ULg Library
http://lib.ulg.ac.be
matheO
http://matheo.ulg.ac.be

Master thesis : Fast Internet Topology Discovery

Auteur : Ernst, Bertrand
Promoteur(s) : Donnet, Benoît
Faculté : Faculté des Sciences appliquées
Diplôme : Master en sciences informatiques, à finalité spécialisée en "computer systems and networks"
Année académique : 2016-2017
URI/URL : http://hdl.handle.net/2268.2/2572

Avertissement à l'attention des usagers :

Tous les documents placés en accès ouvert sur le site le site MatheO sont protégés par le droit d'auteur. Conformément aux principes énoncés par la "Budapest Open Access Initiative"(BOAI, 2002), l'utilisateur du site peut lire, télécharger, copier, transmettre, imprimer, chercher ou faire un lien vers le texte intégral de ces documents, les disséquer pour les indexer, s'en servir de données pour un logiciel, ou s'en servir à toute autre fin légale (ou prévue par la réglementation relative au droit d'auteur). Toute utilisation du document à des fins commerciales est strictement interdite.

Par ailleurs, l'utilisateur s'engage à respecter les droits moraux de l'auteur, principalement le droit à l'intégrité de l'oeuvre et le droit de paternité et ce dans toute utilisation que l'utilisateur entreprend. Ainsi, à titre d'exemple, lorsqu'il reproduira un document par extrait ou dans son intégralité, l'utilisateur citera de manière complète les sources telles que mentionnées ci-dessus. Toute utilisation non explicitement autorisée ci-avant (telle que par exemple, la modification du document ou son résumé) nécessite l'autorisation préalable et expresse des auteurs ou de leurs ayants droit.

UNIVERSITY OF LIÈGE
FACULTY OF APPLIED SCIENCES

ATFE0002-1: Master thesis

Fast Internet Topology Discovery

Graduation Studies conducted for obtaining the Master's degree in Computer Science
by Bertrand Ernst under the supervision of Prof. Benoit Donnet

Academic year 2016-2017

June 2017

Contents

Contents 2

Introduction 5

1 Background 7
- 1.1 Topology 7
- 1.2 Probing and Traceroute 8
- 1.3 Techniques survey 10
  - 1.3.1 SNMP, DNS and broadcast ping based techniques 11
  - 1.3.2 Traceroute-based techniques 11
  - 1.3.3 What's next? 15

2 YARRP 17
- 2.1 Problematic behind existing traceroute-based approaches 17
- 2.2 YARRP design 18
  - 2.2.1 Preprocessing step 18
  - 2.2.2 Probing step 20
  - 2.2.3 Postprocessing step 22
- 2.3 Limitations 23

3 Improving YARRP 27
- 3.1 Improved preprocessing step 28
  - 3.1.1 Filtering input IP addresses 28
  - 3.1.2 TTL window restriction 30
- 3.2 Improved probing step 31
  - 3.2.1 Communication between the sender and the receiver 31
- 3.3 Improved postprocessing step 33
  - 3.3.1 Sorting beforehand 33

4 Optimised YARRP 35
- 4.1 Large-Scale Measurement Campaign 35
- 4.2 Results 36

Conclusion **39**

Bibliography **41**

Introduction

To this day, performing Internet-scale probing campaigns in order to obtain a "snapshot" of the Internet topology remains a laborious task both in terms of time and accuracy. Indeed, probing substantial parts of the IPv4 address space using classical Traceroute-based approaches can take up to several days. Moreover, the Internet is quite poorly instrumented in terms of measurement mechanisms and Internet Service Providers (ISP) usually tend to hide topology information [1].

Being able to obtain topological maps is of great importance as numerous network researchers and administrators rely on these maps to prevent infrastructure problems from happening or even to develop security policies for instance.

In 2016, Robert Beverly introduced YARRP [2]. YARRP stands for "Yelling At Random Routers Progressively". It is a new way of probing that can considerably reduce the required probing time thanks to stateless operations. This Master thesis heavily relies on the tool that was introduced by this paper. By using it, we hope to address one of the major issue related to current large-scale probing techniques: the speed. Indeed, as we will see in chapter 2, YARRP enables quite fast large-scale probing campaigns owing to stateless operations.

However, YARRP is subject to a major flaw: its redundancy. Indeed, by re-implementing YARRP as described in Dr. Berverly's paper, we have found that even though it enables fast probing campaigns to be made, it is a very aggressive probing method in terms of redundancy. For instance, measurements made with the original version of YARRP have a mean uniqueness rate of roughly 8% meaning that over 100 responses received, there were only 8 different source IP addresses. That is why, in this Master thesis, we first propose to re-implement YARRP from scratch in order to validate it (we will call it "the original version [of YARRP]")[1]. Then, we will try to improve it in no less than three different ways. Obviously, we will evaluate the impact of each of these improvements independently as well as discuss their advantages and drawbacks. In order to evaluate them, we will use several metrics (*e.g.* the number of probes sent, the response rate or the uniqueness rate). These metrics will be properly defined in chapter 3. Lastly, we will carry out a larger measurement campaign using our improved version of YARRP in order to see how well it performs compared to the original version.

In the end, we have been able to develop an improved version of YARRP that is stable as well as less aggressive than the original one. For instance, we have doubled the uniqueness rate and increased the response rate from 34% to more than 80%. Obviously, all the improvements have been made by keeping in mind that the probing step should remain as fast as in the original version of YARRP.

[1] The full code is available for download here: `http://tfe.bernst.be`

This Master thesis in organised in four chapters.

1. The first one will set the scene by redefining the key concepts we will need such as the different levels of a network topology or even Traceroute. It will also discuss the state-of-the-art of probing techniques.
2. The second one will present the original version of YARRP as it was originally imagined by Robert Beverly in his paper. Obviously we will base the presentation of the original version of YARRP on our own re-implementation of the tool.
3. In the third chapter, we will discuss our three different improvements over the original version of YARRP, namely: filtering input IP addresses, TTL window restriction and the sender/receiver communication.
4. Last but not least, the fourth chapter will discuss a larger and certainly more realistic measurement campaign on which we ran our improved version of YARRP.

Chapter 1

Background

1.1 Topology

A network topology is a representation of the connections that exist between peers in a network [3]. It is usually represented as a graph (in other words, a set of nodes and a set of edges). The semantic associated with the nodes and the edges of the graph differ depending on which level the network is represented. There are typically three different levels at which a network topology can be represented.

- **The link layer topology** is the level at which the nodes are considered to be interfaces of link layer devices and end-hosts. The edges represent the different physical connections that exist between them. Being aware of the link layer network topology is of great importance when it comes to network management as, for instance, it can help with the diagnostic process in case of failure.

  An application example would be the Microsoft Link Layer Topology Discovery Protocol (LLTD) [4]. It is a protocol that is included in Windows since Windows Vista and that is used to display a graphical representation of the Local Area Network to which the computer is connected.

- **The network layer topology** (also known as the Internet topology) is the level on which this Master thesis is mostly focused on. It can itself be represented at four sub-levels.

  - **The IP interface level** represents the IP interfaces of the routers and end-hosts as nodes and the connections between them as edges.
  - **The router level** represents, as its name indicates, the routers as nodes and the connections between them as edges. It is in fact very similar to the IP interface level in which we would have regrouped the interfaces belonging to a same router into one node. The process is called *alias resolution* [5].
  - **The PoP level** takes into account the geographical location of the routers and gather the ones that are close to each other into one node. The nodes therefore represents what is called a Point of Presence (PoP).
  - **The AS level** represents the nodes as ASes and the edges between them as the relationships that exists between the ASes. An Autonomous System (AS) is a set of Internet-connected sub-networks that have a coherent routing policy and that are usually under the control of a same organisation. [3] The relationships between ASes often are business relationships. For

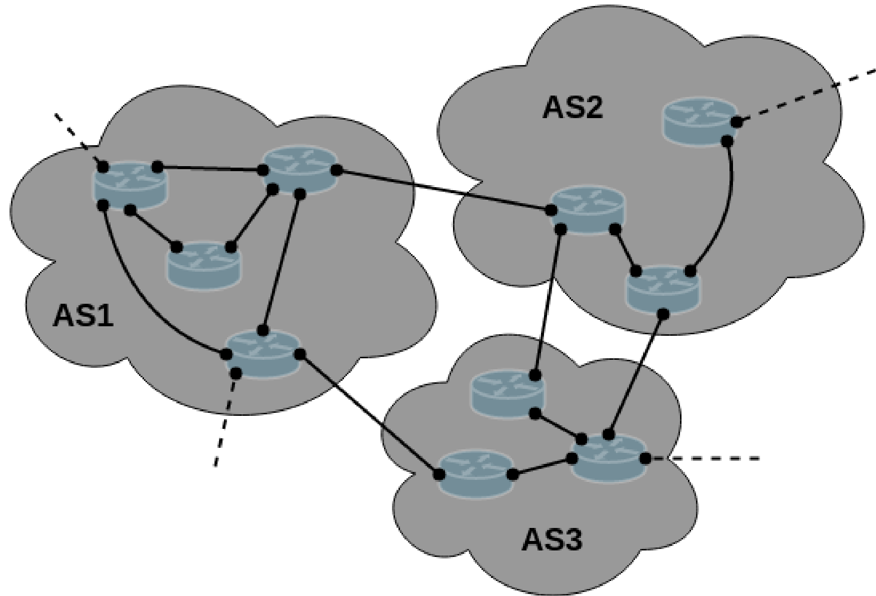

Figure 1.1: Three levels of the Internet topology

instance, in the case of an ISP and its client, we have a provider-to-customer relationship (or p2c).

Figure 1.1 represents three levels of the Internet topology. The IP interface level is obtained when only considering the black dots and the links between them. The router level is illustrated by regrouping every interface belonging to a same router into one blue router icon. The AS level is represented by the grey clouds and the links between them.

- **The overlay topology** refers to the level at which the nodes are peers and the edges are the logical connections between them. An overlay network is a network that is built on top of another. For instance, the Internet, at its origin, was built as an overlay of the telephone network. Typical applications that use an overlay network are Peer-to-Peer applications and Client-Server applications.

1.2 Probing and Traceroute

In this section, we will give definitions for two key concepts that are used in this work: **probing** and **Traceroute**.

Probing a network is the process of acquiring data for the purpose of learning something about the state of this network. There are two types of probing: passive and active. Passive probing refers to probing without any interaction with the network being monitored. As an example of such a passive probing technique, EXPOSURE [6] is a tool that was introduced in 2014 and that performs large-scale analysis on DNS records in order to detect malicious domains. By opposition, active probing refers to probing methods that send and receive packets (called **probes**) from and towards IP addresses from the network of interest in order to perform measurements. When doing so, one must be aware that an abuse of an active probing technique can result in a lot of packets being sent and received and could therefore be considered as an attack by a remote network administrator.

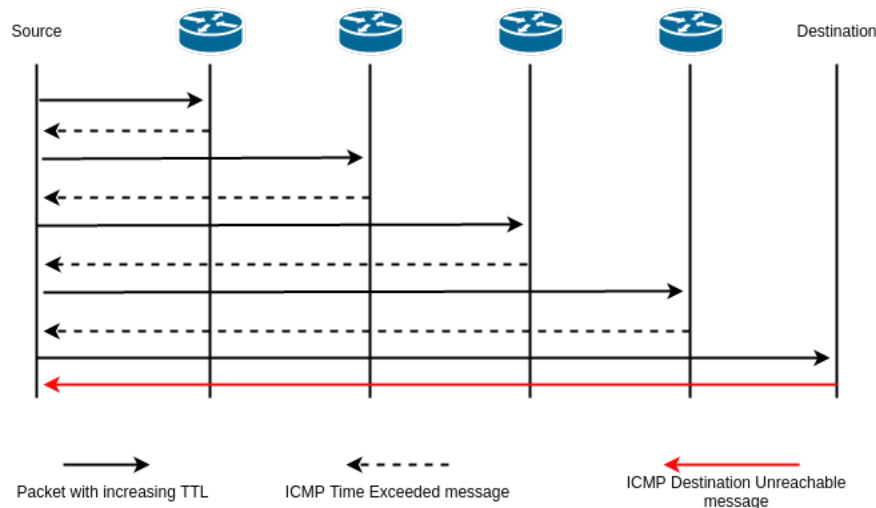

Figure 1.2: The different packets exchanged when using Traceroute

Traceroute [7,8] is a typical example of an active probing method that allows to discover the interfaces along a path from a source to a destination. Figure 1.2 illustrates the way Traceroute works. Depending on the implementation, Traceroute can rely on a variety of different protocols including ICMP, TCP and UDP. For instance, by default, Windows "tracert" uses ICMP while both Mac OS X and Linux "traceroute" commands rely on UDP. However, all versions of Traceroute use the ICMP Type 11 (Time Exceeded) messages in order to discover the hops along the path. It sends packets from the source towards the destination with increasing TTL values (the black straight arrows). Whenever a router receives a packet, it decreases its TTL values. If this value reaches 0, the packet is considered to have consumed enough resources on the network and is thus dropped. An ICMP Time Exceeded message is sent by the router to notify the source that its packet has been dropped. Thanks to this ICMP message, the interface with which it was sent is now known by the source. Eventually, when the probe reaches the destination (the last black straight arrow), the destination will, if the probes relied on UDP, reply with an ICMP *Destination Unreachable* message with the code *Port Unreachable* (the red arrow) due to the fact that the probe specified a high and supposed unused port number. If on the other hand the probes were ICMP packets, the destination should reply with an ICMP Echo Reply. Each time it sends a probe, the source activates a timer. If the timer expires and there has been no response for that particular TTL, then the router is considered as anonymous. It is also possible that the destination does not respond to probes (for instance if it is behind a firewall that blocks ICMP packets). In such case, an upper bound is used on the number of consecutive anonymous routers.

Traceroute has several variants, one of which is **Paris Traceroute** [9]. It has been developed because Traceroute has a flaw when it comes to load balancers. These machines have the responsibility of balancing the workload onto several computing resources (hence "load balancers"). They are typically used when packets need to be balanced between different links in order to avoid congestion (thus assuring a certain Quality of Service). Indeed, in the presence of load balancers along a path, Traceroute could output a path that does not exist in reality. Figure 1.3 illustrates that flaw. Indeed, using classic Traceroute, each probe will be sent from the source (S) to the destination (D) but it it not possible to predict whether the load balancer (LB) will forward the packet to A or to B. In such circumstances, Traceroute could very well output (LB, A, E, F, D) as a path (the probe with $TTL = 2$ being sent to A and the one with $TTL = 3$

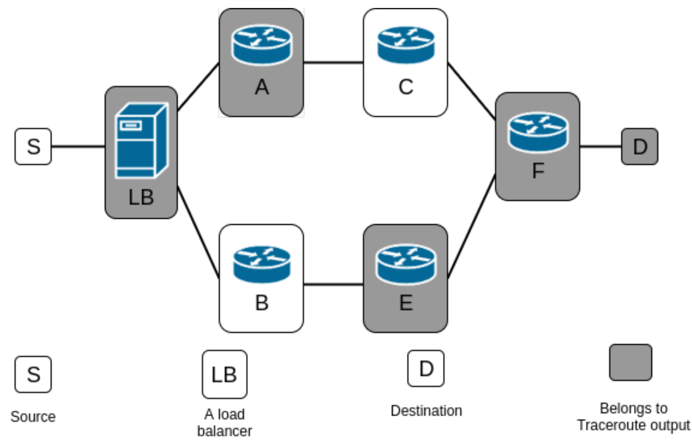

Figure 1.3: Traceroute in the presence of a load balancer.

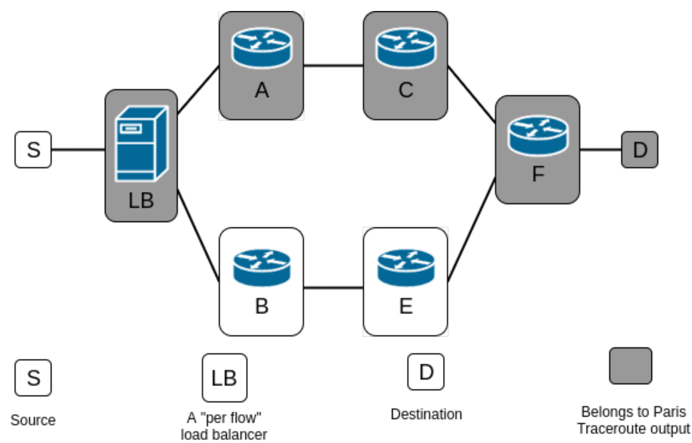

Figure 1.4: Paris Traceroute in the presence of a "per flow" load balancer.

to B). In order to solve that issue, Paris Traceroute manages the probe headers in such a way that all probes belonging to a same trace will be "load balanced" the same way in the presence of a "per flow" load balancer. Figure 1.4 gives the output of Paris Traceroute on the same topology as the one from figure 1.3 and supposes the load balancer to have a "per flow" policy. When a "per packet" load balancer is present, the packets are still subject to being load balanced differently. Indeed, the only way to ensure that every packet will be load balanced the same way is for them to have the same headers. However, it would then be impossible to match the responses to the probes. Therefore, Paris Traceroute does not ensure that packets will be treated the same way in the presence of a "per packet" load balancer. However, when it is the case, Paris Traceroute is able to notify the user so that he can treat the output accordingly.

1.3 Techniques survey

In this section, we are going to discuss some existing techniques for Internet-scale topology discovery and their efficiency.

1.3.1 SNMP, DNS and broadcast ping based techniques

Siamwalla *et al.* [10] introduced several algorithms to discover IP layer topology. The first one we are going to present is based on SNMP and is described in algorithm 1. SNMP (Simple Network Management Protocol) is a protocol designed to allow network administrators to remotely manage, monitor and diagnostic their networks. SNMP can be used to obtain information about link layer devices by querying their Management Information Base (MIB). This first algorithm takes as input a "temporary set" which consists of IP address(es) that may or may not correspond to actual host(s) and router(s). Then, for each element of the temporary set, it determines whether it is alive (via ping) and if so, obtains the list of hosts within its ARP table entries and the list of neighbouring routers from its ipRouteTable MIB entry. Address Resolution Protocol (ARP) [11] is a protocol that is used to map interface IP addresses to their Ethernet address. Although this algorithm is simple, efficient and accurate, it is based on the assumption that SNMP is available everywhere. This assumption is not verified in practice and therefore this algorithm can not be used when considering topologies at the scale of the Internet.

They also introduce a second algorithm based on DNS Transfer zone and broadcast ping. A broadcast ping is a ping packet which destination is an entire subnet. That way, every host in the subnet will receive that ping and will reply to the sender. The "transfer zone" command can be used to query a DNS server in order to retrieve the list of every name in a domain. That list can be useful to find all hosts and routers within the domain. The general idea behind this algorithm is to obtain the list of hosts and routers within the domain and then to check the validity of the list using a broadcast ping. Just like the first algorithm, this algorithm can not be used to compute an Internet-scale topology because not only is it slow but it also heavily relies on the DNS Transfer Zone and the broadcast ping which might both be unavailable for security reasons.

Algorithm 1 SNMP based topology discovery algorithm

$temporarySet \leftarrow getInputRouter()$
$permanentSet \leftarrow \emptyset$
for all $router \in temporarySet$ **do**
  $ping(router)$
  if $alive(router)$ **then**
    $permanentSet \leftarrow permanentSet \cup router$
    $permanentSet \leftarrow permanentSet \cup SNMP_GetHostsFromARPTable(router)$
    $routerList \leftarrow SNMP_GetNeighbouringRoutersFromIPRouteTable(router)$
    $permanentSet \leftarrow permanentSet \cup routerList$
    $temporarySet \leftarrow temporarySet \cup routerList$
  end if
end for

1.3.2 Traceroute-based techniques

In this subsection, we will present a few techniques that are quite similar to Traceroute but that try to improve upon it by reducing the number of probes sent while keeping the same quality of topology.

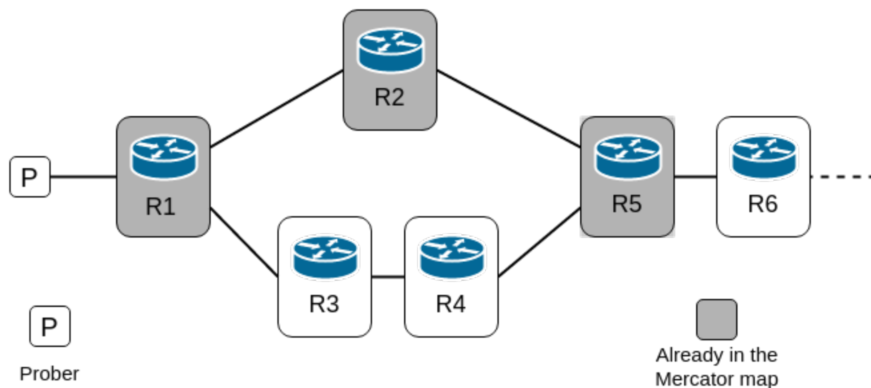

Figure 1.5: Example of Mercator trying to limit the overhead generated by the probes

Mercator

Mercator [12] is a tool used to infer an Internet map and that uses a mechanism quite similar to Traceroute (hop-limited probes). It also makes use of novel alias resolution techniques in order to identify interfaces belonging to a same router. In other words the resulting Internet topology will be at the router level. Mercator performs its measurements from a single point of the network only using hop-limited probes. It also does not require an input list of destinations to probe as it makes use of a so-called *informed random address probing* heuristic. This heuristic allows Mercator to probe different destinations in a more efficient way as it exploits both the results from previous probes and common IP address allocation policies. In order to reduce the overhead generated by the probes, Mercator uses two mechanisms:

- A probe is only sent if the previous one is done. In other words when a probe is sent, either the previous one has received its response or it has timed out (waiting for a response that might never arrive).

- Not every probe starts with a TTL of one. For each path, Mercator identifies the furthest router in that path such that it is already present in the generated topology. It then estimates the TTL to set the probe to such that this router will be the first one to respond. If the first received response is indeed coming from the identified router, it continues to probe that path. Otherwise it simply backtracks and starts over probing that path with a TTL of one. This is illustrated by figure 1.5. R1, R2, R5 are already part of the Mercator map, therefore, it identifies R5 as the furthest router in that path. It will therefore send the first probe with $TTL = 3$ such that if the probe takes the path (R1, R2, R5), then R5 will be the first one to respond. Upon reception of such a response, Mercator will carry on and send the next probe with $TTL = 4$ and thus discover R6. On the other hand, if the first probe had taken the path (R1, R3, R4, R5) then (as $TTL = 3$), R4 would have been the one to respond and therefore Mercator would have started from the beginning with $TTL = 1$.

RocketFuel

RocketFuel [13] is a tool that tries to discover the topology of a single ISP (generally at the center of the network). It relies on three main principles.

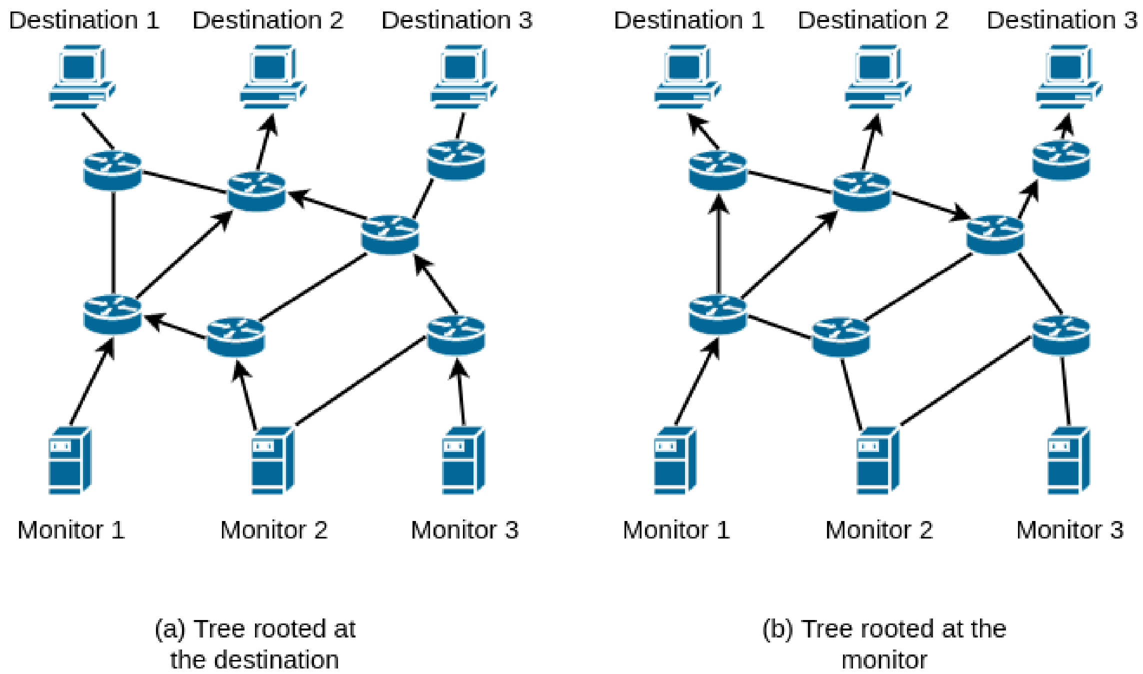

Figure 1.6: Illustration of the Internet tree-like structure that is assumed by DoubleTree.

- It performs **measurement selection** *i.e.* reduces the number of required probes by using two heuristics: *directed probing* and *path reduction*. The first one is used to only perform traceroute that go trough the ISP of interest while the second one limits the length of the paths to avoid intra and inter monitor redundancy as much as possible.

- It also relies on **alias resolution** *i.e.* identifies interfaces belonging to the same router.

- Finally, it uses **router identification techniques** *i.e.* determines which router belongs to the ISP of interest and retrieves information about the router's geographical location and role in the topology using some DNS and ISP naming conventions.

DoubleTree

DoubleTree [14] is a probing technique that takes advantage of the structure of the network based on the assumption that it is a tree-like structure. This is illustrated in figure 1.6. Indeed, it assumes that routes originating from multiple different monitors towards a single destination have a tree-like structure rooted at this very destination (*cf.* figure 1.6 (a)). It also assumes this tree-like structure for routes originating from a single monitor towards multiple different destinations (*cf.* figure 1.6 (b)).

In order to reduce the number of probes sent, DoubleTree will limit the length of the paths it will probe so that once it re-discovers an interface (or in other words "touches the tree"), it stops. In order to achieve that, it maintains information about the tree in two sets. The *local stop set* is, as its name indicates, a set that is local to each monitor and that contains all the interfaces that the monitor has discovered. The *global stop set* is global to all monitors and contains all the pairs (interface, destination).

High frequency probing strategies

In 2010, Robert Berverly *et al.* [15] presented three probing strategies that aims at giving primitives towards high-frequency active Internet measurements. To do so, they first quantified the unnecessary probing made by several existing large topology measurement platforms and developed three probing strategies that leverage prior knowledge about the network.

- **Subnet Centring Probing** is quite simple. Instead of randomly choosing which targets to probe in a subnet (which often leads to wasted probing or missing information), it ensures that the probing is done so that the targets probed in a same subnet are "as distant as possible" so that it gathers as much information as possible.

  For instance, to choose 4 probes to be sent to $192.168.0.0/16$, it will pick 4 addresses from the distinct prefixes $192.168.0.0/18$, $192.168.64.0/18$, $192.168.128.0/18$ and $192.168.192.0/18$.

  Then, as the probing progress, it uses the edit distance (according to the definition of Levenshtein) [16] in order to determine which prefixes are more worthy to be probed than others. In practice, that translates simply by an empirically defined threshold under which a prefix is not probed anymore as it is considered not to give any more useful information.

- **Interface Set Cover** is a probing strategy which goal is to leverage the previous probing cycles by relying on its interface-level graph. Indeed, it selects paths such that the packets that will be sent will probe interfaces that are not yet present in the graph.

  Obviously, such a strategy requires a "bootstrap" step during which the Subnet Centring Probing could be used.

- **Vantage Point Spreading** is the last probing strategy they presented. In it, the problem of dividing the probing among several vantage points (VPs) is taken as an opportunity to reduce the probing traffic.

  This strategy suggests to use as many distinct VPs as possible for the destinations of a given prefix. Each VP then applies the Subnet Centring Probing strategy and the potential new destinations that emerge will be assigned by Vantage Point Spreading to (if possible) VPs that have not yet been used to probe that particular prefix. Otherwise it simply distributes them uniformly among the set of available VPs.

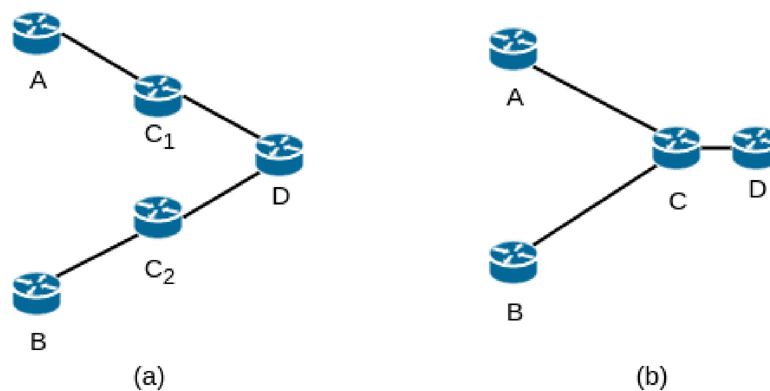

Figure 1.7: Example of IP alias resolution

1.3.3 What's next?

When using Traceroute (or one of its variants) in order to discover interfaces, we only reveal one interface of each router on the path. Therefore, when running several traceroute instances, the next step would be to identify interfaces belonging to a same router in order to obtain a router-level topology. This process is called *IP alias resolution* and is illustrated in figure 1.7. Indeed, we can see in the figure that interfaces C_1 and C_2 are aliases (*i.e.* they belong to the same router). The alias resolution process therefore gather them into a single node of the graph.

Chapter 2

YARRP

2.1 Problematic behind existing traceroute-based approaches

In chapter 1, we have reviewed traceroute and traceroute-based probing techniques. In this section, we are going to discuss the problematic behind those techniques and why we can not probe a substantial part of the IPv4 address space quickly (*i.e.* in terms of minutes instead of hours or days) using one of those.

First of all, traceroute-based approaches have to maintain a **state**. Indeed, the prober has to be able to establish a correspondence between any response received and the probe that triggered it. This is usually done by having the prober wait for the response after having sent a probe. Either the corresponding response arrives or a timeout threshold is reached and the target is considered unresponsive. This results in slow probing in which most of the time is spent waiting especially when the probing is done sequentially.

Another problem linked to traceroute-based approaches is the fact that, by design, the paths towards each target are probed **sequentially** (refer to figure 2.1). This implies a concentration of the load along the paths that might trigger rate limitation or IDS alarms. These consequences are obviously not desired and the only way to prevent them from happening is to limit the rate at which the prober operates. It then results again in a slow probing technique.

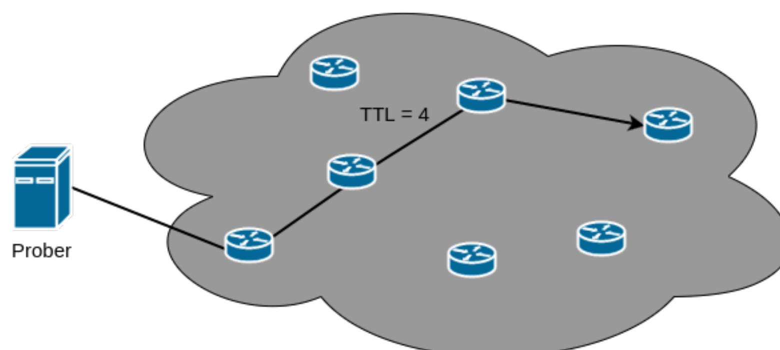

Figure 2.1: Example that shows the sequential probing of traceroute-based techniques with a probe of TTL = 4. It also shows the concentration of the load on the path from the prober to its target.

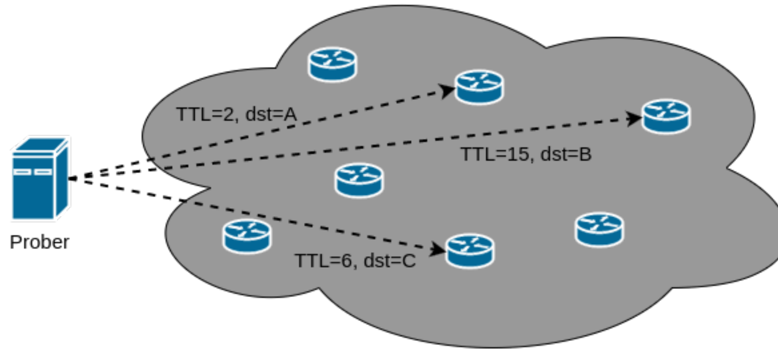

Figure 2.2: Example that shows the random probing order used by YARRP over the set of couples (IP, TTL).

2.2 YARRP design

Just like any traceroute-based approach, YARRP is a probing technique and stands for "*Yelling At Random Routers Progressively*". It is a very recent technique that was introduced by Robert Beverly in 2016 [2]. Its aim is to solve the slowness issue of other traceroute-based approaches by performing stateless operations and the concentration of the load issue by randomising the probing order.

In the next three subsections, we will describe the different steps that are taken in order to have such a fast probing technique. Throughout these sections we will consider a *reference problem*: probing each hop of the paths towards each $/16$ of the IPv4 address space. Even though we will use that problem to illustrate how YARRP works and give examples, the logic behind each of the following steps is still valid for bigger parts of the IPv4 address space *e.g.* all /24.

2.2.1 Preprocessing step

As its name indicate, YARRP will probe the different paths in a pseudo-random order. The aim is to uniformly randomise the probing order to avoid the load concentration issue that was discussed in section 2.2. For instance, a probe could be sent to IP address A with a TTL of 2, to address B with a TTL of 15 than to C with a TTL of 6 and so on until all the TTL for each address has been considered. Figure 2.2 illustrates the randomisation of the probing order and can be compared with the sequential probing shown in figure 2.1.

Let us consider the *reference problem i.e.* probing a path towards each /16. There remain three parameters to be set for this problem to be well defined:

- **The minimum TTL**: the lowest TTL from which we will probe. We will call it *TTLMin* and must be an integer strictly superior to zero.
- **The maximum TTL**: the highest TTL to which we will probe. We call it *TTLMax* and must be an integer strictly superior to *TTLMin*.
- **The address in each /16**: the methodology used to pick the target address in each /16 we will probe. For instance, each address could be picked at random.

When these three parameters are set, we can generate a list of couples (IP, TTL). There will be one element in this list per probe to be sent. Algorithm 2 illustrates how such a list is generated (for our *reference problem)* in which the *addressPicker* function corresponds to the third parameter we just discussed.

Algorithm 2 IPListGenerator

```
ls ← []
for all i ∈ {0, ..., 255} do
    for all j ∈ {0, ..., 255} do
        targetIP ← addressPicker(i, j)
        for all k ∈ {TTLMin, ..., TTLMax} do
            ls.add((targetIP, k))
        end for
    end for
end for
```

However, the generated list is not yet randomised. If we used it as such, it would be equivalent as a traceroute-based approach. In order to shuffle it, we will use the common *shuffle* Python function [17] that relies on the pseudo random number generator *Mersenne Twister* [18].

Each IP address in the list being actually represented as four *unsigned char* (*i.e.* 4 bytes) and each TTL as one *unsigned char* as well (1 byte), each couple weights at least 5 bytes. When considering our reference problem and setting *TTLMin* to 1 and *TTLMax* to 32 for instance, we have $2^{16} * 32 = 2,097,152$ couples which represents roughly $10.5MB$ in memory. While this still fits into memory, it will quickly become a problem *e.g.* if we had considered all /24, the list would already weight $2.7GB$. We simply can not feed the shuffle function with such a list. We therefore used a different approach.

Algorithm 3 inputs a list and outputs this exact same list shuffled based on the *Mersenne Twister* pseudo-random number generator. Its memory usage is $O(1)$ but it is $O(n)$ in terms of time (where n is the number of couples). Let N be the maximum number of couples that it is reasonable to have in memory at all times. The algorithm cuts the input file (*i.e.* the file containing all the couples generated by algorithm 2) into temporary files each containing at most N couples. Each temporary file is then shuffled independently. Then, while there exists a temporary file that has not reached EOF, it generates a random number between 0 and the number of remaining temporary file(s) in order to choose from which of these files the next couple to be written in the final output file will be. When done, the output file contains the same list as the input file but shuffled which is exactly what we needed.

Finally, the file mentioned above is converted into a binary file which format is described in figure 2.3. This binary file contains every couple (IP, TTL) and represents the output of the preprocessing step as well as the input of the probing step.

The preprocessing step as presented in this section can involve large files and substantial computing times *e.g.* for the shuffling. The key is that all these operations can be performed **offline** and **before** the probing campaign itself. Indeed, no matter how the binary file is generated, as far as it is compliant with the binary format described in figure 2.3, next steps will work just as well.

Algorithm 3 hugeShuffle

```
N ← "Number of couples that it is reasonable to put in memory"
inputFile ← openInputFile()
ls ← []
nbTempFiles ← 0
while ! inputFile reached EOF do
    ls ← "N next couples from file"
    shuffle(ls)
    writeInNewTempFile(ls, nbTempFiles)
    nbTempFiles ← nbTempFiles + 1
end while
close(inputFile)
outputFile ← openOutputFile()
ls ← "List of temporary files"
while ! all temporary files reached EOF do
    r ← MTRandom(0, size(ls) − 1)
    c ← getNextCouple(ls[r])
    outputFile.write(c)
end while
close(outputFile)
```

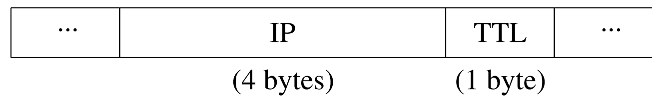

Figure 2.3: Format of the binary file that is the output of the preprocessing step.

2.2.2 Probing step

The probing step is the one in which the actual measurement will be performed. Let us first give a high-level view of the step.

The input of this step will be the binary file containing the (IP, TTL) couples described in the previous section. The goal will be to send one ICMP Echo Request per couple and receive the potential corresponding response. The probes will be sent to the IP and with the TTL of its corresponding couple. Each response received will be logged in a structured way into a file. This file will be the output of the probing step as well as the input of the postprocessing step.

Now that we have given an overview of the whole step, let us dive into it in more details. The probing step is actually a C program that inputs the binary files containing the couples and a packet rate. This program is composed of two processes that run in parallel: the sender and the receiver.

- **The sender** reads the binary file and loops on the couples it contains. It sends one ICMP Echo Request per couple, sleeping between each in order to comply with the packet rate chosen by the user.
- **The receiver** listens on a socket for responses and outputs them in a structured way to the output file every time it receives one.

As already stated in a previous section, we do not want the sender to wait for the corresponding response of a probe (or a timeout) before sending the next one. To do so, we need the sender and

0		8	16		24	31
Version	HL	Type of service	Total length			
Identification			Flags	Fragment offset		
TTL		Protocol	Header checksum			
Source IP address						
Destination IP address						
Options						

Figure 2.4: A regular IPv4 packet header

the receiver to be **stateless**. In other words, all the pieces of information that will be required by the postprocessing step (refer to section 2.2.3) should be present in the responses. In order to be able to reconstruct the topology in that step, we need three pieces of information:

- The source IP of the response
- The original destination to which the probe that triggered that response was destined
- The original TTL that was set in the probe that trigger the response

Let us now consider the main two types of response we are going to get and see if all three items listed above are contained in them. Many of the TTL we are going to set in the probes will not be high enough for the probes to reach their destinations. Therefore, the first type of response we are going to have is *ICMP Time Exceeded (type 11)* [19]. For this type of response, the two first items are already in it. Indeed, the source IP address of the response is obviously in the corresponding IP header field and the original IP address of the probe that triggered the response is contained in the quote of the ICMP packet. The original TTL of the probe however is not contained in the response. In order for it to be, we will overwrite the IPID IP header field with the original TTL in the probe. That way, when the response is received by the receiver, it will interpret the IPID field of the quote of the ICMP packet as the original TTL of the probe. Figures 2.4 and 2.5 illustrate the difference between the header of a classical IPv4 packet and the one of a YARRP probe.

The second main type of responses we are going to get is *ICMP Echo Reply*. We will get this kind of responses when the probe has been able to reach its destination and the destination is responsive to ICMP Echo Request packets. In such a case, it is obvious that the two first pieces of information (the source IP address of the response and the original destination) are the same and are contained in the response. As there is no quote in ICMP Echo reply of the corresponding ICMP Echo Request, there is no way to retrieve the original TTL from the response. When receiving such a response, we therefore only have the information that we have been able at some point to reach this particular destination.

Experiments have shown us that the receiver catches a negligible minority of other kinds of packets or even inconsistent packets. These packets are ignored by the receiver and therefore by the following postprocessing step as well.

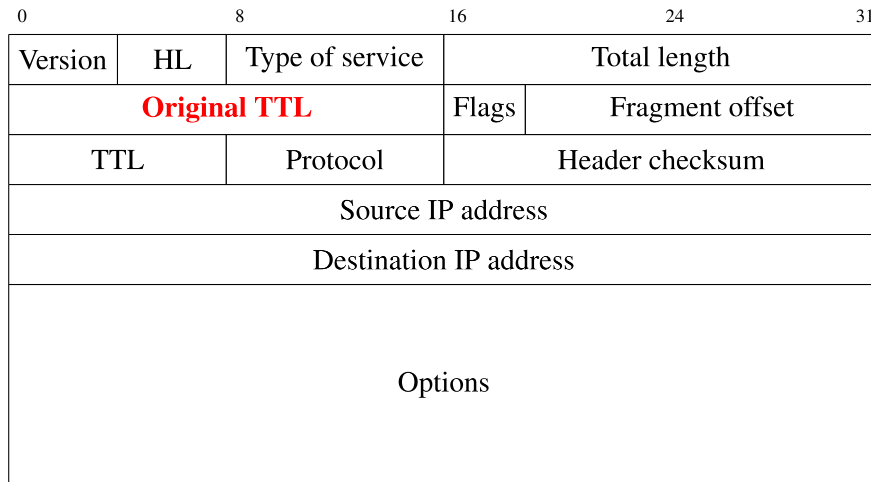

Figure 2.5: The IP header of a YARRP probe

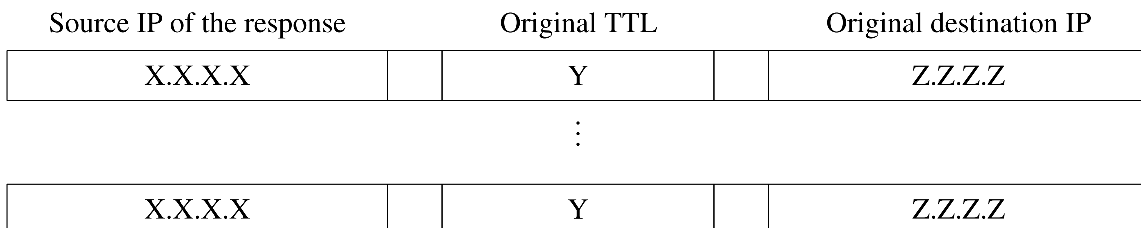

Figure 2.6: Format of the probe records in the ASCII version of the probing step output file.

Every time it receives a response, the receiver will output a "probe record" into an output file. This file will be the output of the probing step and the input of the next one *i.e.* the postprocessing step. It can either be in binary or in ASCII format depending on a program argument. For instance, in ASCII, there will be one probe record per line in the file and each of them has the format illustrated in figure 2.6. We clearly see that the three pieces of information we have discussed are in each probe record. When the response is an ICMP Echo Reply, the corresponding probe record will have its original TTL set to 0 and the original destination IP will be undefined. Therefore, when the original TTL is 0, we know that all we can rely on is the source IP it comes from.

2.2.3 Postprocessing step

The aim of this step is to compute the topology of the probed network based on the set of responses we had during the previous step. The output file of the probing step is therefore the input file of this one. The set of responses is composed of both ICMP Echo Reply (type 0) and ICMP Time Exceeded (type 11). The format of this input file has been defined in figure 2.6.

The output of this last step (which will therefore also be the general output of YARRP) will be a JSON formatted file representing the topology of the network that has been probed. The format of this file is described in listing 1. As we can see, it is basically a list of objects. Each object represents a path towards a target IP. It is composed of a "target" field that represents the IP address of the interface that has been probed, it has a "hops" field which is itself an ordered list of IP addresses representing the hops in

the path and lastly it has a "targetAnswered" field which is a boolean that tells us whether the target has answered with an ICMP Echo Reply or not. Lastly, it is worth mentioning that missing hops on a path will be represented by the IP address 0.0.0.0 in the "hops" list.

Listing 1 Example of JSON output file representing the topology

```
1  [
2      {
3          "target": "94.105.68.15",
4          "hops" : [
5              "165.47.82.3",
6              "54.23.98.74",
7              ...
8              "47.15.0.84"
9          ],
10         "targetAnswered" : true
11     },
12     ...
13 ]
```

For this original version of YARRP, the algorithm used to generate such a topology JSON file based on the set of responses is quite similar to a brute force approach. It is described in algorithm 4[1]. As it will often be impossible to load the entire set of responses into memory, the algorithm will iterate in the input file looking for not-yet-treated targets. Once it finds one, it will look again in the input file looking for responses that had this very same target as original destination IP. All these responses will correspond to the different "hops" on the path towards the target. Once it reaches the end of the file, it will output the path into the JSON file. Once there is no more targets to be treated, it closes all opened files and returns.

The algorithm presented in this section is a brute force approach of the topology reconstruction problem. Its complexity in time is $O(n^2)$ which gives quite long execution times when considering an input file with millions of responses. However its complexity in terms of memory is constant ($O(1)$). Even though the postprocessing step can be done entirely offline, we will see in section 3.3 that improvements can be found to make it faster.

2.3 Limitations

The original version of YARRP as it is presented in this chapter has three main limitations. First of all, if we consider our *reference problem*, YARRP will send probes to parts of the IPv4 address space that are very likely not to respond (for some we even know for a fact that they will not). For instance, here is a list of /8 address blocks in which the addresses are very likely not to respond to any ICMP Echo Request.

- **0.0.0.0/8** is reserved for self-identification [20]
- **10.0.0.0/8** is reserved for use in private networks [21]
- **127.0.0.0/8** is reserved for loopback addresses [22]

[1]Algorithm 4 disregards quite a lot of implementation details that makes it correct. This description of the algorithm is mainly given for the overview it provides to the reader.

Algorithm 4 Postprocessing algorithm

$inputFile \leftarrow$ "File pointer to input file"
$treatedTargets \leftarrow \emptyset$
while $inputFile \neq EOF$ **do**
  $P \leftarrow$ "Empty path"
  $r \leftarrow getNextResponse(inputFile)$
  if $r.target \notin treatedTargets$ **then**
    $P.target \leftarrow r.target$
    if $isICMPEchoReply(r)$ **then**
      $P.targetAnswered \leftarrow true$
    end if
    while $inputFile \neq EOF$ **do**
      $r\prime \leftarrow getNextResponse(inputFile)$
      if $r\prime.target == P.target$ **then**
        $P.addHop(r\prime.sourceIP)$
      end if
    end while
  end if
  $rewind(inputFile)$
end while

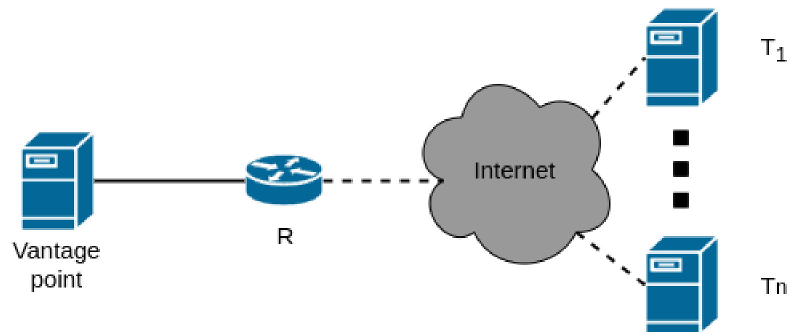

Figure 2.7: Illustration of intra-monitor redundancy.

- **224.0.0.0/8 – 239.0.0.0/8** are reserved for multicast [23]
- **240.0.0.0/8 – 255.0.0.0/8** are reserved for future use [24]

This issue is actually easy to solve as we can simply prevent the program that creates the list of couples from adding IPs from those /8 in the file. This will be discussed in more details in section 3.1.1.

Another limitation of the original YARRP is that it chooses the IP address it will probe in each /16 randomly. We therefore can not be sure that the interface corresponding to this IP address can be reached or whether it will respond to our probes. We could do better by only adding couples in the preprocessing step file with IP addresses from each /16 that responds to ICMP Echo Request. That way the probing step will only work with reachable and answering IP addresses. This will also be discussed in more details in section 3.1.1.

One last limitation we are going to discuss is the fact that the original YARRP is subject to a very high redundancy. This redundancy takes its source into two different issues.

- **Intra-monitor redundancy**

Because of the tree-like structure of the traceroute measurements, interfaces that are close to the vantage point will be re-discovered multiple times. This is illustrated by figure 2.7. Indeed, we can see on the figure that when probing targets T_1 until T_n, the router R will be rediscovered n times by YARRP. A solution to the issue is to find the optimal TTL window and it will be discussed in section 3.1.2.

- **Too large *TTLMax***
  When a target is too close to the vantage points (in terms of number of hops between them) compared to what the *TTLMax* has been set to, then we will probe this particular target multiple times. If it answers to ICMP Echo Request, we will receive multiple ICMP Echo Reply. As an example, let's suppose that a target T is 12 hops away from the vantage point and that we probe with a *TTLMax* of 32. If T answers to ICMP Echo Request, it is very likely that the receiver will receive 11 times the same ICMP Echo Reply from T.

  This is the second source of the redundancy issue and it can be partially solved by making the sender and the receiver processes of the probing step cooperate. This will be discussed in section 3.2.1.

Chapter 3

Improving YARRP

In this chapter, we are going to present several improvements over the original YARRP presented in chapter 2. Obviously, for each improvement, we will need to evaluate its impact. To do so, we will first define a few metrics and then we are going to give the values of these metrics for our *reference problem*. Finally, for each improvement that we are going to present, we will compute those metrics, compare them with the ones of the *reference problem* and discuss the advantages and the drawbacks.

It is worth mentioning that every measurement campaign of this chapter as well as the one from chapter 4 have been done from the exact same vantage point[1]. We have made these measurements independently, one after the other. Therefore the elapsed time between each is at most a few hours.

Here is the list of the five metrics we are going to use to evaluate the different improvements.

- **The number of probes sent** corresponds to the raw number of probes that is required to be sent during the probing step in order to obtain the desired topology. It is an indication of the level of "aggressiveness" of the probing technique. Indeed, the more probes sent, the more likely it is that our measurement campaign will be considered as an attack.

- **The response rate** corresponds to the number of received responses divided by the total number of probes sent during the probing step. It gives us a good indication of how responsive the probed network is and thus how well-chosen were the targets.

- **The uniqueness rate** corresponds to the number of unique IP addresses discovered in the responses divided by the total number of responses. It is a good indication of how redundant the probing method is. For instance, a uniqueness rate of 10% tells us that on average we rediscovered 10 times each interface.

- **The completeness rate** corresponds to the number of complete paths divided by the total number of discovered paths. A path is said to be complete if there is no missing hop from *TTLMin* until the last hop and if the target of the path answered with an ICMP Echo Reply. This metrics tells us how well the goal of the measurement campaign was achieved. Indeed, if the goal is to find a path towards every /16 and we have a completeness rate of 1, then not only did we find a path to each /16 but every hop of each of these paths answered to our probes. That would mean that the resulting topology will be quite accurate.

[1] In Strasbourg, France to be precise.

Metric	Value
Number of probes sent	$2^{16} \times 32 = 2,097,152$
Response rate	33.94%
Uniqueness rate	8.2%
Completeness rate	0.94%
Number of ASes	$3,878$

Table 3.1: Mean of the values obtained on five instance of measurement for our *reference problem*.

Although very interesting, this metric has to be used with caution. Indeed, if for a given path every hop from *TTLMin* but the very last one before the target answered, the path will be considered as complete anyway.

- **The number of ASes** corresponds to the number of distinct Autonomous Systems (AS) from which we have had a response during the probing step.

As a reminder, our *reference problem* consists in probing a path towards each /16 of the IPv4 address space by picking one IP address in each at random with $TTLMin = 1$ and $TTLMax = 32$. Table 3.1 summaries the mean of the values we obtained for five measurement campaigns of this *reference problem* from a same vantage point.

As we can see, YARRP without any improvements on our *reference problem* is quite aggressive: more than two millions probes are sent for a response rate of only 33%. It is also very redundant: only 8.2% of all the responses contained a unique IP address. Also, a vast majority of the paths are not complete. This is due to the fact that we ran this campaign with a *TTLMin* of 1 and that some routers at $TTL = 1$ and $TTL = 2$ were never responding to our ICMP probes. As most paths went through these same first hops, most of them are considered incomplete. In the next sections, we will discuss different improvements that will help to make these metrics better for our *reference problem*.

3.1 Improved preprocessing step

In this section, we will discuss two improvements that were made on the original preprocessing step. As a reminder, the preprocessing step is the one during which we will generate a randomised list of couples (IP, TTL) that the probing step will input. With the two improvements we will present, we are going to address two of the main issues that were identified in section 2.3: the **intra-monitor redundancy** and the **too-large *TTLMax*** issue.

3.1.1 Filtering input IP addresses

The original version of YARRP generated the list of couples (IP, TTL) simply by picking an IP address at random in every /16. There are two issues related to that methodology.

- A non-negligible part of the addresses that are picked will never answer to any ICMP Echo Request. This is due to the fact that some parts of the IPv4 address space are reserved for special uses *e.g.* 10.0.0.0/8 is reserved for private networks [21]. These addresses are not routable on the Internet and it is therefore next to impossible to receive a response. A list of these reserved subnets was already given in section 2.3.

- Even when considering a routable /16, picking an address at random in it will likely give us an unresponsive interface. That will result in paths that are not complete in our topology which we obviously would like to avoid.

In order to solve these two issues, we will **filter the targets IP addresses**. Indeed, instead of having one address for each /16 picked at random, we will have one responsive address[2] for as much as possible /16 omitting the ones that are not routable. The algorithm that was implemented to build such a list is presented in algorithm 5 and 6 (respectively for the sender process and the receiver process).

Algorithm 5 AnsweringIPListGenerator - Sender process

```
state ← "shared set with the receiver process"
while !state.hasOneAnsweringIPPerSubnet() do
    for all subnets S do
        if isRoutable(S) & S ∉ state then
            target ← pickRandomIP(S)
            sendICMPEchoRequest(target)
        end if
    end for
end while
```

Algorithm 6 AnsweringIPListGenerator - Receiver process

```
state ← "shared set with the sender process"
while !state.hasOneAnsweringIPPerSubnet() do
    waitForICMPEchoReply()
    response ← getICMPEchoReply()
    if isValid(response) then
        writeInOutputFile(response.sourceIP)
        state.add(response.subnet)
    end if
end while
```

As we can see, the sender and the receiver process share a set-like data structure that is called "state". This set will contain the subnets for which we already have an answering interface. It is therefore initialised to the empty set.

The role of the sender process is to send an ICMP Echo Request to randomly picked IP addresses in each subnet that is not yet present in the state. It will continue to loop until one address per subnet has been found. The role of the receiver process is to receive the ICMP Echo Replies that have been triggered by the sender process. While receiving those packets, it will produce the list of answering IP addresses and inform the sender process through the shared state set (so that the sender will not send unnecessary probes).

Now that we have discussed a new methodology to generate a list of couples (IP, TTL) containing only interfaces that respond to ICMP Echo Request, it is time to evaluate the impact of this improvement. To be able to compare the results with our *reference problem*, we tried to generate such a list with one responsive IP address per /16. It is worth noting that there are 2^{16} /16 subnets in the IPv4 address space. The goal was therefore to find 2^{16} responsive IP addresses. In practice and after a substantial time of probing, we were only able to find around 53% of these responsive IP addresses.

[2] An interface is said to be responsive if it sends an ICMP Echo Reply upon reception of an ICMP Echo Request.

Metric	Value
Number of probes sent	$1,105,150$
Response rate	81.59%
Uniqueness rate	11.12%
Completeness rate	0.96%
Number of ASes	$5,949$

Table 3.2: Metrics obtained for our *reference problem* improved only by filtering the input IP addresses.

Even though we did not find one responsive address per /16 subnet, we will see that such an IP filtering has a great effect of the metrics we defined in section 3. Indeed, after having generated the list of couples (IP, TTL) with only responding IP addresses and with $TTLMin = 1$ and $TTLMax = 32$, after having done the probing step from the same vantage point as our *reference problem* and after having run the different postprocessing scripts on the output of the probing step, we obtained the results presented in table 3.2.

As we can see the total number of probes sent is almost divided by two. This is of course due to the fact that we were able to find only 53% of the 2^{16} IP addresses we wanted. However, the response rate is way better than it was with the original version of YARRP. This is obviously due to the fact that all interfaces we probed responded to the ICMP Echo Requests we sent them and are thus reachable. The completeness rate on the other hand remains quite low. Indeed, we performed the probing from the same vantage point as the one of the *reference problem.* Therefore the two first unresponsive hops are still present and prevent a lot of paths from being complete. If we consider the paths only from the second hops, more than 12% of them are complete. This will be discussed in more details in the next section.

Filtering the input IP addresses is therefore very efficient when it comes to improving the response rate but has the drawback of not being able to probe a path to every /16 as it is not possible to find a responsive address for each /16 in a reasonable amount of time. It also has next to no impact on the completeness and the uniqueness rate.

3.1.2 TTL window restriction

As already explained in section 2.3, YARRP suffers from very high redundancy due to two main factors: *intra-monitor redundancy* (*i.e.* rediscovering interfaces that are close to the vantage point multiple times) and what we have called *too large TTLMax* (*i.e.* sending unnecessary probes and receiving multiple ICMP Echo Reply from a same target).

Both of these issues can be resolved quite simply by respectively increasing *TTLMin* and decreasing *TTLMax*. Obviously this represents a trade-off as increasing *TTLMin* will make YARRP miss some of the interfaces that are close to the vantage point and decreasing *TTLMax* will make it miss the end of the longest paths of the topology. However, we will show that the reduction of the level of redundancy will be worth the trade-off.

We thus have to choose new values for *TTLMin* and *TTLMax*. Ideally, these values should be such that YARRP will still be able to catch most of the internet paths and such that the level of redundancy decreases significantly. With that in mind, we will set these values to respectively 12 and 27. This choice of parameters is motivated in two ways.

- First, figure 3.1 from the CAIDA website [25] represents the complementary cumulative distribution

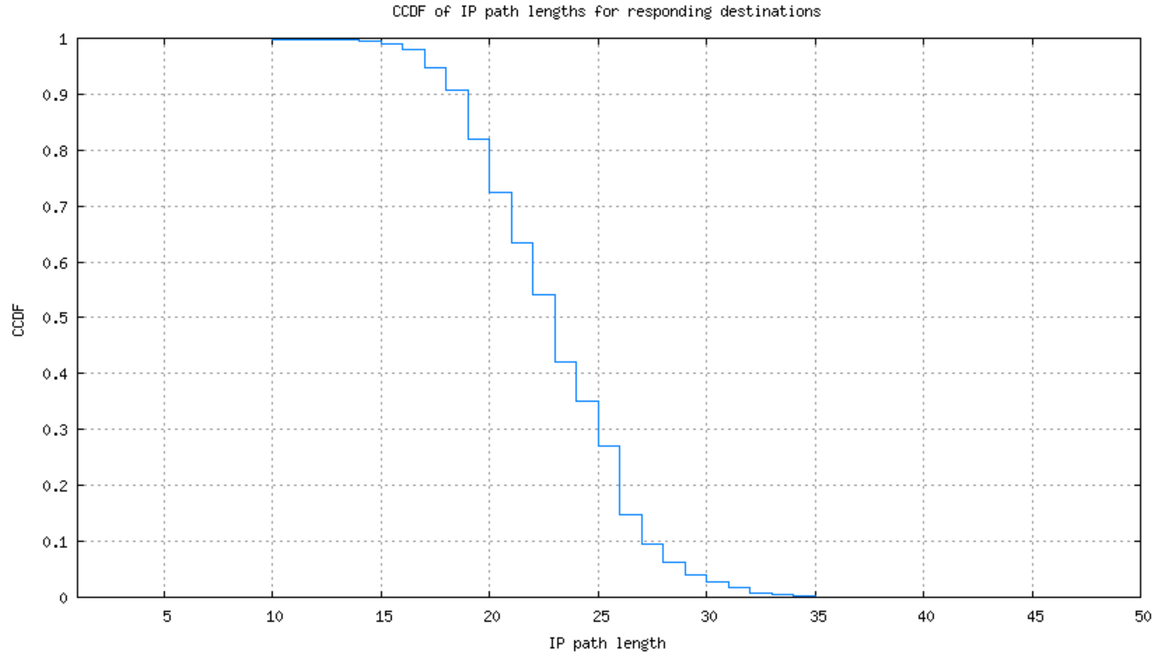

Figure 3.1: Complementary cumulative distribution function (CCDF) of path lengths (in number of hops) to a destination that responds to probing (taken from the CAIDA website [25]).

function of responsive IP path lengths. We can see that if we consider the *TTLMax* of 27 we just chose, we will be able to reach more than 90% of the responsive IPs. Our choice for the *TTLMax* is therefore relevant.

- Then, figure 3.2 represents the uniqueness rate as a function of the *TTLMin* and the *TTLMax*. Each point in this figure represents the mean of the uniqueness rates of five measures with the corresponding *TTLMin* and *TTLMax*. We can see that when it comes to reducing the redundancy (and thus increasing the uniqueness rate), our choice of parameters is relevant as the zone of the graph from $TTLMin = 12$ and $TTLMax = 27$ tends to stay away from the dark side.

Let us now evaluate the impact of this improvement. We performed a measurement similar to our *reference problem* but in which the TTL window has been set with $TTLMin = 12$ and $TTLMax = 27$. Table 3.3 illustrates the results. As we can see, the number of probes sent is divided by two while the uniqueness rate increases from 8.2% to 21.5% and the completeness rate goes from 0.94% to 11.89%. In other words, a little bit more than one response over five received gave us a not-yet-discovered IP address while the number of probes sent is half less. However this improvement has a cost: we do not recover the topology from the closest and the furthest hops to the vantage point and the response rate decreased a little going from around 33% to 22%.

3.2 Improved probing step

3.2.1 Communication between the sender and the receiver

Let us remind that the probing step is composed of a sender which job is to send the probes to the different targets and a receiver which job is to receive the corresponding responses and output them in a file. In

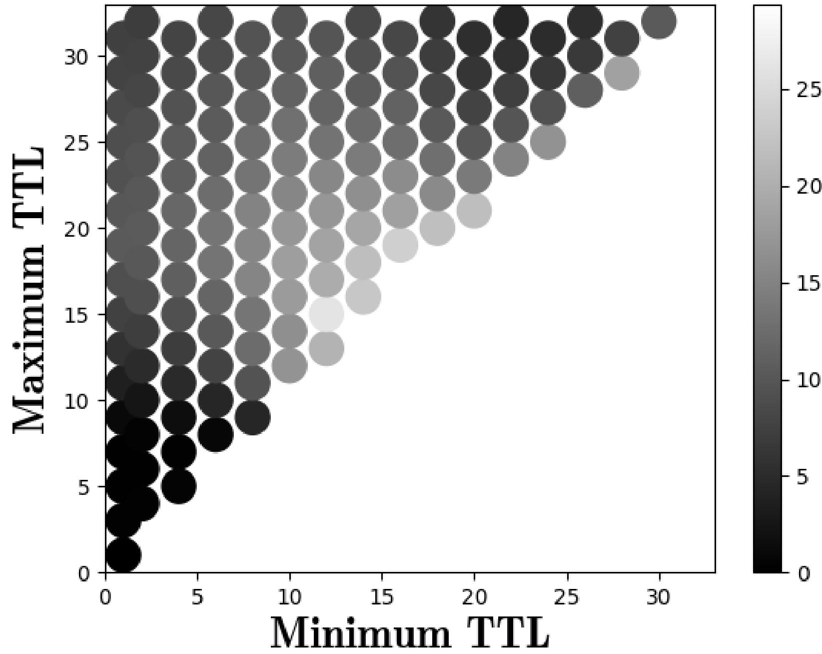

Figure 3.2: Scatter plot of the uniqueness rate as a function of the *TTLMin* and the *TTLMax* on our *reference problem*.

Metric	Value
Number of probes sent	$1,048,576$
Response rate	22.31%
Uniqueness rate	21.5%
Completeness rate	11.89%
Number of ASes	$3,486$

Table 3.3: Metrics obtained for our *reference problem* improved only by restricting the TTL window to $TTLMin = 12$ and $TTLMax = 27$.

practice, both of these are implemented in separate processes.

Let us now consider the following scenario. As the sender sends the probes, one of them triggers an ICMP Echo Reply from target T that is received by the receiver. Let us note T_{TTL} the TTL with which the probe that triggered this response. As we have received an ICMP Echo Reply from T, it means that the probe with TTL T_{TTL} was able to reach it. Therefore, the sender could skip any future probes to target T for which the TTL is higher than T_{TTL}. Indeed, not only will it be an unnecessary probe but it will also trigger the exact same ICMP Echo Reply and thus increase the redundancy.

In order for this to work, the sender needs to maintain a little bit of state. Indeed, it should remember what was the highest TTL of the probes sent to any target. That way, when the receiver notifies the sender[3], it knows the minimum TTL from which it can skip probes.

Let us now evaluate the impact of this improvement. Table 3.4 summaries the result of a measurement similar to our *reference problem* but for which we have "turned on" the communication between the sender and the receiver.

[3]In practice, this Inter-Process Communication (IPC) is implemented using message queues.

Metric	Value
Number of probes sent	$2,048,365$
Response rate	32.02%
Uniqueness rate	9.13%
Completeness rate	0.7%
Number of ASes	$3,709$

Table 3.4: Metrics obtained for our *reference problem* improved only by enabling the communication between the sender and the receiver.

The communication between the sender and the receiver allowed to avoid sending a little less than 50,000 probes. Is it clear that the impact of such improvement is quite limited: the response rate remained quite stable while the uniqueness rate increased of roughly 1%. However, there is absolutely no reason not to do it. Indeed, this improvement **is not a trade-off** as the probes that we skip are guaranteed to be unnecessary as they will not bring any more new information. It is also interesting to note that as expected, the completeness rate remains stable.

3.3 Improved postprocessing step

The aim of the postprocessing step is to rebuild the topology based on the set of responses obtained by the probing step. Even though the algorithm presented in section 2.2.3 allowed to do it correctly, it is quite slow as its time complexity is $O(n^2)$. In this section, we will present a way of making this step faster by **sorting beforehand**.

3.3.1 Sorting beforehand

The problem with the algorithm presented in section 2.2.3 is that is contains a double loop. The first one loops on each response to identify the list of targets. The second one loops on each responses for each target to find the packets belonging to its trace. Therefore, its time complexity is $O(n^2)$.

However, if we sorted the list of responses beforehand according to the target IP, all the responses concerning a particular target would be contiguous in the list. As the maximum number of responses per target is 32, the second loop would execute at most 32 times and therefore would make the time complexity of the whole process $O(n)$.

Obviously, sorting the responses beforehand only makes sense if the sorting algorithm itself has a time complexity that is inferior than $O(n^2)$. Some classic sorting algorithm (like heap-sort) can already reach a time complexity of $O(nlog(n))$ but we will now consider a different approach that will be able to sort the responses in a time complexity of $O(n)$.

Let us consider our *reference problem*. The algorithm works as follows:

1. Allocate a bi-dimensional array where each dimension goes from 0 up to 255. Each element of this array will serve as a "bin" for the responses a /16 *e.g.* responses from targets with $IP \in 123.85.0.0/16$ will go to the bin of coordinate [123][85][4].

[4] If we where considering the problem in which we want to probe a path towards every /24, we would have taken a 3-dimensional array with the same dimensions.

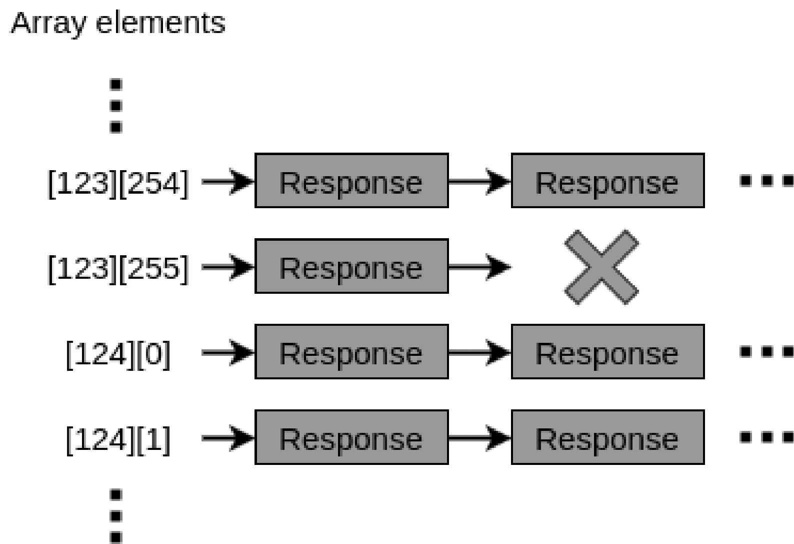

Figure 3.3: Illustration of the data structure that will help with the sorting of the list of responses.

2. Loop on the list of responses and put each of them in its corresponding bin. The bins could be implemented as linked lists in which the responses are inserted at the beginning of the lists. Figure 3.3 illustrates the data structure we described.

3. Once every response has been treated, go through the array and output in in a file.

While this algorithm has a time complexity of $O(n)$, it consumes a non-negligible amount of memory. Indeed, if we still consider our *reference problem*, and if each "bin" of the array is a pointer of 8 bytes to its linked list, then the empty array itself is already $256^2 \times 8$ bytes. If we consider the worst case scenario in which the response rate is 100%, then each linked list contains 32 elements of 17 bytes[5]. Therefore, the total weight of this data structure would already be $(256^2 \times 8) + (256^2 \times 32 \times 17) = 36.18MB$. While this might not represent a large amount of memory, it quickly increases with the size of the problem. Thus, if we make the same calculations for the problem in which we probe a path towards every /24, we obtain $(256^3 \times 8) + (256^3 \times 32 \times 17) = 9.26GB$. However, as the postprocessing step can be done offline and on any machine, nothing prevents us from using machines like the NIC4 SEGI cluster from the University of Liège which has $64GB$ of RAM per node [26].

[5]A response is composed of 2 IPs of 4 bytes each, a TTL of 1 byte and a pointer to the next element of 8 bytes.

Chapter 4

Optimised YARRP

In this short chapter, we will run a complete YARRP measurement campaign that takes full advantage of the improvements we have discussed in chapter 3. In the first part of this chapter we will describe the problem, what we are trying to achieve as well as the preprocessing, probing and postprocessing steps. Then, in the second part of the chapter, we will present the detailed results of the measurement campaign and try to interpret them.

4.1 Large-Scale Measurement Campaign

In our *reference problem*, we wanted to find a path towards as much /16 as possible. For this measurement campaign, we will do the exact same thing but instead of considering /16's, we will consider /24's. The choice of this new problem was not made randomly. Indeed, there already exists several large Internet-scale topology discovery experiments like Archipelago [27] or iPlane [28]. Large projects like these often make assumptions regarding the Internet structure by probing only one randomly-chosen IP address per sufficiently small subnetwork [15].

As the title of this subsection indicates, this is a bigger measurement campaign. Indeed, it increases the size of the problem by a factor of 256 (there are 2^{24} /24 in the IPv4 space while only $2^{16}/16$). We will now describe each of the three steps that were performed for the measurement.

- For the **preprocessing step**, no modification of the scripts and programs were required except for the one that generates the list of responding IP addresses (refer to section 3.1.1 for more details). Indeed, instead of randomly selecting one IP address to probe by /16, it now picks one per /24 and probe it to see if it answers. As before, if it does, the receiving process will output it in the list. The list of responding IP addresses was then extended as the corresponding list of couples (IP, TTL) with $TTLMin = 12$ and $TTLMax = 27$ as recommended by section 3.1.2. Finally, the list of couples was transformed into binary format (see figure 2.3) so that it can be fed to the probing step.

- For the **probing step**, almost no modification was required. Indeed, the sender is a program that inputs a list of couples (IP, TTL) and sends the corresponding probes. It does not have much "intelligence" as it absolutely pays no attention to the characteristics of the probes it sends. This logic is also valid for the receiving program as it pays no attention whatsoever to the characteristics

	0	1	2	3	4
0	0	0	0	0	0
1	0	5	0	27	0
2	0	0	23	0	0
3	15	8	0	2	0

Figure 4.1: Illustration of state of the sender. For instance, we can see that the highest TTL of the probes that were sent to the IP address picked for 1.3.0.0/16 is 27.

of the response it gets (as far as they correspond to a probe that was sent by the sender). For more details about the sender and the receiver program, refer to section 2.2.2.

The communication between the two processes however had to be adapted. Indeed, as explained in section 3.2.1, the sender needs to maintain a state so that it remembers the highest TTL of the probe(s) sent to each target at any time. In order to save memory, this state was originally implemented by taking advantage of the fact that we picked only one IP per /16. A two-dimensional array of 256×256 was allocated and zeroed. This array was the state of the sender as each element $[x][y]$ represented the highest TTL at which probes had been sent to $x.y.0.0/16$. This is illustrated in figure 4.1. For instance, we see in this figure that the highest TTL of the probes sent to the IP address picked for 1.3.0.0/16 is 27. Therefore, if later a couple (IP, TTL) is read by the sender such that $IP \in$ 1.3.0.0/16 and $TTL >$ 27, then this probe will not be sent as it will not provide any additional information.

That mechanism was easily adapted for our new problem by keeping the exact same mechanism on a 3-dimensional array. Figure 4.2 illustrates that array. For instance, we see in this figure that the highest TTL of the probes sent to the IP address picked for 113.1.2.0/24 is 19. Therefore, if later a couple (IP, TTL) is read by the sender such that $IP \in 113.1.2.0/24$ and $TTL > 19$, then this probe will not be sent.

- The **postprocessing step** remained exactly the same. Indeed, the number of probes sent, the response rate, the uniqueness rate and the completeness rate can be computed without considering the probing logic that was used in the first place. The aim of this step remains the same: rebuilding the topology based of the set of responses and then compute the different metrics. For more details, refer to section 2.2.3.

4.2 Results

In this section, we will present the results of the measurement campaign. We will proceed step by step to see the different metrics along with the amount of time spent on each of them.

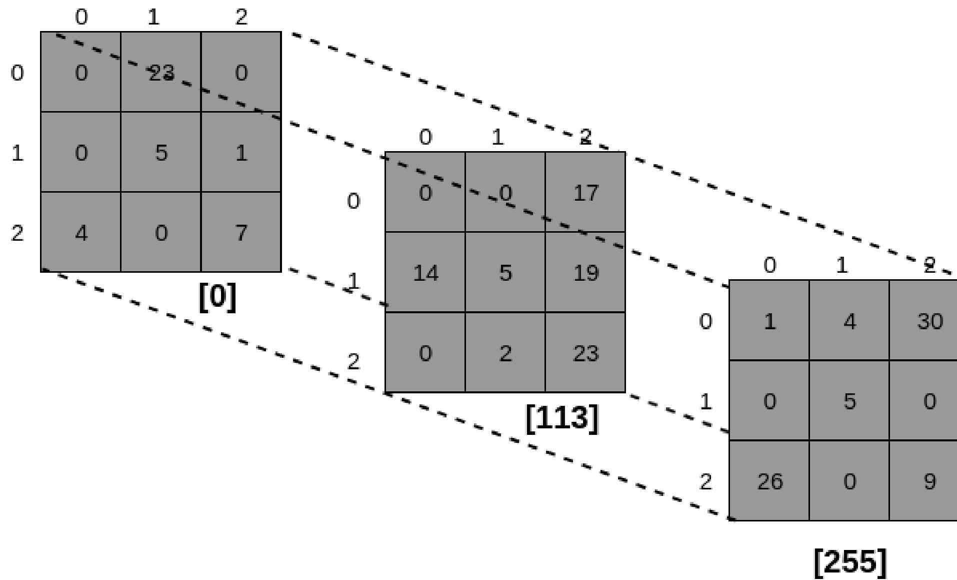

Figure 4.2: Illustration of state of the sender for the /24 problem. For instance, we can see that the highest TTL of the probes that were sent to the IP address picked for 113.1.2.0/24 is 19.

Preprocessing step

As discussed in the previous section, the first step is to generate the list of responding IP addresses. The sender process was set to send 1000 probes per second and ran for two days without interruption. This packet rate is quite low compared to what could have been chosen. However we really wanted to avoid triggering any alarms for our measurements. In the two-days time frame, it was able to gather roughly 3.36 millions different responding IP addresses from different /24. That number might not seem very high as it represents only 20% of the 2^{24} theoretically possible addresses but there are two things to take into account here:

- First of all, plenty of $/8$ of the IPv4 address space are reserved for special purposes (refer to section 2.3 for a list). Moreover, even though not reserved for special purposes like the ones in the list, some other $/8$ are allocated to different institutions. For instance, no less than 13 different $/8$ are allocated to the *United States Department of Defense* [29]. In practice, we have observed that these $/8$ along with some others are not very talkative. Therefore, finding responsive IP addresses within these subnets might not always be possible.

- After two days, the program was still running and the number of responding IP addresses was still increasing. We stopped it because the 3.36 millions was considered as a reasonable amount of responding IP addresses in order to demonstrate the optimised version of YARRP. If we had had the time to run it for a longer period, we certainly would have covered a larger part of the IPv4 address space. It is worth noting that our program that gathers huge lists of responding IP addresses could be subject to some optimisations as well (*e.g.* parallelisation, focusing more on some part of the IPv4 address space than others, ...).

The list of 3.36 millions IP addresses generated, we transformed it into another list of 53.76 millions couples (IP, TTL). Indeed, each target IP will be probed between $TTLMin = 12$ and $TTLMax = 27$ which gives us 16 probes per IP. Thus, $16 \times 3.36 \times 10^6 = 53.76 \times 10^6$ probes in total.

Metric	Value
Number of probes sent	$47,349,496$
Response rate	80.3%
Uniqueness rate	15.1%
Completeness rate	65.3%
Number of ASes	$35,415$

Table 4.1: Metrics obtained when trying to probe the paths towards a list of 3.36 millions responding IP addresses from different /24 using all the improvements discussed in chapter 3.

Probing step

Given the list of roughly 53 millions potential probes to be sent, the entire probing step took approximately half a day to be completed. As the packet rate was fixed (1000 probes per second), we initially expected to spend $53.76 \times 10^6/1000/60/60 \approx 15h$ in the probing step. In practice, we spent two hours less thanks to the communication between the sender and the receiver improvement.

Indeed, the sender/receiver communication improvement allowed us to not send $6.411.746$ useless probes. That corresponds to 12% of the probes in the list. If we compare this result with the one made on our *reference problem* (see section 3.2.1), then we can emit the hypothesis that the impact of the sender/receiver communication improvement grows with the size of the measurement campaign.

In the end, a little bit more than 38 million probes responses were received which corresponds to a response rate of 83.3%. While not being the highest response rate we have had during our measurements, this remains acceptable.

Postprocessing step

This step was the longest one: the postprocessing program ran for roughly a week without interruption rebuilding the topology. Indeed, our lack of access to machines with large amount of RAM prevented us from using the improved version of the postprocessing algorithm presented in section 3.3. We therefore stuck with its slow original version.

Nevertheless, we were able to compute every single one of our metrics on this topology which are summarised in table 4.1. While we have a quite high response rate, the uniqueness rate is a little bit low. However it still is roughly twice the uniqueness rate obtained with the original version of YARRP. The completeness rate on the other hand is the highest one we have obtained in this work.

Conclusion

As we have seen in chapter 1, numerous different probing methods exist if one wants to obtain an Internet-wide topology. However, to this day, even the fastest methods will require a substantial probing time during which the underlying topology might change.

In this master thesis, we have presented, re-implemented and improved a very recent stateless probing method: YARRP. Throughout our improvements, we have managed to make it both more efficient and less aggressive (from the perspective of the probed network administrator).

That said, it is always possible to go further. For instance, the current version of YARRP only supports IPv4 probes. Adding support for IPv6 probing would enable us to gather information about the IPv6 topology. However, we would need to find another IP header field than the IPID to encode the original TTL of the probe in order for YARRP to keep its stateless property. Indeed, this field is not present anymore in the IPv6 header.

Another future improvement would be related to security. Indeed, in the current version of YARRP, nothing prevents a malicious user to mess with the measurements by forging fake replies to the probes and sending them to the receiver process. A shared key between the sender and the receiver process along with a hash function could be useful in the future in order to ensure the authenticity and integrity of the probes. As the communication between the sender and the receiver process has already been established in section 3.2.1, sharing a key should not be a problem: the sender generates a pseudo-random key and send it using IPC to the receiver process. Then, before sending any probe, the sender computes $hash(K|H)$ (where K is the shared key and H are the probe headers) and put it in a field of the packet headers. Upon reception of the quote of the probe, the receiver will be able to check both the authenticity of the packet (*i.e.* check that it is indeed a response to a probe sent by the sender and nobody else) and the integrity (*i.e.* check that the response was not modified in-flight). Once again, the challenge for this improvement will be to find a suitable header field to put the result of the hash function in.

"*The Internet is the first thing that humanity has built that humanity doesn't understand, the largest experiment in anarchy that we have ever had.*"

— Eric Schmidt

Hopefully, with tools like YARRP and its future versions, we will be able to understand it a little bit better.

Bibliography

[1] D. Clark. "The Design Philosophy of the DARPA Internet Protocols". *AMC SIGCOMM Computer Communication Review (CCR)*, 18(4):106–114, August 1988.

[2] Robert Beverly. "Yarrp'ing the Internet: Randomized High-Speed Active Topology Discovery". In *Proc. ACM Internet Measurement Conference (IMC)*, November 2016.

[3] B. Donnet and T. Friedman. "Internet Topology Discovery: a Survey". *IEEE Communications Surveys & Tutorials*, 9(4):56–69, April 2007.

[4] Microsoft Corporation. "Link Layer Topology Discovery protocol(LLTD)". `https://msdn.microsoft.com/fr-fr/library/windows/desktop/dn594471.aspx`.

[5] K. Keys. "Internet-Scale IP Alias Resolution Techniques". *ACM SIGCOMM Computer Communication Review (CCR)*, 40(1):50–55, Jan 2010.

[6] L. Bilge, S. Sen, D. Balzarotti, E. Kirda, and C. Kruegel. "Exposure: A Passive DNS Analysis Service to Detect and Report Malicious Domains". *ACM Transaction on Information and System Security*, 16(4):14:1–14:28, April 2014.

[7] die.net. "Traceroute Linux manual page". `https://linux.die.net/man/8/traceroute`.

[8] CISCO. "Understanding the Ping and Traceroute Commands". `http://www.cisco.com/c/en/us/support/docs/ios-nx-os-software/ios-software-releases-121-mainline/12778-ping-traceroute.html#traceroute`.

[9] B. Augustin, X. Cuvellier, B. Orgogozo, F. Viger, T. Friedman, M. Latapy, C. Magnien, and R. Teixeira. "Avoiding Traceroute Anomalies with Paris Traceroute". In *Proc. ACM Internet Measurement Conference (IMC)*, October 2006.

[10] R. Siamwalla, R. Sharma, and S. Keshav. "Discovering Internet Topology". Technical report, Cornell Network Research Group, Department of Computer Science, Cornell University, Ithaca, NY 14853, July 1998.

[11] D. C. Plummer. "Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware". RFC 826, November 1982.

[12] R. Govindan and H. Tangmunarunkit. "Heuristics for Internet map discovery". In *Proc. IEEE INFOCOM*, March 2000.

[13] N. Spring, R. Mahajan, and D. Wetherall. "Measuring ISP Topologies with Rocketfuel". *AMC SIGCOMM Computer Communication Review (CCR)*, 32(4):133–145, August 2002.

[14] B. Donnet, P. Raoult, T. Friedman, and M. Crovella. "Deployment of an Algorithm for Large-Scale Topology Discovery". *IEEE Journal on Selected Areas in Communications*, 24(12):2210–2220, Dec 2006.

[15] R. Beverly, A. Berger, and G. Xie. "Primitives for Active Internet Topology Mapping: Toward High-frequency Characterization". In *Proc. ACM Internet Measurement Conference (IMC)*, November 2010.

[16] National Institute of Standards and Technology. "Levenshtein distance". `https://xlinux.nist.gov/dads/HTML/Levenshtein.html`.

[17] Python Software Foundation. "Python manual page of the shuffle function". `https://docs.python.org/2/library/random.html#random.shuffle`.

[18] M. Matsumoto and T. Nishimura. "Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator". *ACM Transaction on Information and System Security*, 8(1):3–30, January 1998.

[19] J. Postel. "Internet Control Message Protocol". RFC 792, September 1981.

[20] D. Karrenberg, Y. Rekhter, E. Lear, and G. Jan de Groot. "Address Allocation for Private Internets". RFC 1918, February 1996.

[21] M. Cotton and L. Vegoda. "Special Use IPv4 Addresses". RFC 5735, January 2010.

[22] M. Cotton, L. Vegoda, B. Haberman, and R. Bonica. "Special-Purpose IP Address Registries". RFC 6890, April 2013.

[23] M. Cotton and L. Vegoda. "IANA Guidelines for IPv4 Multicast Address Assignments". RFC 5771, March 2010.

[24] S. Deering. "Host extensions for IP multicasting". RFC 1112, August 1989.

[25] University of Melbourne and CAIDA. "CCDF of IP paths for responding destinations". `https://www.caida.org/projects/ark/statistics/mel-au/resp_path_length_ccdf.html`.

[26] CÉCI. "CÉCI Clusers specification website". `http://www.ceci-hpc.be/clusters.html`.

[27] CAIDA. "Archipelago (Ark) Measurement Infrastructure". `http://www.caida.org/projects/ark/`.

[28] H. V. Madhyastha. *"An Information Plane for Internet Applications"*. PhD thesis, University of Washington, 2008.

[29] IANA. "IPv4 Address Space Registry". `https://www.iana.org/assignments/ipv4-address-space/ipv4-address-space.xml`.