter the target IP (or domain name). After users click "submit" button, Rails controller will create the task in database and immediately send back the unique id of the task. The browser will then execute the Ajax code periodically (every 3 seconds) to check the status of that task. Based on the JSON response, Ajax will behave differently:

1. Result available: JavaScript will reload the whole page to do the plotting.

2. Result not available: Do nothing; Continue to display the loading picture.

3. Traceroute failed: JavaScript will display the failure message carried in the response and stop requesting.

3.2.3 Task Structure

The submitted task and its associated result will be stored permanently in the database. The result of each task is a sequence of hops.

The structure of task and hop are defined as follows. Usage of these fields will be described in the next section.

[7]http://slim-lang.com/
[8]http://getbootstrap.com/
[9]http://www.postgresql.org/
[10]http://memcached.org/

Table 1: Task Definition

Field	Type	Description
id	integer	internal self-incremental id
uuid	string{40}	unique id of the task(for user)
probe	string{10}	source probe id of RIPE Atlas
dst	string{40}	destination (user's input)
dst_addr	string{40}	resolved address
msm_id	integer	measurement id of RIPE Atlas
submitted	boolean	whether the task is submitted to RIPE Atlas (index)
available	boolean	whether the result is available (index)
failed	boolean	whether the submission is failed or the task is outdated (index)
created_at	timestamp	task creation time
updated_at	timestamp	task updating time
endtime	timestamp	traceroute end time

Table 2: Hop Definition

Field	Type	Description
id	integer	internal self-incremental id
traceroute_id	integer	id of traceroute (foreign key)
no	integer	hop number
from	string{40}	hop IP address
rtt	float	round trip time of that hop

3.2.4 Task Forwarding & Fetching

Task forwarding and result fetching are processed on background with two daemons: submitter and fetcher. The submitter will continuously forward the tasks from database to RIPE Atlas as long as the tasks are not submitted and are not marked as failed:

```
loop {
  Traceroute.where(:submitted => false,
                   :failed => false).each do |tr|
    if Time.now - tr.created_at > 600
      tr.mark_failed
      next
    end
    tr.submit
    sleep 0.1 # interval between each POST
  end

  sleep 1
}
```

Once submitted, we will change the flag ("submitted") to avoid the re-submission. RIPE will return a measurement id ("msm_id") by which we can fetch the result later. Notice that before submitting to RIPE, we also check the "created_at" timestamp to drop the outdated tasks. This could be useful when we find the database has some old tasks that were not submitted before and we will simply discard them.

In a similar way, the fetcher will check the result from RIPE Atlas periodically with a slightly different condition:

```
loop {
  Traceroute.where(:available => false,
                   :submitted => true,
                   :failed => false).each do |tr|
    if Time.now - tr.created_at > 600
      tr.mark_failed
      next
    end
    tr.fetch
    sleep 0.1
  end

  sleep 5
}
```

The fetcher only checks the submitted task whose results are still not available. If a task is marked as failed, we will not check it. Before fetching, we also examine if the task is outdated. This may happen when we submit a task but cannot fetch the result in 10 minutes. Thus we return the failure to user-agent to stop its further Ajax requests.

The inner loop will be executed every 5 seconds. This interval is not long because the RIPE Atlas usually takes at least 20 seconds to accomplish the traceroute.

3.2.5 *Quota limitation*

If one user submits the traceroute requests too frequently, we will deny the submission. We use a simple mechanism to limit the abuse:

```
count = Rails.cache.increment(request.remote_ip,
                              1,
                              :expires_in => 600)
if count > REQS_LIMIT
  @alert_msg = "Please try later."
  return
end
```

Through the "Rails.cache" method, a key-value pair will be stored in memory. The key is the user's IP address, while the value is the counter of requests by that user. We make this pair expires in 10 minutes and set a threshold (REQS_LIMIT). In this way, we can limit the number of requests within 10 minutes for each user.

3.3 Plotting

Visualization of a traceroute path is implemented on Google Maps service with its JavaScript API[11]. There are three kinds of object in Google Maps.

1. Info-window is a container which includes a snippet of HTML. This is used to display information (such as country / city / ASN) of each hop.

2. Marker on maps represents the location of each hop. The markers can be clicked to open info-window.

3. The line connecting two markers represents the path between two hops. We can decorate the line with different colors or draw a dashed line to indicate the "gap".

Following the Ajax refreshing, when the result is available, the full hop list will be returned to browser wrapped in JavaScript. Basically, by looking up latitude and longitude from Geo IP database for each hop, a sequence of markers and lines are drawn with related description in a hidden info-window. The drawing procedure is as follows:

1. Step 1: Find the first hop which has latitude and longitude, draw the corresponding marker (each marker will be bound a info-window, same below). This will skip the private hops, the hops without location information and especially the gaps that are made up of one or more stars ("*").

2. Step 2: Find the next hop with location information and the location should be different with previous marker. Then draw its corresponding marker. This will also skip the private hops and gaps. Moreover, this step will skip the hops that have the same location with previous marker. If we do not skip, the markers will overlap which is meaningless for users. However, we will provide a table to show the full hop list where users can click the specific hop to draw it out.

3. Step 3: Draw the line connecting the previous marker and the current marker in Step 2. If we do skip one or more hops in Step 2, we draw a dashed line here. If the two markers are two consecutive hops, we draw a solid line.

4. Step 4: repeat Step 2 & 3 until we reach the end of the hop list.

There are some additional modes / features can be enabled. We describe them in next two sections.

3.3.1 *RTT Display*

A useful feature is to display the round trip time (RTT, or we roughly call it latency) on maps. We use different colors on the lines to reflect the latency of the path.

Some users may want to know the bottleneck of the path, that is, on which specific hop the latency increase significantly. For this requirement, we will calculate the difference of the RTT between each two markers (hops). Then we compare the difference with certain threshold and draw the line in different colors. For example, in Figure 2, the middle segment is drawn in red because the RTT difference of these two markers is more than 50 milliseconds that we think it a bit slow. For the rest segments that have low RTT difference, we keep it in default color (blue). This mode is called relative RTT mode.

Another mode is absolute RTT mode (Figure 3). In this mode, we decide the color based on the absolute RTT of each hop. This will present the overview of the path's latency.

[11] https://developers.google.com/maps/

Figure 2: Relative RTT Mode

Figure 3: Absolute RTT Mode

When trying to display the latency for some traceroute instance in relative RTT mode, we find there are some cases or challenges that the latency cannot be presented completely. Because it is common that several hops are in the same location and thus there are no lines between them, and meanwhile RTT bump happens within these overlapped hops.

3.3.2 Sanity Check

When we view the final plotting of many traceroute requests, we usually notice there are some strange hops that are not that reasonable. For example, in Figure 4, we suspect that the marker in the center of the country is not accurately located. After we observe other similar plotting, we realize that these hops are always in the center of their countries because the geo database do not have accurate location information for them but only know their countries. The characteristic of these hops' geo information is that the region and city fields are empty. We regard these hops as inaccurate hops and will remove their latitude and longitude information in a special mode. As a result, with the same drawing step, these hops will be skipped and dashed lines will be drawn. Figure 5 shows the path after this sanity check. On the maps, we provide a button to enable or disable the sanity check.

There is a situation that we do not want to hide the inaccurate hops. For example, in Figure 3, the marker in the center of the map is an inaccurate hop in France. We will not hide this hop in sanity check because it is important for user to know that the path is across a third country. So we add a condition that if the inaccurate hops are the only hops in their countries, we do not hide them.

3.4 Misc Features

There are some minor features implemented as well.

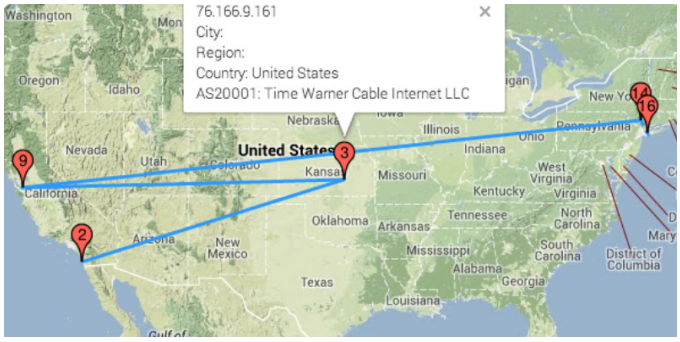

Figure 4: Before Sanity Check

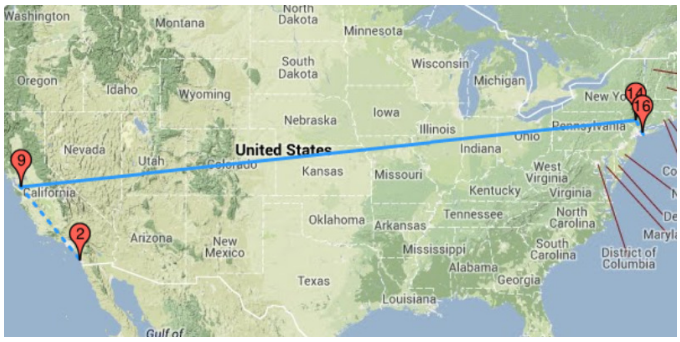

Figure 5: After Sanity Check

- Users can upload (paste) their own results on the portal of GTR. We will do the plotting for the users.
- Beside the map, we have a table listing each hop. By moving the cursor, users can view the detailed description of each hop in the popover window. As mentioned earlier, some hops are not drawn because of overlapping. Through this table, users can click each row to draw the hop on maps. The new marker will be put on top of previous one. The clicking also serves as a feature of navigation, i.e., the map will make the clicked marker in the center of map.
- Beside the hop table, users can switch the tab to view the AS path instead of original hops (see Figure 6).
- After users submit the traceroute requests, we will generate a permanent URL (which contains "UUID" of tasks) for them. With the URL, users can retrieve the result later on.

4. SUMMARY

In this project, we implement a usable web-based visual traceroute on top of RIPE Atlas service.

In the future we can improve GTR in many aspects:

- The accuracy of geo database can be always improved. This is a large topic. But through GTR, if we can find certain ways to distinguish the inaccurate information, it will be a killer feature.

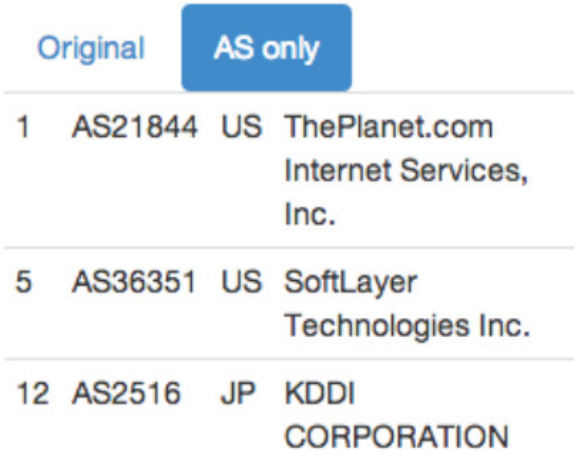

Figure 6: AS Path

- Sanity check with speed of light is a reliable way to examine the situation in which two markers are not accurately located because of the violation of physics.
- In GTR we only send out one packet for every specific TTL. If we send more (e.g., three packets), the path may be more complicated and reveal something interesting.
- We may improve the compatibility to support third-party traceroute result (such as iPlane[12]) format.

[12]http://iplane.cs.washington.edu/