Client infrastructure design and implementation of a client-server Internet topology mapping system for smartphones

Federico Asara

Abstract

Many studies have already acknowledged the importance and usefulness of an accurate Internet network topology map. Current efforts to build this map cannot achieve high coverage and quality levels, though. A critical factor is the monitor implementation. A monitor is a network node that executes analyses. In all current efforts they are in low numbers, ill-located and static, i.e. they do not change location over time.

In this thesis, I propose the mYriadi solution to the Internet mapping problem, focusing on its client side design and implementation. Using cutting-edge traceroute technologies and a crowd-sourcing methodology, we distribute a smartphone monitor appliance. Smartphones are available in high numbers, they are nomadic and always connected to access networks, which lends the best results. A server appliance coordinates smartphones and analyses collected data.

After a summary of the best traceroute methods available, I'll discuss an implementation of a client appliance for iPhone. I'll discuss how to run a high-speed, efficient and battery-friendly parallel traceroute analysis in a restricted and unprivileged environment. I will also present a new technique to contrast the negative effects of a NAPT router, extremely common in almost all IPv4 access networks, on parallel traceroute analyses.

The correctness and validity of the platform has been verified by reconstructing a map of the GARR network, the Italian research network. Finally I'll discuss what could be done to further improve mYriadi and to extend its potential.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

> "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

The Internet origins can be traced back to a 1960s US government project named ARPANET[2]; its commercialization in the 90s resulted in an unprecedented expansion of the network. In December, 1995 there were 16 millions of users, approximately the 4‰ of the world population[3]. In December 31, 2011 there were 2,267,233,742, about 32.7% of the world population; more details are provided in table 1.1.[1]

World regions	Population (2011 est.)	Internet Users		Growth (%)
		2000	2011	
Africa	1,037,524,058	4,514,000	139,875,242	2,988.4
Asia	3,879,740,877	114,304,000	1,016,799,076	789.6
Europe	816,426,346	105,096,093	500,723,686	376.4
Middle East	216,258,843	3,284,800	77,020,995	2,244.8
North America	347,394,870	108,096,800	273,067,546	152.6
Latin America	597,283,165	18,068,919	235,819,740	1,205.1
Oceania	35,426,995	7,620,480	23,927,457	214.0
World Total	6,930,055,154	360,985,492	2,267,233,742	528.1

Table 1.1: world Internet usage and population statistics. Credits to Miniwatts Marketing Group[1]. All statistics are for December 31 of the specified year.

The speed at which the Internet evolved is one of the many insight offered by the aforementioned table: i.e., Internet grew at a very fast pace in twelve years. This table shows that Africa, Middle East and Latin America grew more than 1,000%, noteworthy. The *bigger* Internet becomes, the stronger is the need to discover which laws control its expansion and why, which model defines its topology.

An Internet evolution model would help in several fields; the following is a non-exhaustive1 list of fields and applications that will benefit from this information:

- protocol development could use such knowledge to improve protocols' efficiency and scalability[4];
- data storage and caching techniques;
- informatics virus diffusion and containment[5];
- infrastructure deployment;
- social studies related to Internet coverage and availability[6].

An Internet topology map is strongly needed. Other than providing a better comprehension of the underlying laws that control Internet expansion and

the ability to *anticipate* its growth, such knowledge would allow better, safer and more precise financial investments.

It is, therefore, a profitable achievement from both a scientific and social-economic perspective, and it's constantly gathering attention since the last years. Nevertheless, a reliable Internet topology map is still missing, not for lack of trying:

- ISPs[1], mobile operators and enterprises do not reveal their network topology;
- previous analysis campaigns had few, ill-located monitors[2], with limited scalability and coverage;
- some *enabling* methodologies were not fully developed until the last five years.

Internet can be described by maps with different levels of abstraction:

- interface maps describe the Internet as an ensemble of links that connect two interfaces together. This is the most detailed map possible; the main drawback of this representation is that the methods that generate such output do not group interfaces as routers. Traceroute offers this kind of detail, and this representation is discussed in the next section.
- Router maps describe the Internet as an ensemble of links between routers. It's possible to create a router map starting from a interface map with dealiasing techniques[7].
- AS maps describe the Internet as an ensemble of links between ASes[3]. It's possible to map IP addresses to ASes.

[1] Internet Service Provider(s).

[2] A monitor is a terminal that executes some kind of analysis to (partially) discover the network topology it's attached to.

[3] An Autonomous System is a collection of connected IP routing prefixes under the control of one or more network operators that presents a common, clearly defined routing policy to the Internet.

There are two main analysis methodologies categories: active and passive. **Active methods**, like traceroute, require monitors to carefully craft packets and inject them probe the network.[8] **Passive methods**, like BGP[4] sniffing, require monitors to collect BGP updates and to examine BGP tables[9]; such monitors must be in a privileged position in the network to receive such information. In this work, we will focus on active methods, using the traceroute method described in section §2.1.

1.1 mYriadi: a client-server Internet topology mapping system

mYriadi is an effort to map the Internet using bleeding-edge network analyses in mobile devices, under control of a server. Using a crowd-sourcing paradigm, in which a smartphone user runs a software, the system collects data on cellular and fixed networks in an opportunistic manner.

This architecture provides **nomadic monitors**, all spread over the globe, thus increasing the network coverage and helping the discovery of links difficult to detect otherwise. Clients request jobs to the server with a preset frequency in the background; the server selects targets with respect to several factors:

- geographic location (i.e. in which country the smartphone is);
- network location (i.e. to which network the smartphone is connected).

Clients run a traceroutesection §2.1 based analysis with a parallel, modified version of the MDA[10] algorithm, described in section §2.3, improved to run in unprivileged environments with limited bandwidth. This improved algorithm, described in section §3.3, is capable of bypassing NAPT, Network Address and Port Translator[5], being able to anticipate the new values a router will insert in a packet when it leaves the router. A modified version of the MIDAR algorithm, described in section §2.4, provides an efficient

[4]Border Gateway Protocol.

[5]A Network Address and Port Translator is a device that replaces private IP address with one public IP address, using higher level information to differentiate outgoing packets from different private addresses.

dealiasing functionality, so clients will send router maps instead of interface maps, taking off some computational load from the server.

The server carefully selects targets for each client who requests that with respect to several factors, like their location both in the globe and in the network. It also manages all client-submitted data, executing the following operations:

- router map to AS map conversion;
- conflict identification, isolation and resolution;
- data aggregation into a unified graph.

The router-AS map conversion greatly simplifies graphs, and also mitigates the effect of a per-packet load balancer[6]. The server-side conflict management assures that a misbehaving client won't insert anomalies in the AS server. The server does not need to run any kind of analysis, and is thus free of such a network and computational burden. Server-side algorithms, design and implementation will not be discussed in this work since they are out of its scope; for further information please refer to[11].

These aspects give mYriadi an edge over all Internet mapping projects:

- mYriadi platform has a low cost: since servers only coordinate clients and they do not participate in analyses (only clients do), a single server machine could handle a sheer number of terminals.
- mYriadi platform is opportunistic, nomadic: clients aren't fixed, but they are attached to the extremities of the network, with considerable gains in terms of coverage.

1.2 The challenge

This thesis is focused on mYriadi's client-side aspects. Therefore, the target of this work is to create a smartphone software that acts as a mYriadi client.

A mYriadi client should be:

[6]The router-AS conversion will hide the effect of per-packet load balancers in ASes' networks.

generic it should be able to execute many different analysis typologies;

opportunistic it should contact the server and request an analysis each time it thinks there is an opportunity;

battery-friendly it should run its analysis as quick as possible, generating as much data as it can without depleting its power reserve;

data-friendly it should guarantee a high confidence level on the data it produces using metered connections as little as possible;

unobtrusive it should be as autonomous as possible in order to not disturb the user.

The product of this thesis is composed of two pieces of software:

MapLibrary a static iOS 4+ library that exposes all the functionalities needed to communicate with the server, run background analyses and manage data on the phone's memory. It implements a traceroute analysis, but it's designed to be easy to add other analysis methodologies.

mYriadi client an iOS 4+ application that uses MapLibrary to run network analyses. It uses Apple's MapKit to provide the user with a geographic visual feedback, as in figure 1.1. *Attractive visual feedback* is a very important aspect in a crowd-sourced application.

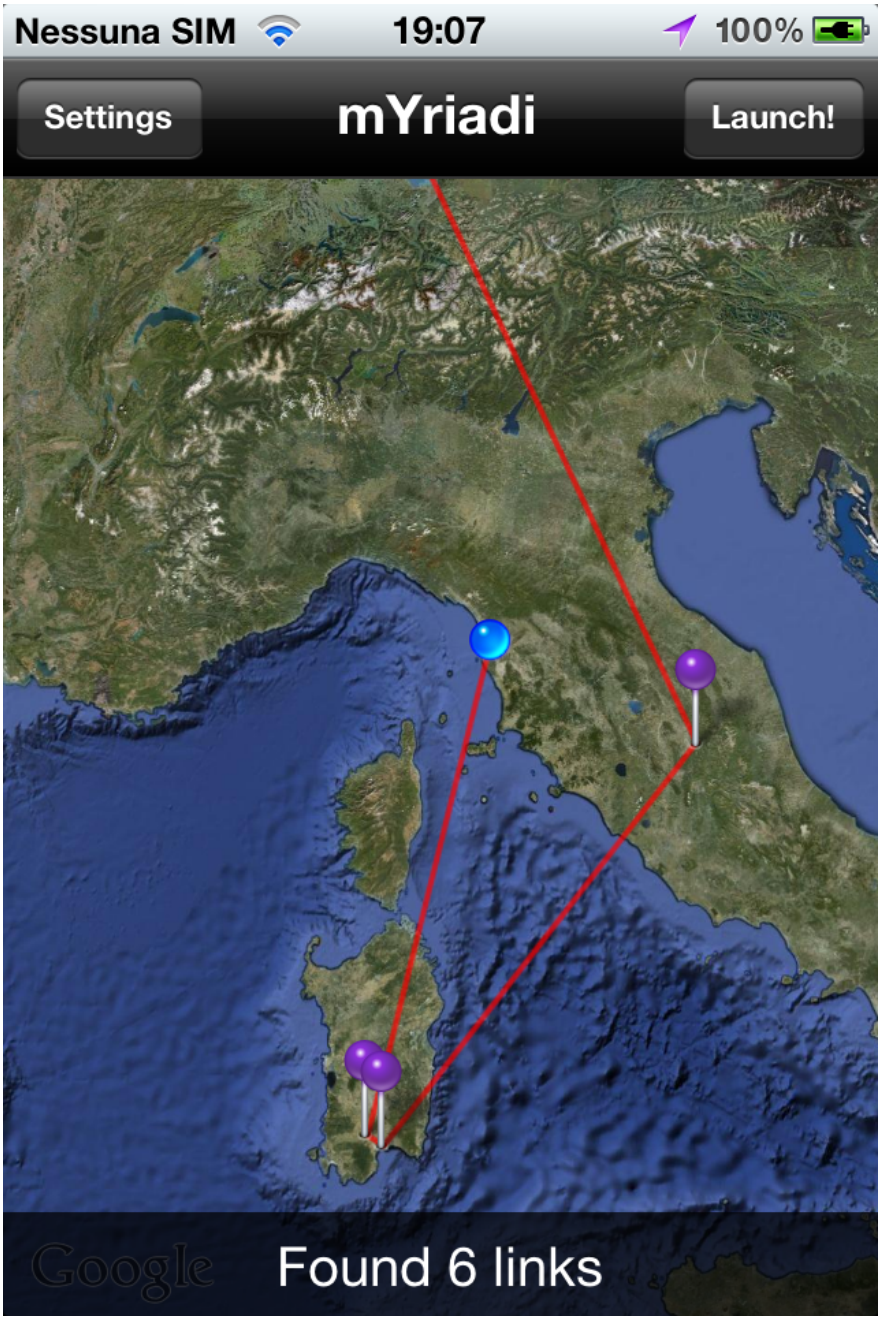


Figure 1.1: mYriadi client interface, running an analysis from Pisa. The user can touch a pin to show more information about each node discovered.

Chapter 2

Traceroute and state of the art

> "If I have seen further it is by standing on ye sholders of Giants."
>
> Isaac Newton

mYriadi is based on traceroute, an active analysis method that discovers routers between two nodes. Traceroute is a well-known technique which also has many flaws. In section §2.1 I'll describe the mechanism behind the *standard* traceroute and its pitfalls. Once these issues have been identified, I'll briefly describe already-developed techniques that solve them: Paris Traceroute in section 2.2 and MDA in section 2.3. MIDAR, a dealiasing technique presented in section 2.4, is used to identify which interfaces are on the same router. In section 2.5 I will also identify some possible competitors and/or related projects and summarize their work and how mYriadi stands up to them.

2.1 Traceroute analysis

A traceroute analysis discovers the sequence of router interfaces an IP packet follows across an IP network in order to reach a given target[12]. The Internet Control Message Protocol[13] is the foundation of this method.

Let's define as **source** the host that runs the traceroute analysis toward a certain **target**. The output of a traceroute analysis is a list of IP addresses: the source therefore creates a list and insert its IP address in it.

The source sends an IP packet to the targeted IP address with increasing TTL, starting from 1. The IP payload contains an ICMP Echo request message[1].

When a router receives an IP packet, it decreases the packet's TTL by 1. If the original value is either one or zero the router will drop the packet and, in addition, it might send back an ICMP Time Exceeded message to notify the source that its packet has been dropped. The source address field of the IP header of this notification is the IP address of the interface that the router used to generate the ICMP Time Exceeded message[2]. The ICMP Time Exceeded also "quote" the IP and level 4 protocol headers of the incoming packet[14].

The source will receive this notification and will then append the notification's source IP address to the list. The source will send then another packet with the TTL value increased by one. If the source doesn't receive any notification, it might do one of the following:

- retry to send the same IP packet again;
- send a new IP packet with the TTL value increased by one, registering that there's a *hole* in the path;
- stop the analysis.

When the destination target receive an IP packet, it might send back an ICMP Echo response message to the source. When the source receives this

[1]Protocols other than ICMP can be used, but they require different analysis arrest criteria.

[2]Some routers will use the same IP address regardless of the interface that received the incriminated IP packet.

message, it will append the target's IP address to the list and stop the analysis.

2.1.1 Inferred topology and dealiasing

Although links are bidirectional, a directed graph is used to represent traceroute-inferred topologies: the interface we discover at each hop (therefore excluding the first IP address, which is the outgoing interface of the host) are the interface the packets reached, assuming all the router behave correctly. There is no way to discover which *outgoing* interface a router used to reach the following router's *incoming* interface.

Obviously, a router is composed of a plurality of interfaces, but the traceroute analysis alone cannot discover when two interfaces belong to the same router. A dealiasing technique is therefore needed.

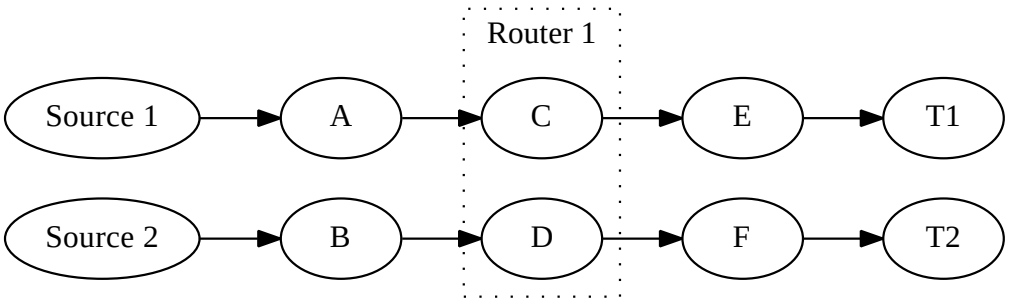


Figure 2.1: Example of a traceroute-inferred topology after two analysis to T1 and T2.

Figure 2.1 show a topology obtained by joining the output of two analyses to T1 and T2 from two different sources. C and D might be interfaces of the same router, but a traceroute-only analysis cannot discover such information. A dealiasing technique must be adopted.

It's possible to evaluate the RTT[3] for each node discovered in the traceroute; unfortunately, there is *no* guarantee that:

[3]Round-Trip Time, the sum of time a packet takes to reach a node in the network and the time the corresponding ICMP notification takes to get back to the source.

- a load balancing operation has not occurred somewhere in the path traveled by the packet;
- the ICMP notification will follow the inverse path back to the source;
- the ICMP will be sent from the same interface that received the originating packet.

From the RTT is possible to evaluate the delay of each discovered link. Let's define the generic link from node U to V as $\{U, V\}$, with delay . A path can be represented as a sequence of links, starting from *Source* and ending with T. If the RTT of a node U, RTT_U, is:

$$RTT_U = 2 \sum_{\{A,B\} \in path\ to\ U} \delta_{\{A,B\}} \tag{2.1}$$

the delay $\delta_{\{U,V\}}$ is:

$$\delta_{\{U,V\}} = \frac{RTT_V - RTT_U}{2} = \frac{RTT_V}{2} - \sum_{\{A,B\} \in path\ to\ U} \delta_{\{A,B\}} \tag{2.2}$$

The information inferred with this method are not accurate with only one measurement, since variations in the traffic load of a link - one of many other factors - may influence the measurement of a node's RTT[15]. This means that a simple delay measurement could lead to negative delay values, which are clearly not possible. Let's consider equation 2.2 and let's suppose that:

$$RTT_U > RTT_V$$

which is a possible outcome. This will lead us to:

$$\delta_{\{U,V\}} < 0$$

2.1.2 Load balancing and its effects on traceroute

A load balancing router (*load balancer* from now on) splits outgoing packets between two or more interfaces; the path starting from each interface is equivalent metric-wise[16]. Network administrators employ load balancing to enhance reliability and increase resource utilization. OSPF[17] and IS-IS[18] intradomain routing protocols both support equal cost multipath. A

multi-homed stub network can also use load balancing to choose which of its internet service providers will receive which packets.[19]

Consider a textbook scenario where the source A wants to execute a traceroute to the target T, as in figure 2.2, where each node represents a router's interface.

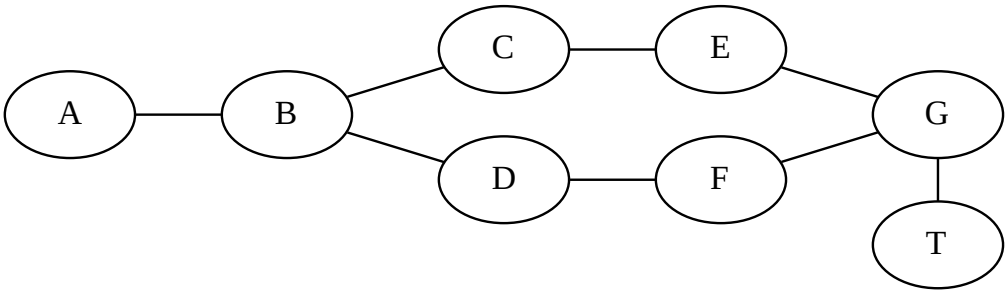
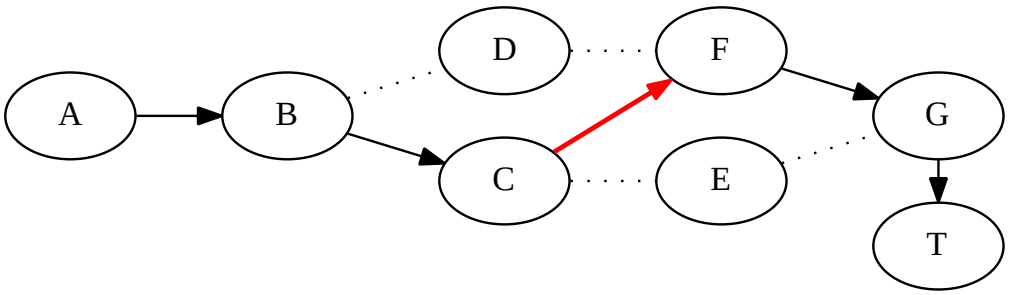


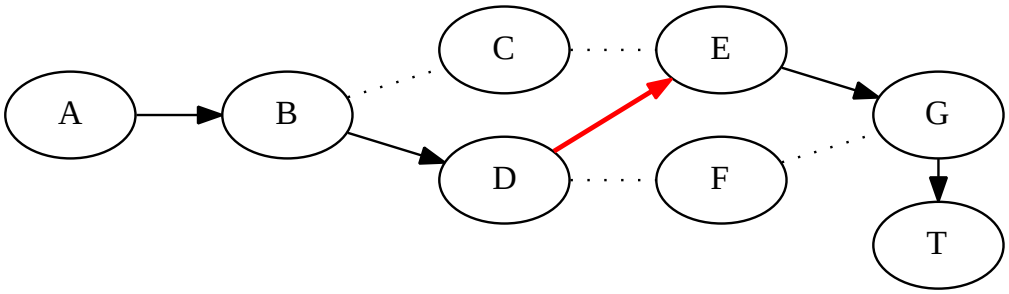
Figure 2.2: load balancing scenario.

Suppose that B is a load balancer that splits 50% of the traffic to T using the C-E-G path, while the remaining 50% uses the D-F-G path. Assuming there is no other network traffic, B alternate between the two paths when forwarding packets to T.

Assuming a packet loss of zero, a traceroute analysis might discover either scenario A or scenario B, as in figure 2.3.



(a) traceroute-inferred topology in a load balancing scenario, outcome A.



(b) traceroute-inferred topology in a load balancing scenario, outcome B.

Figure 2.3: traceroute analysis outcomes in a

In each scenario there is a single false positive (the red arrow) and false negatives (all the dotted arrows).

2.1.3 Zero-forwarding routers

Some routers do not check the TTL value when forwarding an IP packet. These routers are referred as zero-forwarding routers, or zero-forwarders, since they forward packets with a TTL value of 0. Their presence affects the analysis in two ways:

- a zero-forwarder cannot be directly detected;
- the following router in the path will answer twice:
  - the first time when it receives the packet the zero-forwarders should have dropped, with a TTL value of 0;

– the second time when it receives the follow-up packet with TTL set to 1.

This creates a **loop**, that is a router that has itself as a next-hop. Figure 2.4 shows a traceroute analysis scenario where B is a zero-forwarder.

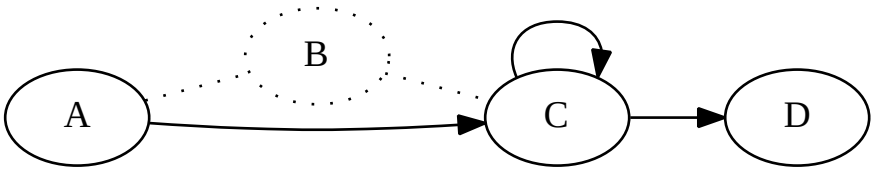


Figure 2.4: traceroute analysis with a zero-forwarding router B.

2.2 Paris Traceroute

Paris Traceroute[20] is an improvement of the classic traceroute analysis that addresses load balancing problems. In their work, the authors recognize the effects that load balancing has with respect to a traceroute analysis, effects already discussed in the previous section.

Let's consider a load balancing router that must choose one of N interfaces to forward a packet. A load balancing policy can be:

per-packet the router uniformly distributes packets one-at-a-time among all interfaces;

per-flow the router uses the five-tuple[4] and the IP ToS field to select the outgoing interface;

per-destination the router only uses the destination address to select the outgoing interface[5]; it's a subset of the per-flow load balancing.

[4]Source and destination IP address and transport port, and transport protocol id (the value of the IP header's Protocol field).

[5]Per-destination load balancing must not be confused with IP forwarding. The latter defines the packet's next hop, the former defines which interface must be used to reach the next hop.

The classic traceroute analysis doesn't define the values that the various fields in the IP and transport headers must have: these values vary during the scan.

Paris traceroute, by requiring that all fields that might discriminate the outgoing interface of a load balancer always have the same value, eludes the negative effects of per-flow (and thus, by extension, per-destination) load balancing[6].

The subset of the headers' fields that discriminates the outgoing interface is called Flow ID. Figure 2.5 shows the IP header's fields that concur in the Flow ID.

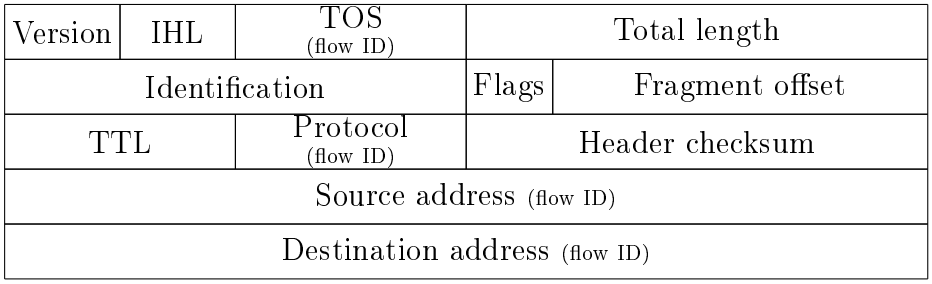


Figure 2.5: IP header with flow ID fields.

Depending on the protocol used, the implementation might vary. While the authors consider ICMP, UDP and TCP, I will not cover the latter, since it cannot be used in a restricted environment.

Let's first consider UDP: figure 2.6 shows the UDP packet header format.

Figure 2.6: UDP header.

The Checksum field is used to recognize an incoming ICMP message as a response to a packet. This requires to add at least 2 bytes of payload to

[6]Per-packet load balancing issues persist.

steer the checksum towards a chosen value. When a UDP packet reaches its target, a ICMP Port Unreachable might be generated and sent back to the source; this notification plays the same role of ICMP Echo Response.

The ICMP implementation uses ICMP Echo Request messages to trigger notifications. Figure 2.7 shows the message format.

Type = 8 (flow ID)	Code (flow ID)	Checksum (flow ID)
Identifier		Sequence Number

Figure 2.7: ICMP Echo request header.

To recognize an incoming ICMP message as a response to a packet, it's necessary to use either Identifier or Sequence Number to store an ID that pairs up the outgoing packet with the incoming notification. Since ICMP Checksum field is part of the flow ID, the unused field must be varied in order to obtain the desired checksum.

Payload may be used, but it's unnecessary and wasteful, and there is not guarantee that the corresponding ICMP notification will carry the original payload.

UDP sockets are always available without special privileges in all systems that implement BSD sockets. Yet the access to ICMP socket eases the pairing process, especially behind a NAPT, using a parallel traceroute analysis. A technique to solve this issue is proposed in subsection3.3.7. In order to use ICMP without using raw sockets, unprivileged ICMP sockets must be available. At the time of the writing, only XNU/BSD kernels[7] and Linux 3.0+ kernels[8] have them. This thesis implements both UDP and ICMP methods over iOS.

[7]Mac Os X, starting from Snow Leopard, and iOS 4 (and above) from Apple are known OSes that support ICMP sockets.

[8]Android 4.0 Ice Cream Sandwich *should* have a 3.0 kernel. Unprivileged ICMP socket support has been introduced in 3.0 kernel as a kernel option, and it has been backported to 2.6 kernel for increased security. Unfortunately no assumption can be made over which device will have such functionality, due to fragmentation in the Android platform.

Some routers might not answer to ICMP requests, but will generate ICMP notifications for UDP packets. This gives UDP an edge over ICMP, although it has a slightly higher data usage.

In conclusion, Paris Traceroute avoids the effects of load balancers, identifying a **single** path. The next step is discovering **all** paths toward a destination.

2.3 MDA: Multipath Detection Algorithm

MDA[10] is an evolution of Paris Traceroute algorithm that tries to discover all the outgoing links of a load balancing router.

Let's assume that we have a packet with flow ID α that is know to reach router U with TTL x. We assume that there are $n = 0$ confirmed routers successors to U, and we want to have a confidence degree of 95%.

A new packet α', created from α with *some criteria*, is then sent with TTL x: until a $\alpha^* \neq \alpha$ packet that reaches U is found, a new α' is created. Then α^* is sent with its TTL increased by one. If a new successor of U is found, n is increased. If six packets are sent without discovering a new link, we are 95% sure that there are n links leaving router U toward our target. In a worst case scenario, where only the sixth packet reaches a new node, 96 packets are sent[9].

There are two packet modification criteria:

fixed destination the IP destination address is kept fixed, while all the other parameters are randomized. This helps identifying per-flow load balancers. Since the destination address doesn't change, the analysis doesn't spread through the network. In a restricted environment, IP source address and ICMP code fields cannot be modified, but are fixed by the OS to the source's outgoing interface and 0, respectively.

varying destination the IP destination address is modified: a /29 subnet is created around the target IP address and new addresses are

[9]The authors assume that routers support up to a maximum of 16 interfaces when load balancing.

created by extracting every possible IP in the subnet, eventually decreasing the prefix length when a subnet is exhausted. All the other fields in the Flow ID are fixed. This method discovers the outgoing links of per-destination load balancers and, since the destination address is part of the flow ID, also of per-flow load balancers. An analysis executed with this criteria has a big spread: this might have a negative impact in some scenarios when running a parallel analysis, inferring non-existent links when encountering a per-packet load balancer.

MDA runs this method for each node it discovers, starting from the default gateway[10]; the output is an interface graph. The next step would be generating a router map from it.

2.4 MIDAR: Monotonic ID based Alias Resolution

MIDAR[21, 22] stands for Monotonic ID-Based Alias Resolution: an IPv4 dealiasing technique, it's an extension of the RadarGun approach[23].

At the basis of MIDAR there is the shared-counter assumption. Each time a router's interface crafts[11] an IP packet, it will insert a value in the IP ID field[12]. This value is generated from a counter, which is assumed to be shared between interfaces of the same router. Therefore, two interfaces on the same router probed closely in time will return similar IP ID values; if probed repeatedly over time, they will return similar time series of IP ID.

MIDAR uses a monotonic test in order to pair up two interfaces. Since all the IP ID values generated from a router come from a counter, the time series built by merging all interfaces' time series must be monotonically increasing. This is a necessary condition, therefore two interfaces that do not meet this criteria do not belong to the same router.

The actual MIDAR algorithm is more complex than the following, since it's standalone. Our goal is to insert its behavior in a traceroute analysis,

[10] We assume that the device running MDA won't use load balancing itself.

[11] This assumption does not apply to forwarded packets.

[12] This field, a 16-bit value in the IPv4 header, is normally used for packet fragmentation and reassembly.

without sending packets explicitly for dealiasing operations. Following this premise, a MIDAR probe could easily be a traceroute probe. An answer to a MIDAR probe is defined as MIDAR answer, and it is composed of the following information:

- answer's IP ID value;
- probe-answer RTT;
- incoming packet timestamp - i.e. its time of arrival.

A timestamp-ordered sequence of MIDAR answers is a time series. Let's suppose that we have two series S_1 and S_2 of their respective interfaces I_1 and I_2. A preliminary analysis will identify all wrap-arounds and it will correct them: since each time series must be monotonically increasing, each value that is not greater than its predecessor must be increased by 2^{16} until it's greater:

Listing 2.1: MIDAR time series wrap-around correction

```
for answer in series:
        while answer.id <= answer.predecessor.id:
                answer.id += 2**16
```

Now let M be the time series created by merging S_1 with S_2. If I_1 and I_2 are aliases of the same router, then it must be monotonically increasing. RTT is used as a timestamp tolerance, as shown in figure 2.8.

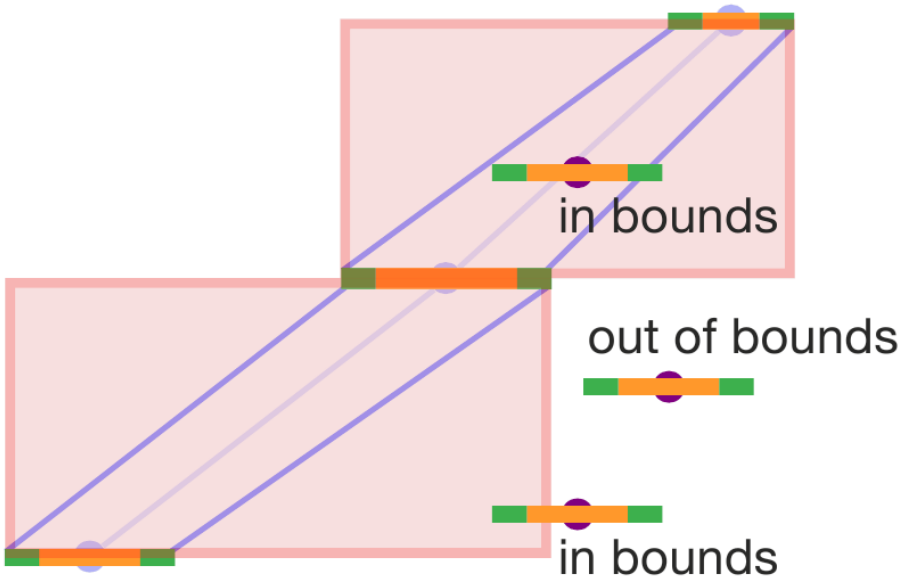


Figure 2.8: MIDAR monotonic test with RTT tolerance

2.5 Competitors

There are, of course, other Internet mapping projects. The most known ones are RocketFuel, CAIDA's ARK and DIMES. I will briefly describe their features and objectives and I will compare mYriadi to them.

2.5.1 RocketFuel

The RocketFuel project produces an ISP's internal network topology mapping system. This project has been tested on 10 ISPs, with about 800 monitors distributed on web servers. Their last paper is dated 2003.

Its monitors are fixed, and each one is restrained to the ISP network it belongs to. In contrast, mYriadi monitors are nomadic and without any

restriction, as they can be anywhere on Earth as long as they can detect their location.

2.5.2 CAIDA ARK

CAIDA(Cooperative Association for Internet Data Analysis) is an organization that fetches, analyses and publishes information about Internet at a global scale. It hosts the ARK project (Archipelago), in which dedicated servers, that act as monitors, are deployed across the globe and they analyse, at regular intervals, the entire set of public IP addresses. Targets are split between monitors. There are three teams with an average of 18 monitors per team. This project is running since 2007.

ARK's monitors are fixed and in low numbers - hardware deployment is needed. mYriadi has a much greater potential, since it's much easier to distribute and host a smartphone application than a device. Our approach requires less investment than ARK's, and has a much bigger user potential.

In addition, mYriadi target selection mechanism is dynamic and aware of the smartphone context; ARK's monitors are not, since their context never changes.

2.5.3 DIMES

The DIMES project offers a free client software that runs on almost all computers and OSes, starting from 2004. Target selection depends on client position and other parameters.

DIMES monitors are fixed, while mYriadi's are nomadic: this provides us greater coverage than DIMES.

Chapter 3

System structure

> "Secretum victoriae in organizatium nunc obvii ist."
>
> Marcus Aurelius Antoninus Augustus

mYriadi platform is composed of a server, that runs the mYriadi server appliance, and a multitude of clients, that run the mYriadi client app.

In section 3.1 I will introduce mYriadi's global architecture. In section 3.2 I will describe the design and implementation of MapLibrary, the library that provides client mapping facilities. A detailed description of an improved parallel traceroute analysis is provided in section 3.3, along with an analysis example to better explain the analysis' logic. In section 3.4 I'll present and describe mYriadi's client-server protocol, which specifies how the two appliances should communicate and which data is available to who.

3.1 Overall system architecture

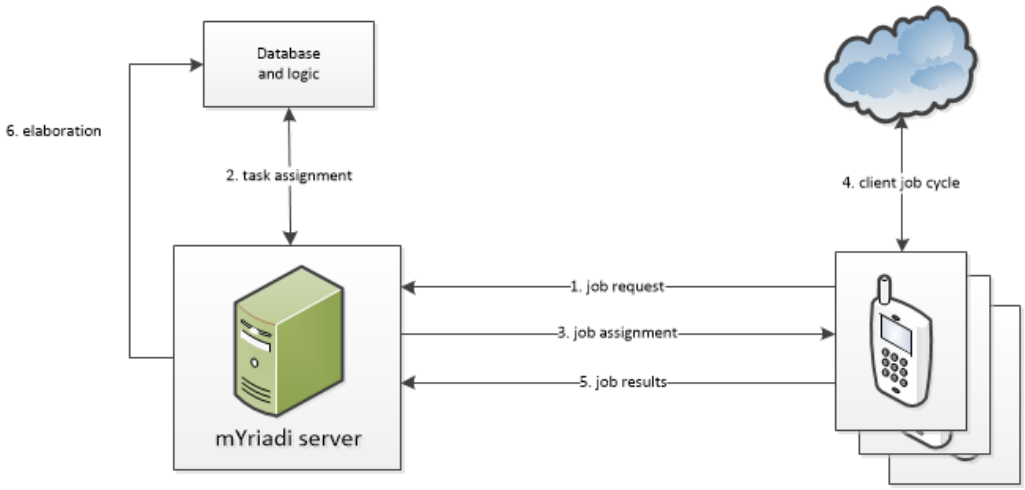


Figure 3.1: overall client-server architecture with job dataflow, supported by mYriadi's client-server protocol

Figure 3.1 shows mYriadi dataflow amongst clients and server. In this subchapter we will refer to this image when discussing mYriadi's core elements.

The overall platform is based on the concept of **job**. A job is a task assigned by the server to a client. Clients request job to the server when they are available or when they think there might be an analysis opportunity. The server may or may not provide a job to a client. If provided, such job would be tailored to the client's network and geographic context, other than its capabilities.

A job defines the analysis that the client will run, along with its parameters. Once a client receives a job, it might start the analysis as soon as the job is assigned or, if something happens[1], delay its execution. Both the client and the server have validation logic that deny or discard jobs: the client can deny a job execution if the network has been changed, or it can discard it if its results are evaluated as not valid. The server can discard incoming job results if they have a low quality. The mYriadi version on which this thesis is built defines a traceroute-based analysis only.

[1]I.E. a short loss of connectivity, or a system reboot.

The interactions between client and server are shown in figure 3.1:

1. job request: the client requests a job to the server;
2. task assignment: the server evaluates whether to assign a job or not and which one, based on its task policy (see subsection 3.1.1);
3. job assignment: the server sends the job to the client, if any (otherwise it sends a no job message and there are no more interactions in this batch);
4. client job cycle: the client "executes" the job, see 3.2.1;
5. job results: the client sends back to the server the outcome of the analysis;
6. elaboration: the server elaborates the received data.

3.1.1 Server's role and task selection policies

mYriadi server's role is twofold:

- It manages clients in order to maximize the amount of network mapping information. It carefully chooses a job for each client according to a certain policy. This is step 2 in figure 3.1.
- It manipulates and merge clients-generated data, step 6 in figure 3.1.

Jobs parameters vary with the specified analysis.

Client geolocation-based policies

A client geolocation-based policy selects targets using the following information:

- client geolocation;
- networks with a known geolocation;
- already known routes between ASes.

In this version, the server implements two different client geolocation-based policies:

- farthest network around the world;
- farthest network in country.

Client network-based policies

A client network-based policy selects targets using the following information:

- the AS number associated to the client's public IP address;
- an already defined target list.

This class of policies is useful to implement mapping campaigns with very precise targets. In this version, only the **static list** policy is defined. This policy has been designed to detect paths starting from a source AS to a predefined ASes list, passing through a chosen transit AS[2].

Client geolocation-network-based policies

This class is a mix of the two aforementioned policies. It selects targets using the following information:

- client geolocation;
- the AS number associated to the client's public IP address;
- networks with a known geolocation;
- network with a know AS number conversion;
- already known routes between ASes.

In the current mYriadi version, the **nearest ASN stub** is the only policy implemented in this class.

[2]I.e. an IXP, Internet eXchange Point.

3.1.2 Geolocation and its power consumption in iOS 4+

When a client requests a job, it also communicates its geolocation. iOS devices have multiple methods to discover their position:

Cell-tower triangulation it requires cellular network connectivity and it's the less accurate, but it's also the less power-hungry method, since it doesn't require to power on additional hardware, and it works indoor too, as long as there's signal.[3]

Wifi-based if the device is in range of a 802.11 network, the access point's MAC address might be used to retrieve the network's geolocation and, by approximation, the device's[4]. If more networks are in range, triangulation might help increasing the accuracy. It requires Internet connectivity and the 802.11 radio must be on, it's more accurate than cell-tower triangulation but less than GPS. It doesn't require wifi authentication and it works indoor. This method is also known as WPS, Wi-fi Positioning System[5]

GPS it requires power to the GPS receiver and it depletes the device's battery really fast. It's also the most accurate method available, but it suffers from poor reception inside buildings, since the GPS signal is quite faint.

It's not necessary to have an high precision[6], so there's usually no need to use GPS - the other network-based location methods are fine, and they are also more power efficient and faster than GPS.

iOS, starting from version 4, offers a Significant Location Change facility, in addition to a fine-tunable facility, that automatically selects the most energy-efficient method (which is usually the least accurate) to detect and signal significant changes in location.

[3] http://searchengineland.com/cell-phone-triangulation-accuracy-is-all-over-the-map-14790 has a non technical description of this feature.

[4] As an example, Google Street View cars detect 802.11 networks and register their location based on the car's. After an access point has been located, a device might use this service within reach of its network.

[5] Not to be confused with Wi-fi Protected Setup.

[6] Server-side task selection policies do not need a very accurate location, since we do not expect any difference when moving between streets of the same city.

3.1.3 Client visual feedback and IP geolocation

The server appliance offers an IP geolocation service. The client can query the server to geolocate an IPv4/v6 address, so it can show on a map the location of each node.

This service concurs in the implementation of a visual feedback for the client's user. In the crowdsourcing paradigm it's very important to motivate users to run the application. Showing them the information collection generated by their devices is a great motivational tool, as other crowdsourced monitoring applications have already demonstrated. For this very reason, it's desirable to run this service, although it's not mandatory.

In order to minimize data usage, these information are cached. The client queries the server on a on-demand basis: it won't generate any query when the application is running in the background.

3.2 Detailed MapLibrary architecture

3.2.1 MapLibrary structure

As showed in figure 3.2, there are three main modules in MapLibrary:

- the Communication module implements the client-server protocol and the job abstraction;
- the Analyses macromodule provides an abstraction to implement analysis methodologies in a modular way;
- the Common classes and utilities macromodule offers commonly-used services and facilities, such as data storage and communication.

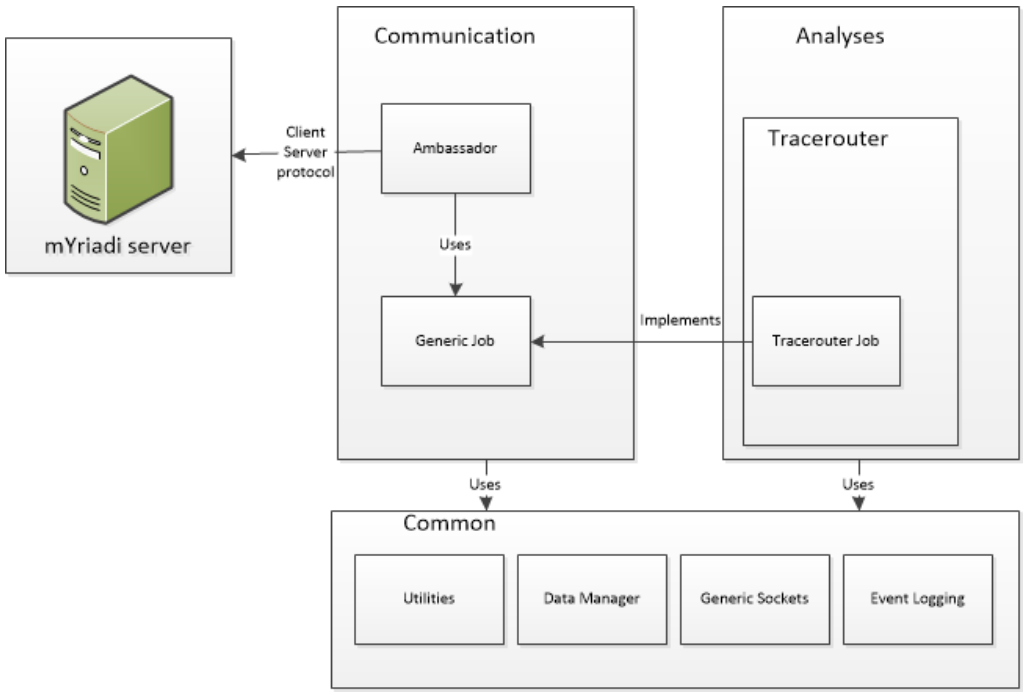


Figure 3.2: MapLibrary modules

Common classes and utilities

This macromodule contains the following elements:

GenericSocket this module offers an easy-to-use generic BSD socket interface, along with a specialized interface that implements traffic shaping functions. There are several advantages over standard BSD socket: they are automatically managed by Objective-C ARC, and they can be specialized to implement specific analysis methods.

IPAddress this interface stores either an IPv4 or IPv6 address, allowing for better modularity. It also performs some operations like detecting if an address is private.

AddressDiscovery this interface fetches the IP address of each device's network interface.

DataManager this interface stores and loads data and settings from the device's persistent memory. This interface can handle all objects that are serializable.

EventLogging this module provides a logging facility, that saves log messages to file and print them on the debugging console.

Analyses macromodule

Each module in this macromodule provides an analysis methodology. An analysis module should provide:

- a specialized job interface, that inherits from generic job, which adapts the already mentioned job operations to work with the analysis;
- a serializable data structure that can be handled by DataManager.

Communication module

The communication module has two main elements, the **Ambassador** and the **GenericJob** interfaces.

The **Ambassador** interface implements mYriadi's client-server protocol. It handles job request, retrieval, execution and data delivery through the GenericJob interface; these four operations are referred as **work cycle**. Ambassador is indeed analysis-agnostic: each Analysis module must specialize GenericJob so that they can be executed by the Ambassador module. This abstraction layer provides modularity and isolation between modules.

Ambassador keeps the application in background when needed. It spontaneously activate the job cycle when a particular condition is triggered, like a change of location or IP address, or the timeout of a periodic timer. In testing environments or intensive campaigns[7] a continuous mode can be enabled, where the client will rentlessly start a job cycle whenever possible - it stresses the battery, so it should be enabled only when necessary.

The full Ambassador flowchart is in figure 3.3. This figure contains both the location manager handler and the work cycle, and it's very useful to understand how the client operates.

[7] As in our validation task, see chapter 4.

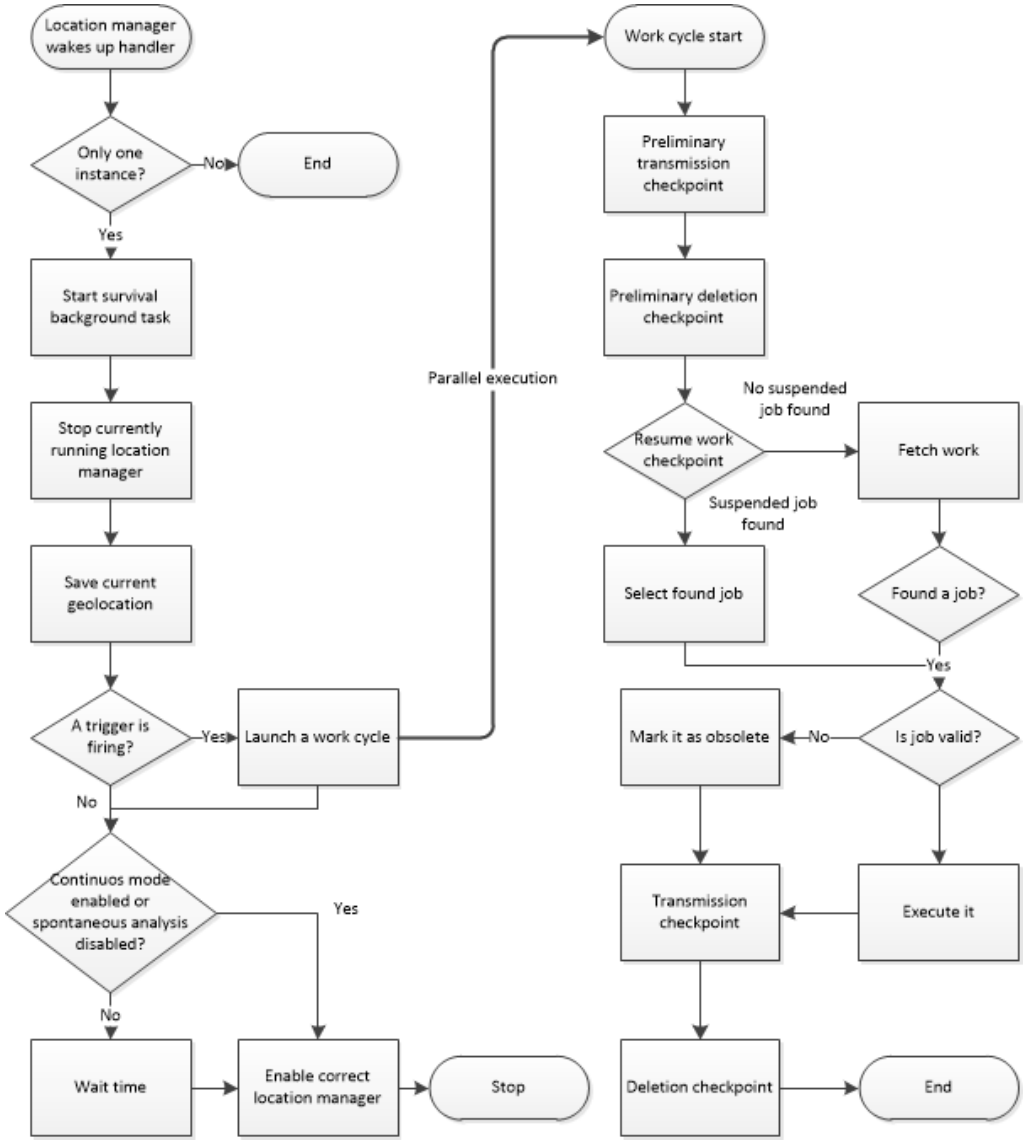


Figure 3.3: full Ambassador flowchart

Job cycle The client-side representation of a job has a predetermined lifespan, which encompasses 8 different statuses. Figure 3.4 shows a status transition graph.

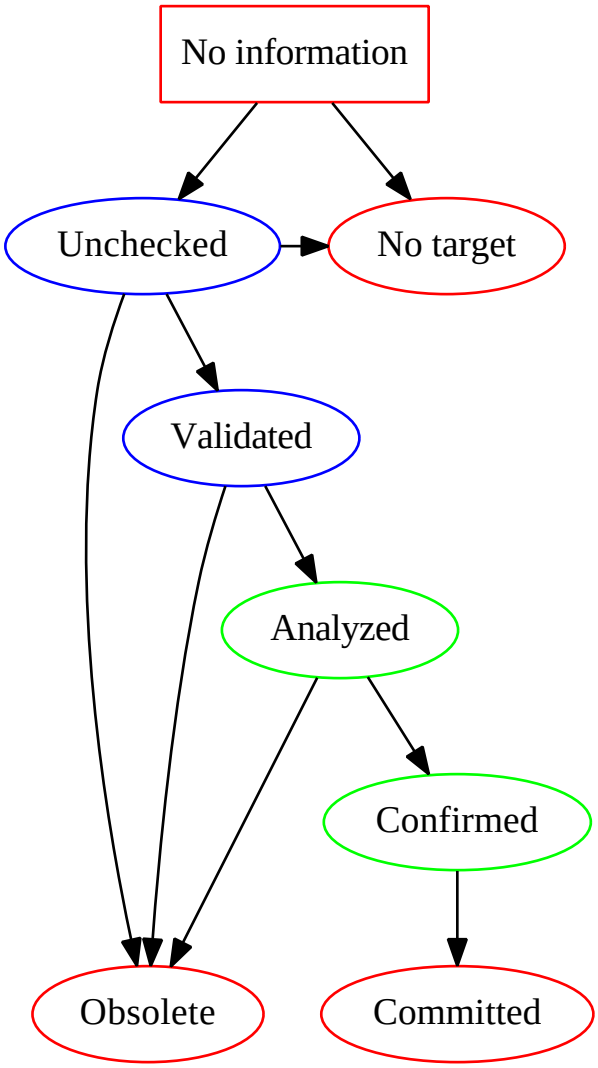


Figure 3.4: job status transition graph. Red nodes are marked as deletable, blue nodes are marked as work required and green nodes are marked as ready for transmission. A newly-created job always start from "No information".

There are three status categories: **deletable**, **work required** and **ready for transmission**; they are used in control points inside a work cycle.

There are a few job operations that modify a job status. These operation

must be executed without interruptions, and any amount of time might pass between the execution of two different operations.

Job request the ambassador requests to the server a job, using a newly created client-side job (with no information status) to store the results of the operation. A job may or may not be assigned, therefore the job becomes either unchecked or with no target.

Job validation an unchecked job must be validated before starting an analysis, since the context might have changed since job request. Already validated jobs can be validated again. A job might become obsolete; the criteria for such decision are analysis-dependent.

Job analysis execution the analysis is executed. To guarantee that a job is not obsolete before executing the analysis, a validation operation is always executed before launching the analysis. The job becomes either obsolete or analysed, starting from a validated or unchecked job.

Job confirmation the analysis results of an analysed job must be validated, implementing a client-side early refusal detection (i.e., the server might always drop results that match a certain pattern - if the client can execute this evaluation it's better to drop the job client-side, to avoid waste of bandwidth, server time and energy). The job becomes either obsolete or confirmed.

Job commit the analysis results of a confirmed job are sent back to the server. The job always becomes committed.

Job transmission the ordered execution of job confirmation and commit.

The following list describes each status:

No information the job has been just created, therefore it stores no information. A stray job with no information can and should be deleted.

Unchecked after a successful job request, a job becomes unchecked. An unchecked job defines the analysis to execute, and stores relevant context information when the job has been assigned. It requires work, since its analysis has yet to be executed.

No target a client-side job becomes with no target after an unsuccessful job request, since the server has no job for the requesting client - it can be safely deleted, since it serves no purpose.

Obsolete a job that refers to an old context and that is, therefore, obsolete and deletable.

Validated a job is valid at a certain time if the context hasn't changed. An already validated job can become obsolete if validated again. It requires work, since its analysis has yet to be executed.

Analysed a job that has analysis results, that is, its analysis has been completed. Nothing is known yet about the validity of its results. This job is ready for transmission.

Confirmed a job with valid analysis results, which is therefore ready for transmission.

Committed a job whose results have been sent to the serve without errors. The server have either accepted or dropped them. It can be safely deleted.

Work cycle A work cycle is composed of the following steps:

1. (preliminary transmission checkpoint) execute a job transmission operation (which in turn executes a job confirmation) for every job that is ready for transmission (step 5 in 3.1);
2. (preliminary deletion checkpoint) delete every deletable job;
3. (resume work checkpoint) find a job that requires work and refer to it as current job - if a job is found skip to step 5;
4. (fetch work) create a job and execute a job request (step 1 and 3 in figure 3.1) - if job has no target, delete it and terminate the work cycle, otherwise refer to it as current job;
5. (execution) run a job analysis execution on the current job (which in turn executes an additional validation), step 4 in figure For 3.1;

6. (transmission checkpoint) execute a job transmission operation (which in turn executes a job confirmation) for every job that is ready for transmission (step 5 in 3.1);

7. (deletion checkpoint) delete every deletable job.

Unless there is a temporary failure, steps 6 and 7 will only work on the current job. A work cycle is composed of multiple job transmission and deletions; only one job analysis per cycle might be executed.

Triggers and background execution There are multiple ways to start a work cycle:

manual the user explicitly starts an analysis interacting with the UI[8];

timeout if enabled, a timer will set off periodically and start a work cycle every x seconds[9];

location changes if enabled, a significant location change will start a new work cycle;

network changes if enabled, a network change (WiFi SSID, cellular network operator, IP address) will start a new work cycle.

iOS applications have severe background execution limitations. When an applications moves from foreground to background, the OS will *freeze* the application's context, which will be either reloaded, if the user brings the application back to foreground, or deleted, if the OS needs more memory. iOS provides a few handlers that allow the application to react to these events and store its state and any unsaved data. An application may request time to finish background tasks; the OS will keep the application running in background until it decides that background execution is not allowed anymore[10]. The application must *signal* that it's executing background-enabled code as in listing 3.1, as well as an emergency expiration handler that stops the background task as fast as possible. If an expiration handler doesn't stop its task, iOS will kill the application.

[8]Some UI designs might not provide such method of interaction by choice.

[9]If the continuous mode is enabled, $x = 0$.

[10]Apple documentation does not specify which conditions must be met to stop background execution.

Listing 3.1: background task signaling

```
1  UIApplication* app = [UIApplication sharedApplication];
2
3  // define a background task with a termination handler
4  UIBackgroundTaskIdentifier bid = 0;
5  bid = [app beginBackgroundTaskWithExpirationHandler:^{
6      // expiration handler code block
7          [app endBackgroundTask:bid];
8  }];
9
10 // background code here
11
12 [app endBackgroundTask:bid];
```

Timeout and network changes methods requires exploitation of iOS location manager to run in background. Whenever the application starts, Ambassador will start a location manager. If spontaneous analyses are enabled it will start the standard location manager, otherwise it will start the significant location changes manager. The standard location manager provides an opportunity to run in background, since it sends a message whenever the location changes or the accuracy improves.

The handler that receives this message will be executed in background: iOS will un-freeze its context, but the application stays in background. The handler will execute the following operations:

1. ensure that at most one instance of this handler is running at any time;
2. signal a *survival* background task, its expiration handler will ensure that the appropriate location manager is enabled;
3. stop the location manager from notifying changes, in order to preserve battery;
4. save the current geolocation;
5. if at least one trigger is enabled launch a work cycle in a new thread;
6. if continuous mode is enabled or spontaneous analysis are disabled re-enable the appropriate location manager and stop;

7. wait until there are x seconds since the last execution of this handler;
8. enable the appropriate location manager and stop.

3.3 Tracerouter module: a parallel traceroute analysis in restricted environments

The **Tracerouter** analysis module provides a traceroute analysis. I developed an enhanced Parallel MDA (PMDA) which remains fully operative in unprivileged and restricted environments.

There are many challenges to overcome:

1. This analysis must support multiple protocols, namely ICMP and UDP. An high modularity is required to achieve this result; this modularity is provided by the introduction of **generic probes**, described in subsection 3.3.1, which specify the content of a probe in a protocol-independent fashion.
2. It must be designed to switch easily to IPv6[11]; IP versions and protocols must be easily interchangeable. This, again, is achieved by the generic probes, and by using the IPAddress interface introduced in subsection 3.2.1.
3. Parallel execution with multiple threads to speed up its execution. This requires a special synchronization technique named **safeguard** introduced in subsection 3.3.3.
4. It must be able to bypass NAPTs. This requires the ability to predict how a NAPT router will modify a probe's field; this method is known as **NAPT bypass** and is described in subsection 3.3.7.
5. It must provide an efficient dealiasing technique, which doesn't require the transmission of additional probes. This is achieved by using a slightly modified version of **MIDAR**, and by saving all the relevant dealiasing information contained in every single answer in the so-called **MIDAR database**. The dealiasing mechanism is defined in .

[11]IPv6 is not supported in this work, but is recognized as a future work.

6. It must work in a restricted environment, without the help of raw sockets. This imposes additional limits on the usage of the selected protocol. All the considerations made in the following pages account for this requirement.[12]

A tracerouter analysis has a well-defined iter, articulated in three steps:

Initialization apply the default settings and then those specified by the server, and then detect the IP of the phone's default gateway.

PMDA concurrently discovers new links, updating the MIDAR database in the progress; falls back to sequential MDA when needed and for the shortest amount of time possible.

Dealiasing use MIDAR to group interfaces of the same router together.

The algorithm described in section §2.3 can be adapted to run in a multiple thread; before going into details I will first introduce a few concepts and mechanisms that are widely used in this analysis.

3.3.1 Generic probe

A generic probe is an abstraction of a probe that is used during the traceroute process, in order to support both ICMP and UDP analysis methods in a seamless fashion. Such probes store a (generic) flow ID and some auxiliary information. Generic probes are created during an analysis, but they are specialized into the chosen analysis and network protocols before being sent.

The flow ID is composed of six fields:

Protocol a byte, corresponding to the IPv4 protocol field or the IPv6 Next Header field.

ToS (Type of Service) a byte, corresponding to the IPv4 ToS field or to the IPv6 Traffic Class field.

[12]This is especially true for all considerations about a possible Android implementation.

IP addresses source and destination IPv4/6 addresses.

First word the first two bytes of the IP payload: ICMP Type & Code fields or UDP source port.

Second word the third and fourth bytes of the IP payload: ICMP checksum or UDP destination port.

This information is not enough to send a traceroute probe. A generic probe stores some additional auxiliary fields:

TTL a byte, corresponding to the IPv4 Time to Live or IPv6 Hop Count fields.

Target Address IP address, either v4 or v6, that the probe is supposed to reach. This value supports our PMDA implementation, but it won't be copied in any field of the IP header. Please note that this is not the same thing as the destination IP address.

SN (Sequence Number) two bytes that *ideally* uniquely identify a probe and its answer. It's ICMP SeqNum field or UDP Checksum field.

The ICMP implementation requires that the checksum won't change even if we modify the SeqNum field. This can be achieved by modifying the **ICMP identification field** to a SN-dependent value that will always produce the same checksum. Since it's not possible to reproduce the value $2^{16} - 1$, this checksum won't be used by the Tracerouter module as part of the flow ID[13]. Please note that a port to an Android device with a 2.6 kernel version cannot use ICMP socket, so this protocol is not available.

The UDP implementation requires that the checksum can be fixed to an arbitrary value but $2^{16} - 1$, as per above. The source ports cannot be changed, and the length field value cannot be controlled; the only way to control the checksum is to add a two byte payload.

Both ICMP and UDP use the same checksum algorithm to protect data; here it is:

[13] It's impossible to have the word-by-word sum equal to zero, which is the only value that can generate $2^{16} - 1$.

1. Let S be a 4 bytes unsigned integer, initialized to 0.
2. Initialize the checksum value to 0.
3. Read the protected data 2 bytes at a time (using 0 as padding if necessary) and sum this word to S.
4. Until $S < 2^{16}$ then let S be the sum of its most significant word and its less significant word.[14]
5. $NOT(S)$ is the checksum.

[24] proposes optimized evaluation methods.

ICMP checksum covers the ICMP packet's header and payload. We are interested in keeping the checksum fixed to **second**, in setting SeqNum to **SN** and not having any payload. Therefore we can use the ICMP Identification field to manipulate the checksum, as the pseudo-code in 3.2 shows, assuming that the initial ID value is 0.

Listing 3.2: ICMP Identification field evaluation

```
1 word current = evaluateICMPChecksum(probe);
2 word desired = probe.second;
3 word invc = ~current;
4 word invd = ~desired;
5
6 word ID = invd -invc -(invd < invc);
```

UDP checksum covers the whole UDP packet plus the pseudo-header, which is composed of the IP source and destination address, the IP Protocol field value and the payload length. The pseudo-code in listing 3.3 shows how to evaluate the correct payload value to fix a chosen checksum.

Listing 3.3: UDP payload evaluation

```
1 word current = evaluateUDPChecksum(probe);
2 word desired = probe.sn;
3 word invc = ~current;
4 word invd = ~desired;
```

[14] This implies that $2^{16} - 1 + x$, $x \in \mathbb{N}^+$ becomes x, therefore it's impossible to obtain 0.

```
5
6 word payload = networkOrder(invd -invc -(invd < invc))
    ;
```

In line 6 of both listings we have:

$$invc - invd - x \mod 2^{16}$$

where:

$$x = \begin{cases} 1 & invd < invc \\ 0 & invd \geq invc \end{cases}$$

x counters for the one's complement sum effect that verifies if the inverted current checksum is greater than the desired. Be it so, the payload must be big enough to cause a wrap around in the checksum evaluation or, in this case, it will add 1 to the most significant word of the 4 byte counter. This means that there would be a +1 added to the 2 byte checksum value before complementing, so it would assume the value $invd + 1$. x counters this effect.

3.3.2 Probe-answer couplings

An answer is defined as a notification triggered by a certain probe. Pairing correctly an answer to its originating probe in a parallel environment is both crucial and challenging; this operation is defined as coupling. Incorrect coupling will surely corrupt the inferred topology.

We can identify two coupling methods:

Time coupling there is only one probe that it's waiting for its answer at a time. This means that the first answer received *should* be assigned to that probe.

SN coupling an answer, which usually stores a portion of the generating probe including its SN, is assigned to the probe that has the same SN.

They both have pitfall:

- time coupling cannot be used in a parallel environment by definition;

- time coupling could be *poisoned* if a stray notification of a previous probe reaches the phone;
- SN coupling might fail for a number of reasons:
  - a stray notification can again *poison* this coupling method;
  - some routers do not include the originating packet header in the notification payload;
  - some router might even alter the notification payload;
  - the probe crossed a NA(P)T.

Stray notifications are, at a first glance, troublesome, because there isn't nothing that we can do to ignore their effect. Still, they are rare events and we can go further using a **shared, increasing SN counter** to generate these values: since SNs are stored in two bytes, there can be 2^{16} of them. The counter might do a wrap around during an analysis, but it will take enough time so that the IP protocol will have already dropped roaming notifications (they too, of course, must adhere to the TTL logic).

When routers alter the notification payload, only time coupling can be used.

When using UDP with fixed destination mode, a different coupling can be used:

Port coupling there is only one probe that it's waiting for its answer at a time *for a certain UDP destination port*. This means that the first answer received *should* be assigned to the only probe with that destination port. This method cannot be used in conjunction with a varying destination method.

It limits the degree of parallelism achievable, but in turn offers a much easier way to cope with NAPT. This should be used only when necessary, since time coupling allows for faster executions. A positive effect of this method is that it nullifies NAPT effects, as seen in subsection 3.3.7.

This coupling method is critical for a parallel implementation of MDA in Android platform previous to version 4.0, precisely all the devices with a 2.6 kernel. They all lack ICMP support, so they cannot use ICMP socket to receive notification, but they are forced to rely on SOCKERROR. This

method does not provide access to the UDP checksum, so it is not possible to implement SN coupling. These device cannot provide a parallel MDA analysis with varying destination.

Answer Dispatcher and SN reservation

Sequence Numbers are mainly created using a shared counter and a generic probe stores only one of them. The Answer Dispatcher entity is used to control which SNs can be used, and it couples notifications to the corresponding probe. When a thread creates a probe, it register the generated SN to the Answer Dispatcher: this operation is called SN reservation. If a SN is not available, the thread will generate a new one until it surrenders (after a predetermined number of retries) or until it succeeds.

A SN reservation associates a SN to a probe. Only one probe can use a SN at a time, but a thread might reserve more than one SN for the same probe.

3.3.3 Sending, receiving and safeguard mechanism

To send a probe, a **sender thread** extract from a shared counter a sequence number and it will register it to Answer Dispatcher; after this reservation, it can finally send the probe. After getting the current system time, it will wait on a private semaphore, linked to its reservation with Answer Dispatcher. This wait has a tunable timeout, so a thread doesn't wait forever for a notification that might never come.

Receiving notifications is accomplished by using an ICMP socket; a single thread, the **receiver thread**, is allowed to operate on it. The receiver thread will use sequence number coupling to dispatch every incoming notification, using Answer Dispatcher. After receiving a notification, the receiver thread will immediately store the system time in the Answer object.

When the receiving thread can't dispatch an answer, it will raise a safeguard needed flag and it will drop the answer. At least one sender will timeout. A sender reacts to this event by queuing for a safeguard send (or safesend). in the so-called safeguard queue. A sender in this queue is denoted as safe sender. The safeguard rules are the following:

- the safeguard needed flag can be raised only if there are senders waiting for an answer, thus preventing stray answers to trigger the safeguard;
- when a thread tries to send a probe, the safeguard will freeze it if the flag is up and there is at least one thread in the safeguard queue;
- the safeguard queue will release a safe sender at a time only when no other thread is sending, safeguard or not;
- the last thread leaving the safeguard queue will lower the safeguard needed flag.

This mechanism allows for non-continuous parallel execution, using SN coupling by default, falling back to time coupling only when it's necessary.

When an answer is dispatched, the corresponding sending thread will wake up. It will therefore compute the round trip time of the probe and it will store it in the answer, along with the transmission time.

When using UDP with fixed destination and a NAPT device has been discovered, each port has its thread queue to guarantee mutual exclusion. More precisely, this is the implementation of port coupling.

3.3.4 Retransmission mechanism and TTL skipping

A sender thread has a finite number of retries available, that allows it to resend a probe which didn't receive any answer. A not-working node is not the only cause to a sender thread not receiving any answer to a specific probe: some routers might operate correctly without emitting any notification. Such routers will truncate an MDA branch, reducing the amount of topological information that can be inferred.

TTL skipping changes this behavior: a sender might increase the probe's TTL and restart the send procedure again. TTL skipping has four parameters that defines its behavior. TTL skipping is enabled by default, but the server might disable it, or it might even tune its parameters. These parameters are:

Max Initial Bonus the maximum TTL increment that can be used when detecting the gateway;

Max Vertical Bonus the maximum TTL increment that can be used when detecting regular nodes;

TTL Token Limit maximum number of tokens at a time;

TTL Token Refill amount of tokens added after a node discovery.

When discovering nodes other than the gateway, a token mechanism is used. Tokens are stored in a dispenser which has a maximum capacity and a refill ratio as mentioned above. When a thread wants to increase its probe's TTL it fetches a token from the dispenser, with respect to Max Vertical Bonus. If there are no more tokens, or if the bonus limit has been reached, the thread will close that MDA branch.

3.3.5 Topology graph

The outcome of a traceroute analysis is a graph. The algorithm, during the first two phases, creates links between interfaces. During these two phases, the graph nodes are interfaces that produced a notification[15]; the edges are not links between the known interfaces, though. With a traceroute analysis it's not possible to discover what's the interface that does not produce a notification, therefore edges are links between an interface node and an interface.

A node stores different information:

- the IP address of the interface it represents;
- the router ID[16] of the interface, which defaults to -1;
- the list of outgoing edges;
- the list of incoming edges.

A router ID is an integer number that groups nodes into routers. A value of -1 indicates that interface is to be considered as a per-se router. If two nodes have the same value $k \geq 0$ then the router k has two interfaces.

[15] With the notable exception of node 0, which is always the iPhone and always has IP address 0.0.0.0.

[16] This has nothing to do with router ID as seen in many routing protocols.

It's important to note that nodes *cannot* hold information about round trip times, since there might be different paths that could reach that node; RTT is not an information that depends uniquely on the node position, but also depends on the path two packet takes to complete the round trip.

A link stores the following information:

- the IP address of the two interfaces that it's connecting:
  - the target IP address is the IP address of the interface that produced the notification;
  - the precursor IP address is the IP address of the interface that preceded the target interface during its discovery;
- the link's delay, in milliseconds;
- a TTL skip value.

The TTL skip value is a positive integer that indicates how many fake nodes are between the two nodes. This value defaults to zero, but there are three different events that might change it to something different:

1. the TTL skipping mechanism increased the probe's TTL - the skip value would be the TTL increment used;
2. an incoming notification had in its inner IP header a TTL value to 0, indicating a zero-forwarding router - the skip value would be 1.

The first method is common practice in almost all traceroute implementation. The second method could not be tested since I never got access to a router that I was 100% sure it was a zero forwarder, although this method has been already proposed in [20].

3.3.6 Traceroute algorithm's phases

As already mentioned before, the traceroute procedure is composed of three main phases: initialization, Parallel MDA and dealiasing.

Initialization

The main thread[17] load settings from memory, and it will then override them with server-specified settings. It will then discover the phone's IP address of the interface that will support the analysis.

Next, it will start the send-receive mechanism, initializing the receiver thread. It will then craft a probe with a random flow ID, that will provide a basis for the rest of the procedure. The probe is sent with TTL set to 1, in order to discover the default gateway; if enabled, TTL skipping might be used.

It will then create an **horizontal explorer** (HE) thread for the just-discovered node. An HE thread coordinates the exploration from a node, identified by the probe who reached it. The probe stores this information in his target field. It also stores the path such probe crossed.

The initialization phase is concluded. The main thread will wait until there are no more horizontal explores; then it will enter the dealiasing phase. Until then, the algorithm moves to the parallel MDA phase.

Parallel MDA

This phases defines the behavior of HE threads. There are also **vertical explorer** (VE) threads, generated by a HE, which discover new links using a copy of the HE's probe.

Let's define two numbers: W, the exploration increment, and L, the exploration limit. These numbers control MDA's confidence level. An HE creates a Target Generator and a Target Dispatcher objects, that will control how each probe is changed by a VE. These objects can be configured to implement MDA's fixed and varying destination policies. Target Generator modifies the destination IP address, if needed, while Target Dispatcher guarantees that at most L vertical explorers will be created, and it will also assign to each explorer a different IP destination address, again, if needed.

An HE creates W vertical explorers; it will then wait until there are no more VE created, directly or not, by him.

A vertical explorer will execute the following operations, in order:

[17]The main thread would be the thread that actually starts the traceroute analysis.

1. submits its probe to the Target Dispatcher, which will change the flow ID according to the selected MDA policy or it will deny the VE's execution, killing it;
2. it sends the modified probe without changing its TTL, to verify if it reaches the HE's node: if not, it will stop;
3. it increases TTL by one;
4. it sends the modified probe to discover a new interface, optionally using the TTL skipping mechanism;
5. if it receives an answer, a subset of this information is stored in MIDAR database;
6. if it discovers a link to an new interface, it launches a new batch of W vertical explorers and it will make the just received Answer to its horizontal explorer.

The HE receives a set of Answers from its sibling. After eliminating duplicates and invalid links (loops, links to invalid addresses), it updates the graph by adding a link to a newly-created node for each answer. It then starts a new HE thread for each node, with an updated probe and path.

An HE won't analyse a node if it's already been analysed, therefore a list of all the already explored nodes is kept. The server can specify an **exclusion list**; it's a list of IP address that the server is not interested into. If a HE detects a link to an IP in the exclusion list, it will completely stop the analysis. Nevertheless, the client will still send the inferred data to the server, that will then choose what to do with it.

When all the HE finish, the main thread wakes up and starts the last phase, dealiasing.

Dealiasing

This last phase uses data collected during the other two phases. In fact, each time a sender thread receives an Answer it creates a **MIDAR answer** object, a subset of a regular Answer. It contains:

- the IP ID value, found in the IPv4 header;

- the computed round trip time;
- its timestamp, the arrival time of the answer.

This object is then inserted in the MIDAR database. This database is, simply put, an associative map that assign a set of MIDAR answers to an IP address. Therefore, a sender thread will insert the answer in the set corresponding to the IP address that sent it.

Suppose the inferred topology graph has N nodes, and that each node can be identified by a number $i \in \mathbb{N}$, where $0 \leq i < N$. The main thread creates a $N \times N$ reachability matrix R from the graph. The generic element R_i^j at row i and column j is either 1, if node i can reach node j, or 0, if otherwise. Computing this matrix is very useful to avoid merging interface that *cannot belong* to the same router.

In section §2.4 I've described the monotonic test between two time series extracted from two different sets of MIDAR answers. Let's represent a router A as a set of IP addresses. A router will have a total time series obtained by merging all the time series of each interface it has.

The algorithm is composed of the following steps:

1. for each interface, create a router with the interface's IP address as its sole element.
2. for every possible pair of routers (A, B):
   a) if $\exists\, i \in A,\, j \in B : R_i^j = 1 \, OR \, R_j^i = 1$ then do not merge this pair.
   b) if the monotonic test over the total time series of A and B passes, then merge the two routers together:

      $$AB = A \cup B$$

      otherwise do not merge this pair.

This algorithm stops when there is only one router or when all the possible combinations fail. To avoid running the monotonic test multiple time, a subtly different approach can be used:

```
routers = [list of routers, length N]

for i in 0 to length(routers):
    j = i +1
    while j < N:
        if [i and j notConnected] and monotonicTest(
            routers[i], routers[j]):
            routers[i] = routers[i] + routers[j]
            routers.delete(j)
        else:
            j = j +1;
```

Using this algorithm only one test per pair is executed. After all routers have been identified, each node's router ID is set to the router number it belongs to.

MIDAR can be implemented only on Android devices that sport a 3.0 kernel or greater, since all the required information used in the dealiasing process can only be obtained via ICMP notifications, which are not accessible without ICMP sockets.

3.3.7 NAPT bypass

NATs[18] introduce variations in the source IP address: they replace private address (from a private, internal network) with a public address, extracted from a pool[25]. An outgoing packet therefore has a different IP source address before and after the NAT. This actually means that everything that *depends* on the source address will be altered. This IP address will be the same for all the duration of the analysis, therefore this behavior allows for the flow ID to be constant, just not the same in the two domains. When an incoming packet is being sent to an address in the NAT pool, its destination IP address will be changed to the corresponding private address and the packet will be forwarded to the correct host.

A ICMP-based tracerouter tries to keep the ICMP checksum field constant, and identifies probes using the SeqNum field. None of these fields are altered

[18]Network Address Translator, device that translate IP addresses with a one-to-one relationship from internal and external addresses.

by NATs, so the probe contained in its corresponding notification will have the same values in these important fields.

NAT has a destructive effect on UDP traceroute: the router modifies the source address, which concurs in UDP checksum. A NAT router must therefore recompute the UDP checksum and replace the original with the new, modified one. When a notification comes back, the NAT router cannot recover the correct checksum value in UDP. UDP-based tracerouter uses the checksum as a sequence number while operating in UDP, therefore the standard mechanism won't be able to assign notifications to probes.

A flexible way to solve this issue requires the knowledge of the device's public IP address. There are two ways to achieve this goal:

- use a UPNP message to discover the router's public IP;
- the server sends this information back to the client when assigning a job.

Using UPNP is faster and somehow more reliable, since this operation can be repeated each time an analysis starts. This method is implemented in mYriadi but it's completely disabled, since only a few routers offers UPNP support and have it on by default. mYriadi uses the second method, which cannot be repeated. A traceroute job is not validated if it's old enough, since this IP address might have been changed.

Once the public IP address is known, it's trivial to compute the value of this checksum in the public network. Each UDP probe will be registered to two different sequence numbers: the regular checksum and the external checksum.

NAPTs[19] are even more *destructive*: they not only change the IP source address but also the transport-level source port. [26] first introduces the NAPT term; the process is also known as PAT[20] and IP masquerading. Let's consider a NAPT which only has one public IP address available for translation; a NAPT will then create a mapping between a private source port and a public source port; this binding will last some time, according to whichever policy the device is using. As shown before, we can compute

[19] Also known as PAT, Port Address Translation.
[20] Port Address Translation.

what the checksum would be outside the private network. Each time a thread sends a probe and fails, if at least an unknown notification is received safeguard will be triggered. Each safe send operation will be extended to detect which public source port the router has assigned to the private source port.

Suppose that a safe sender received an answer. This thread will then check if the checksum is one of those two it would expect behind a NAT. If they differ, then the thread will compute:

$$\Delta = Checksum - Checksum_{est}$$

We are expecting a single mismatch, localized in the source port field. Therefore, the new port would be:

$$new = old + \Delta \mod 2^{16}$$

or:

$$new = old + \Delta + 1 \mod 2^{16}$$

We cannot tell which one it is: the last phase of the checksum introduces this indetermination due to one-complement sum.

We can compute two more values of checksum, so an UDP probe can be identified with up to four sequence numbers.

In a varying destination analysis, the source port will always be the same, therefore we must compute Δ at most L times, where L is the exploration limit - we will use at most L different IP destination addresses.

In a fixed destination analysis, the source port will be different between vertical explorers. Using the delta mechanism would introduce a much higher number of safesends, since almost all port can be used at least once and there is no guarantee that the NAPT will maintain a binding for all the duration of the analysis, especially because the analysis will *hog* the NAPT pool. To achieve better results, if a NAPT is detected PMDA uses port coupling and destination port queues.

3.3.8 Analysis example

In this subsection I will describe the operations that a clients executes when running a traceroute analysis. Assume that the server gave a client

a tracerouter job to the address **G** with UDP and fixed destination; all the other parameters are set to their default. The tracerouter's main thread discovers its IP address and the default gateway's one, using a randomly created probe. It then starts an HE over the newly-found node, shown in red in figure 3.5. The newly-created HE carries the initial probe used to discover the default gateway.

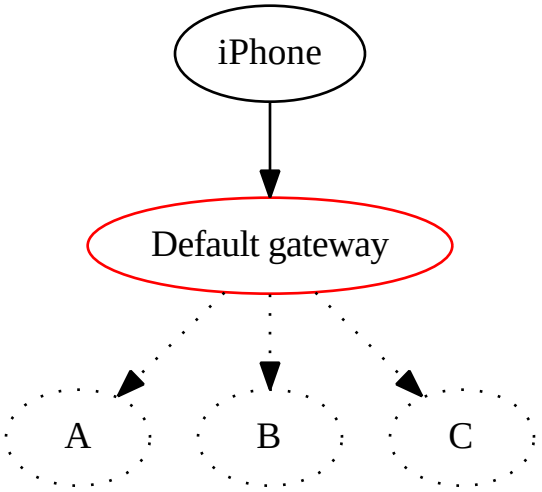


Figure 3.5: analysis example, first step

Let's suppose there are three nodes after the default gateway yet to be discovered. The HE launches a first batch of six VE, as in figure 3.6; the edge labels list which probe actually discovered which node. Each probe has a different flow ID, and is launched by a separate VE. Please note that, if the default gateway acts as a NAPT, we won't be able to send multiple probes with the same destination port at a time. If we were using a varying destination approach, the first transmission of each probe would have failed to evaluate Δ.

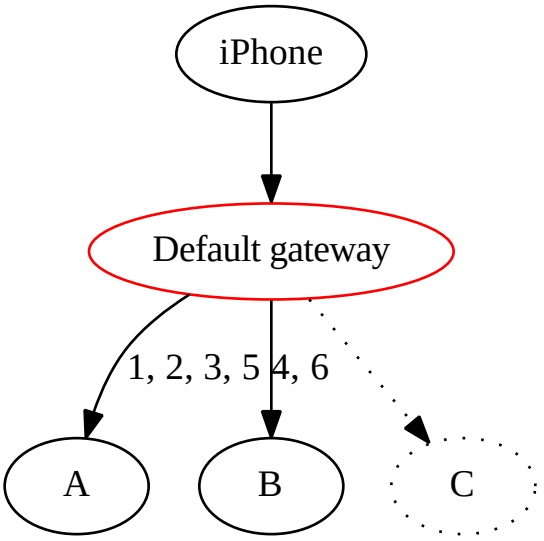


Figure 3.6: analysis example, second step

Nodes A and B were discovered, so MDA will spawn 12 other VEs. They will eventually reach C, as in figure 3.7.

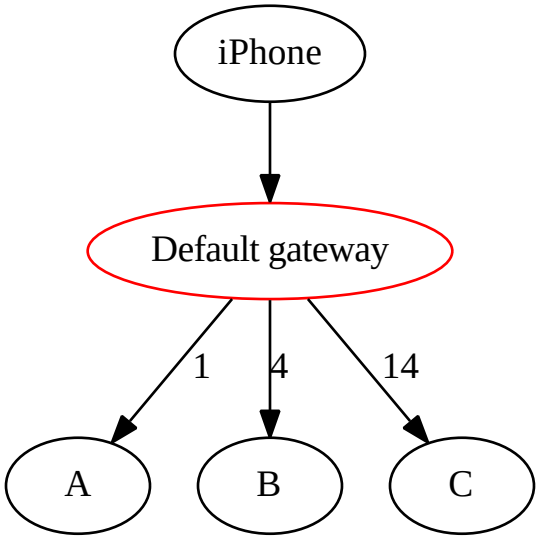


Figure 3.7: analysis example, second step completed

A total of 24 VE were launched for default gateway. Even if most probes reached already-known nodes to reach the 95% confidence level, their answers are still useful since they contribute to MIDAR DB. The HE will then filter the received data and create a new HE for each discovered node, as in figure 3.8:

- node A's HE will be created with path $iPhone \rightarrow gateway \rightarrow A$ with probe 1 (red);
- node B's HE will be created with path $iPhone \rightarrow gateway \rightarrow B$ with probe 4 (blue);
- node C's HE will be created with path $iPhone \rightarrow gateway \rightarrow C$ with probe 14 (green).

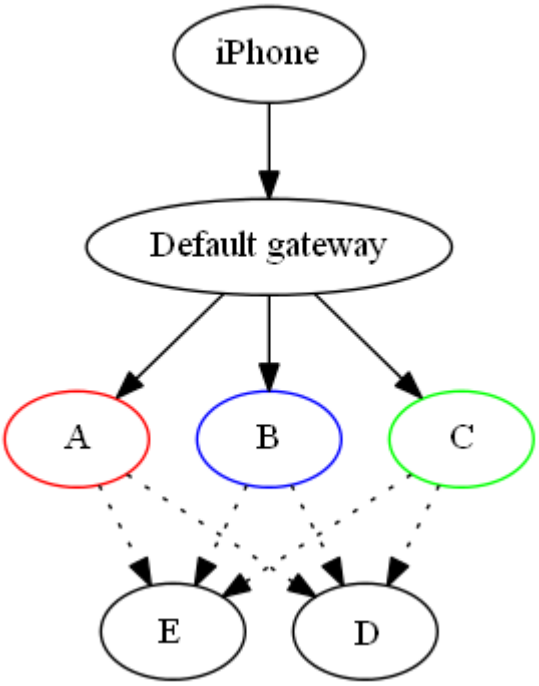


Figure 3.8: analysis example, third step

The three HE operate all at the same time: they will discover both D and E. Suppose B finishes before A and C; two new HE would be created,

since D and E are unknown nodes. Node D's HE will be created with path $iPhone \rightarrow gateway \rightarrow B \rightarrow D$, in purple; node D's HE will be created with path $iPhone \rightarrow gateway \rightarrow B \rightarrow E$, in blue. They both inherit a probe from B's HE.

In figure 3.9, there are four active HEs: A, C, D and E.

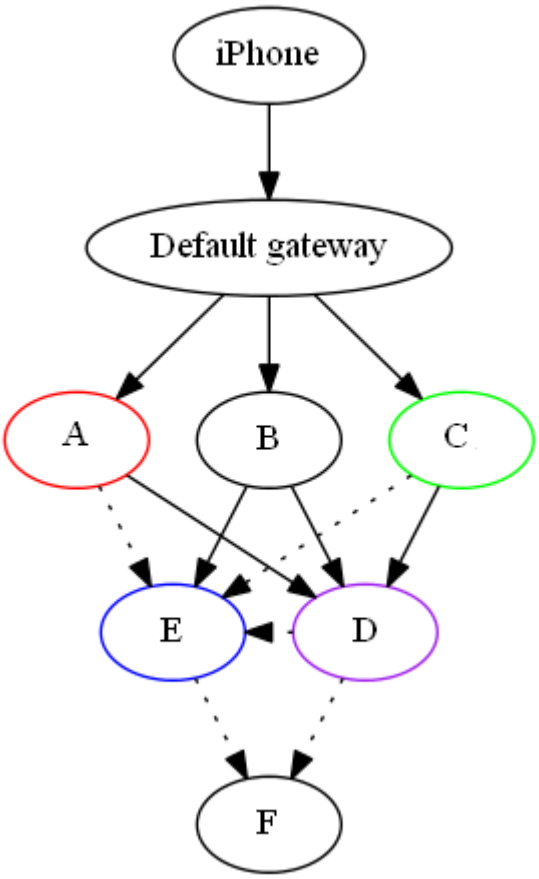


Figure 3.9: analysis example, fourth step

When A and C horizontal explorers finish, they won't start any additional explorer, since they have been already explored, or are being explored right now. One HE between E's and D's will finish before the other and will create the HE to F, which will lead us to the full inferred interface topology as in figure 3.10.

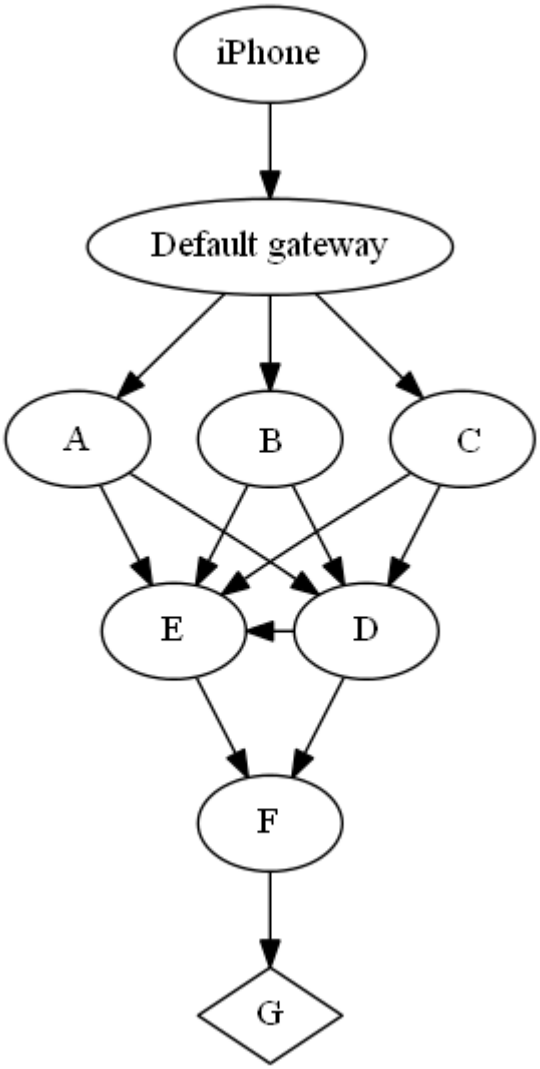


Figure 3.10: analysis example, full interface topology

MIDAR first builds the reachability matrix in table 3.1.

	iPhone	gateway	A	B	C	D	E	F	G
iPhone	0	1	1	1	1	1	1	1	1
gateway	0	0	1	1	1	1	1	1	1
A	0	0	**0**	**0**	**0**	1	1	1	1
B	0	0	**0**	**0**	**0**	1	1	1	1
C	0	0	**0**	**0**	**0**	1	1	1	1
D	0	0	0	0	0	0	1	1	1
E	0	0	0	0	0	0	0	1	1
F	0	0	0	0	0	0	0	0	1
G	0	0	0	0	0	0	0	0	0

Table 3.1: analysis example, reachability matrix

The **bold** values show that A, B and C are not connected. Suppose that for each of them there are 8 MIDAR answers[21]. MIDAR will run the monotonic test between A and B first, using *fictitious* data[22] from tables 3.2 and 3.3:

Time (relative) [ms]	RTT [ms]	IP ID
0	2	56
6	3	65
7	1	69
9	3	74
13	4	81
15	1	86
19	2	96
21	1	102

Table 3.2: MIDAR answers for A

[21]They actually are a lot more: each VE starting from one of them will first create a probe that should reach them.

[22]The values reported for Time and RTT are not real, but their distribution is.

Time (relative) [ms]	RTT [ms]	IP ID
2	3	59
6	1	67
8	2	68
10	1	73
14	2	84
16	3	88
22	2	89
23	3	106

Table 3.3: MIDAR answers for B

MIDAR monotonic test will then be run between A and B, A and C and then B and C. If a merge occurs between A and B, then it would run the test between (A, B) and C. If there is a merge between A and C MIDAR won't run a test with AC and B, since it already knows that A and B are not compatible.

With the time series provided A and B can and will be merged into the same router. I'll skip the following step of running the monotonic test between (A, B) and C, assuming that has been passed. The final topology is reported in figure 3.11.

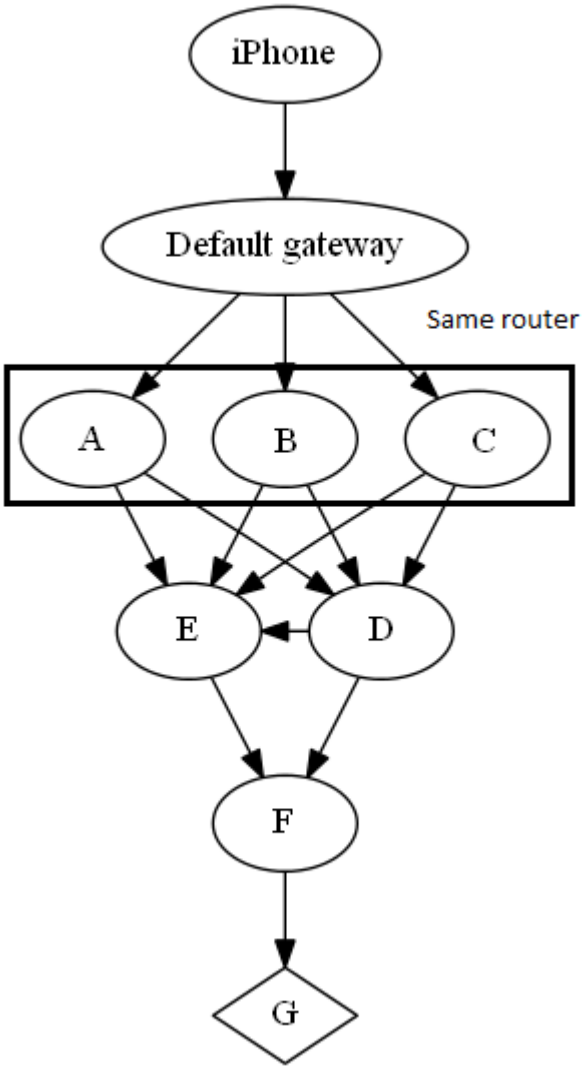


Figure 3.11: analysis example, topology after dealiasing

3.4 Client-server protocol

mYriadi client-server protocol implements the following features:

- clients service requests to the server;

- server notification that a client is available to run an analysis, and subsequently assignment of a job to such client;
- clients analyses results delivery to server;
- client identification.

This protocol has a modular structure, therefore enabling effortless future expansions to accommodate new analyses and operations.

Every implementation of this protocol must adhere to the following conventions:

- network order (big endian) must be enforced whenever applicable;
- the protocol is designed to run over TCP;
- a floating point value V_f is never transmitted as-is, instead it is always transmitted as a 32 bit integer V_i:

  $$V_i = \lfloor V_f \, 10^4 \rfloor$$

  In the following pages, this is called **decimal representation of a floating point number**.

In this document, **bytefields**, a particular kind of diagram, are used to show the protocol's data structure and ordering. Grey-colored padding boxes are used to improve the readability of such diagrams: they must be ignored when implementing the protocol.

3.4.1 Identifiers

The protocol uses four different identifiers:

CID Client ID; it identifies a particular client between many. A CID is a 4 byte unsigned integer.

OID Operation ID; it represents a particular operation that the client wants to do, or a particular service that the client would like to receive from the server. An OID is a 1 byte unsigned integer.

AID Analysis ID; it discriminates between all the possible analyses MapLibrary (the client-side library) supports. An AID is a 1 byte unsigned integer.

JID Job ID; it is an identifier linked to a particular job assigned to a client. A job is an instance of an analysis assigned to a particular client. The server generates a JID for each job assigned to every client. A JID is a 8 byte unsigned integer. JID namespace is shared between all the analysis IDs.

Table 3.4 summarizes their size and usage.

Id	Size (B)	Identified entities
CID	4	Clients
OID	1	Operations (service requests typologies)
AID	1	Analyses
JID	8	Job

Table 3.4: protocol identifiers

3.4.2 Establishing protocol sessions

A client must run a session establishment procedure with the server:

- The client sends its CID to the server and waits for approval.
- The server acks with a 1 byte unsigned integer that must be either 1, allowed, or 0, refused.

After session establishment, the client can send can communicate an operation to the server. An operation may or may not close the session; that is, if the client wish to communicate another operation it must run the aforementioned session establishment procedure.

3.4.3 Operations

Each operation represents a specific request of the client to the server; an operation may have one or more parameters. table 3.5 lists the operations defined.

OID	Operation
0	Refresh status
1	Ask for a job
2	Send results
3	Geo-locate an IPv4
4	Geo-locate an IPv6

Table 3.5: OID to Operation

Refresh status

A client performs this operation to refresh its status, sending to the server its location as parameters.

As in table 3.6, the client must send the OID 0, followed by the decimal representation of latitude and longitude, in this order.

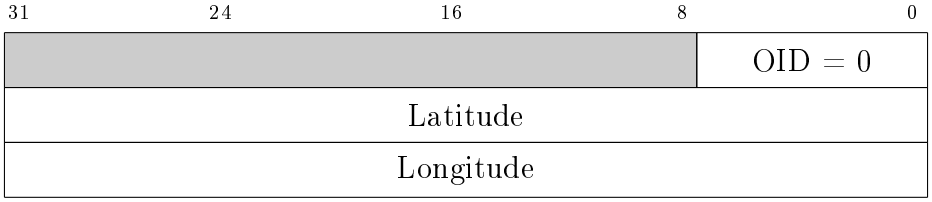


Table 3.6: Refresh status client message

There is no further interaction between client and server.

Ask for a job

With this operation, the client request a job to the server: the client performs this operation when it's ready to run an analysis. This operation requires two parameters, latitude and longitude. The client must send the OID 1, followed by the decimal representation of latitude and longitude, in this order.

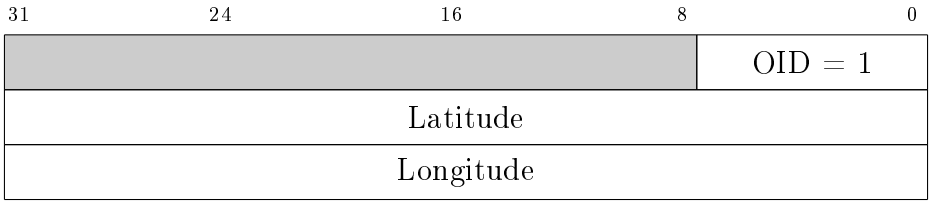


Table 3.7: Ask for a job client message

The server answers with an AID and, if applicable, a JID. Depending on the AID, the server might send additional data. If AID is set to 0, the client hasn't been assigned any job.

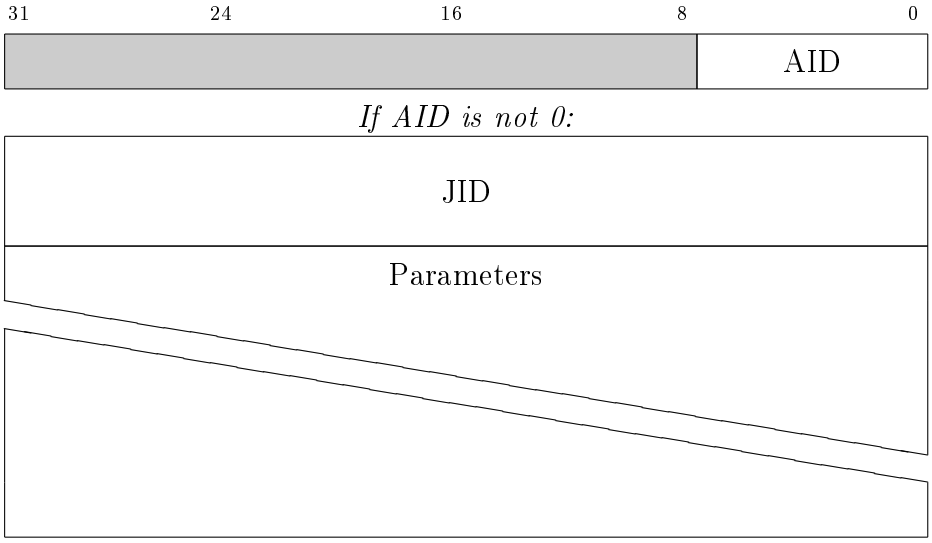


Table 3.8: Ask for a job server full message

Send results

A client sends the results of a job to the server; the job's JID is the unique parameter of this operation.

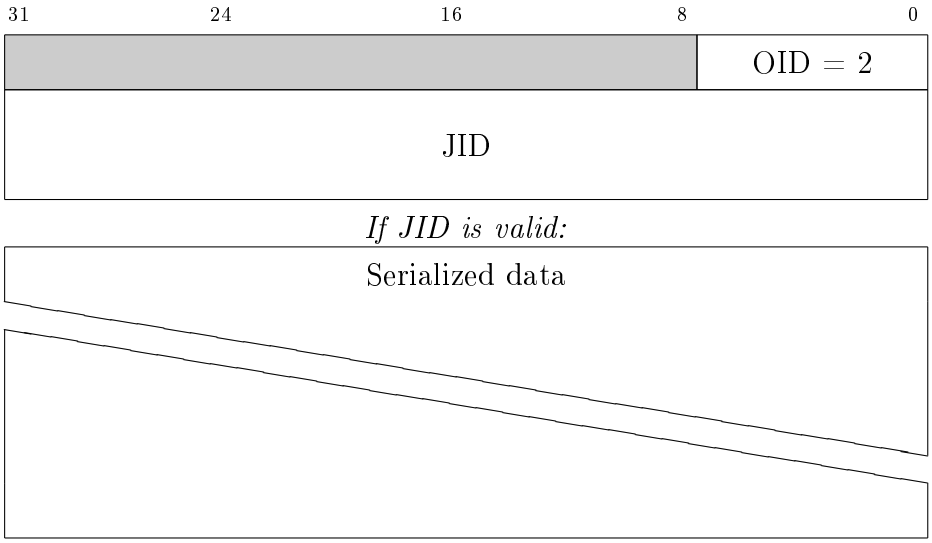


Table 3.9: Send results client full message

This operation is composed by the following steps:

1. the client sends the OID (2) and the JID;
2. the server validates the JID and will send a byte with the following semantic:

0	the JID is not valid;
1	the JID is valid;

3. if the JID is not valid, the client will close the connection and it won't try to send those result anymore;
4. if the JID is valid, the client sends the results to the server, with the data serialization format specified by the analysis type;
5. the server validates the data received, and then it sends back an *outcome* byte, according to table 3.10;
6. if the client won't receive such outcome byte, it may try to retransmit data again, at its own discretion, but in a new protocol session;

7. if the client receives the outcome byte, it will behave accordingly to table 3.10; if a retransmission is needed, it must be performed in a new protocol session.

Value	Description	Delete saved data
0	validation failure	yes
1	validation success	yes
2	retransmission required	no

Table 3.10: outcome byte values

Geo-locate an IPv4/v6 address

A client asks the server to geo-locate an IP address. This operation has one parameter, the IP to locate: depending on the OID, the IP protocol version used is v4 or v6. Please refer to table 3.5.

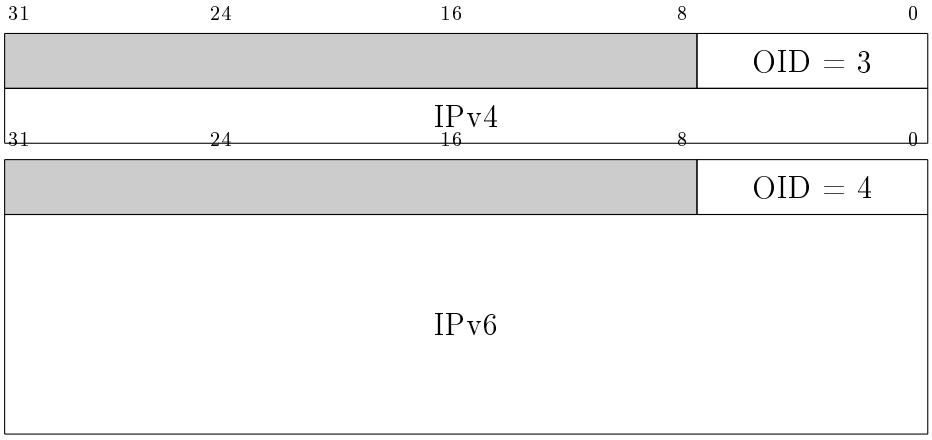


Table 3.11: Geo-location request client full message

The server sends back the decimal representation of latitude and longitude, in this order.

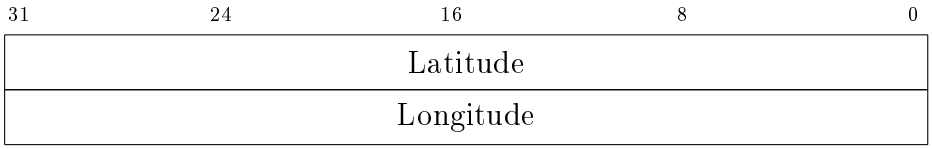


Table 3.12: Geo-location answer full message

Should the server fail to geolocate such address, it will send the maximum value for a 4 byte integer, instead of the coordinates. To handle this outcome, the client must first read the latitude:if its value is out of range[23] the client infers that the server couldn't locate the requested address.

3.4.4 Analyses

Table 3.13 lists the jobs defined in this platform version.

AID	Analysis
0	No analysis
1	Traceroute

Table 3.13: AID to Analysis

Traceroute

This analysis, identified by AID 1, consists of a traceroute toward a target IP address; the traceroute will stop if it finds an IP in an exclusion list. A number of parameters might be forced by the server to the client. To be as flexible as possible, parameters are sent using a TLV[24] blueprint. The server must send the size in byte of the parameters, in a 2 byte unsigned integer.

[23] Latitude is constrained to 90 degrees, therefore $latitude \leq 90\,10^4$.
[24] Type Length Value.

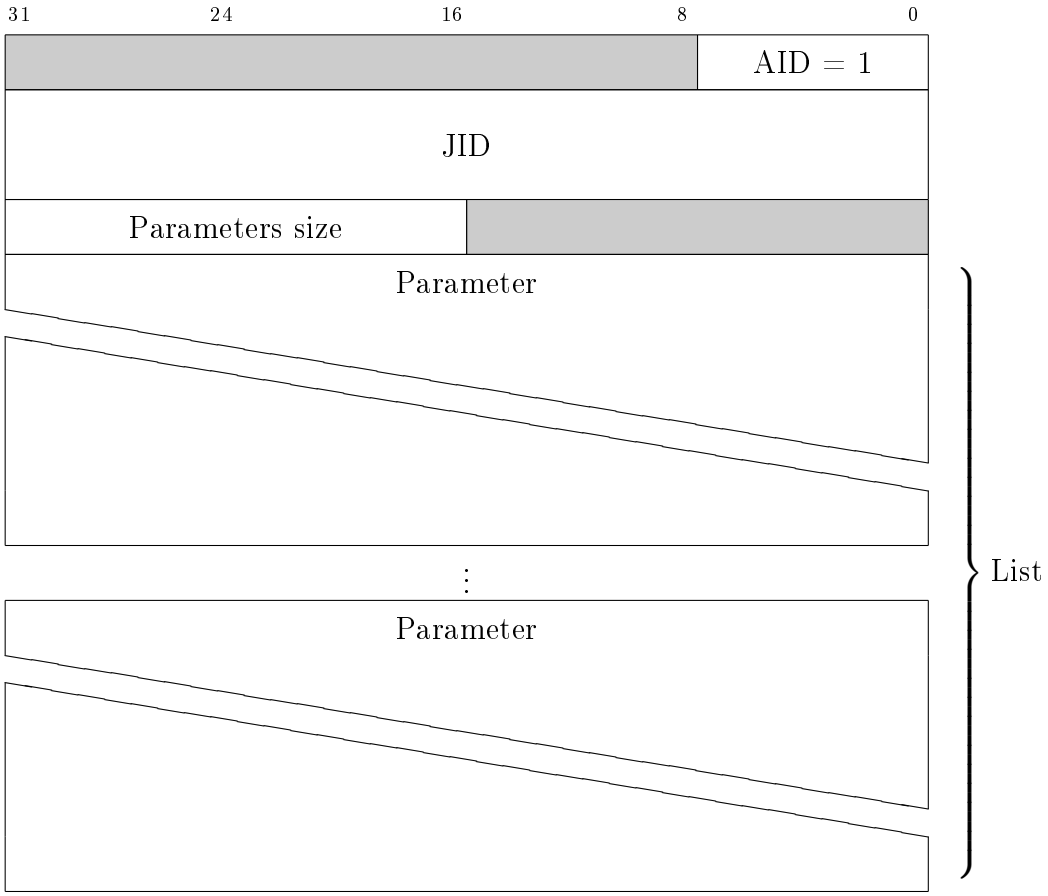


Table 3.14: Traceroute job assignment

A parameter is characterized, in order, by:

1. a type code, encoded in an unsigned byte;
2. data length, encoded in an unsigned byte;
3. the data itself.

Some parameters are optional, whilst others are mandatory:

- if the client doesn't find an optional parameter, it will default to a certain value or behavior;

- if the client doesn't find a mandatory parameter, it will refuse the job.

The server can send the parameters in whatever order it prefers. The server may send more than one instance per parameter type; unless specified, the client will always ignore any repetition of a parameter type. This might be useful for future expansions.

The following is a list of all supported parameters:

Target [Type 0, mandatory] specifies the target IP address; length can be either 4 or 16, in order to support both IPv4 and IPv6.

Probe type [Type 1] specifies the probe type: the first byte in the payload selects the protocol used.

ICMP default, identified by 0, doesn't need additional parameters - length must be 1.

UDP identified by 1, needs the client's IP address; length must be either 5 (IPv4) or 17 (IPv6).

Exclusion list [Type 2] contains a list of IP address that, if encountered, will trigger the stop of the analysis; the number of addresses can be recovered by dividing the length field value with the size of an IP address in byte. If missing, the list is considered empty.

Max TTL [Type 3] specified the maximum TTL value that can be used by the client, as an unsigned byte; length is always 1.

Exploration mode [Type 4] describes the behavior of the client when exploring nodes. The following fields are sent:

Mode [1 unsigned byte] if 0 the client won't modify the destination address. Default is 1.

Exploration limit [1 unsigned byte] the maximum number of probes that can be sent to discover the next hop of a node. Default is 96.

Exploration increment [1 unsigned byte] the number of vertical exploration that will be performed on a node the first time it's discovered, or after a new outgoing interface is discovered. Default is 6.

The client will send the outcome even if the traceroute stopped for any reason: the server will then accept or reject the outcome as defined.

Graph serialization The topology is represented as a list of edges between nodes. A node is identified by a 2 byte integer alias: if this alias is negative, the node is *fake*, otherwise is *real*. A fake node is a node indirectly detected by the traceroute analysis: its existence is inferred from the network behavior.

A **generic edge** is composed of:

- source and destination *real* node aliases;
- the source and destination IP addresses;
- the delay of the link;
- the TTL skip count: if h is the TTL value that reaches the source node, and k is the TTL value that reaches the destination node, then the TTL skip value t is:

  $$t = k - h - 1$$

  this value represents the hypothetical number of *hidden* nodes that are between the source and the destination that the traceroute failed to detect explicitly.

A **1-hop edge** is composed of:

- source and destination node aliases, without restrictions on *fakeness*;
- if the source alias is real:
  - the source IP address;
- if the destination alias is real:
  - the destination IP addresses;
- in addition, if both aliases are real:

– the delay of the link.

The TTL skip count is absent, since it's always assumed to be 0. Therefore, a 1-hop edge has no hidden nodes in between. table 3.15 shows the data format of a 1-hop edge - clients must adhere to the field order specified.

31 … 24	16 … 8	0	
Source alias	Destination alias		
Destination IP			Opt.
Source IP			
Delay			

Table 3.15: 1-hop edge serialization with IPv4

Before client-side serialization, the graph has generic edges only. It is free of *fake* nodes, but there might be *hidden* nodes, implied by non-zero TTL skip count occurrences. The data representation of the graph must be expressed as a list of serialized 1-hop edges: the client must transform the graph representation in order to use only 1-hop edges. For each generic edge from A to B, from IP X to IP Y, with delay D and TTL skip count T, the client will create T fake nodes, and will create $T - 1$ **1-hop edges** that will connect A with B, passing through each fake node once.

The client will send the result of the analysis as in table 3.16, keeping in mind that 1-hop edges have variable size:

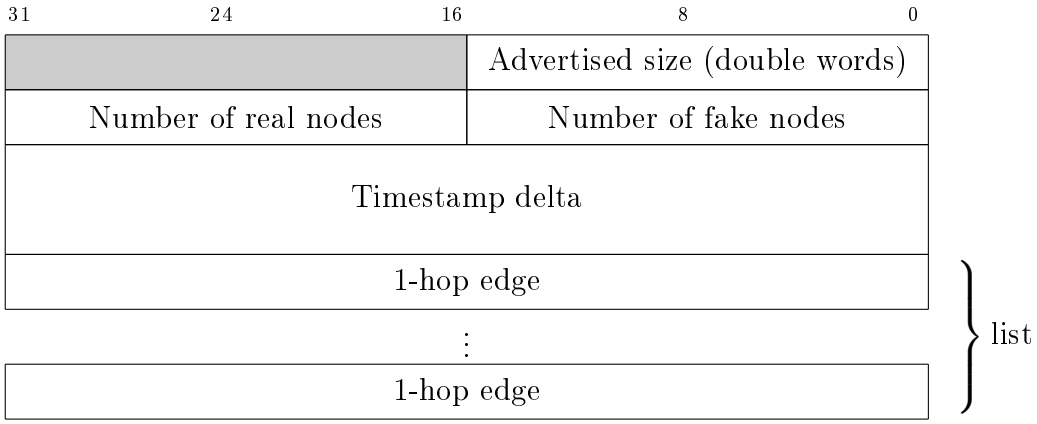


Table 3.16: Traceroute results structure

Assuming that the client finished the analysis at T_F, the client creates the timestamp delta ΔT from the timestamp when it received the job T_S:

$$\Delta T = T_F - T_S$$

The client sends the advertised size S_a in double words instead of the number of edges. This has the following consequences:

- since edges have different size depending on their content, the server doesn't have to parse the data in order to decide how many bytes to read from the TCP stream;
- the number of edges in the results cannot be identified without reading the payload;
- the client sends the number of real and fake nodes, as two byte unsigned integers, right after the advertised size, allowing the server to perform an early validation.

Chapter 4

Validation

> "A fool is a man who never tried an experiment in his life."
>
> Erasmus Darwin

To assure that the whole platform performs correctly, we executed a general test that interests the whole system. Our goal was to reconstruct a map of the GARR network. GARR is the Italian research network, which has an high quality documentation and freely provides its *ground truth*, a map that describes correctly the network's topology. This experiment, described in section 4.1, verifies both the server and the client appliances: it relies both on the client's ability to run a traceroute analysis and on the correct server-side logic that integrates all the topologies into one graph.

A set of four analyses is provided in 4.2; they use different MDA modes and their results are compared one to another to show the difference between them.

In addition, a custom test validates the NAPT bypass method; this experiment is described in section 4.3.

4.1 GARR network

The GARR network (http://www.garr.it/) offers access to the GARR Integrated Networking Suite, also known as GINS. GINS provides various information and statistics. The backbone weathermap describes in realtime the usage of backbone links between the various PoPs, Points of Presence. I did not expect measurement anomalies during the analyses and, in fact, none were found. The UDP protocol was used, altough ICMP would've been fine, too, since GARR routers behave correctly and respond to ICMP probes. Figure 4.1 shows the backbone structure of the network.

The experiment was run in mid May 2012 from Pisa[1]. The device used was connected to the IET department network, since it is attached to the GARR network via the PI1 PoP. I handpicked 53 targets, all located at the edges of the network. In this way I ran each analysis from an access network to another access network, as far as possible. The intermediary nodes and links will define the backbone when the server will merge each graph together.

[1] A first experiment was run in mid December 2011 at Pisa, from the Serra WiFi network, with very good results, since it managed to describe the GARR backbone correctly. The GARR network, however, experienced a lot of updates in the following six months, so I decided to re-run the experiment to have a more up-to-date map.

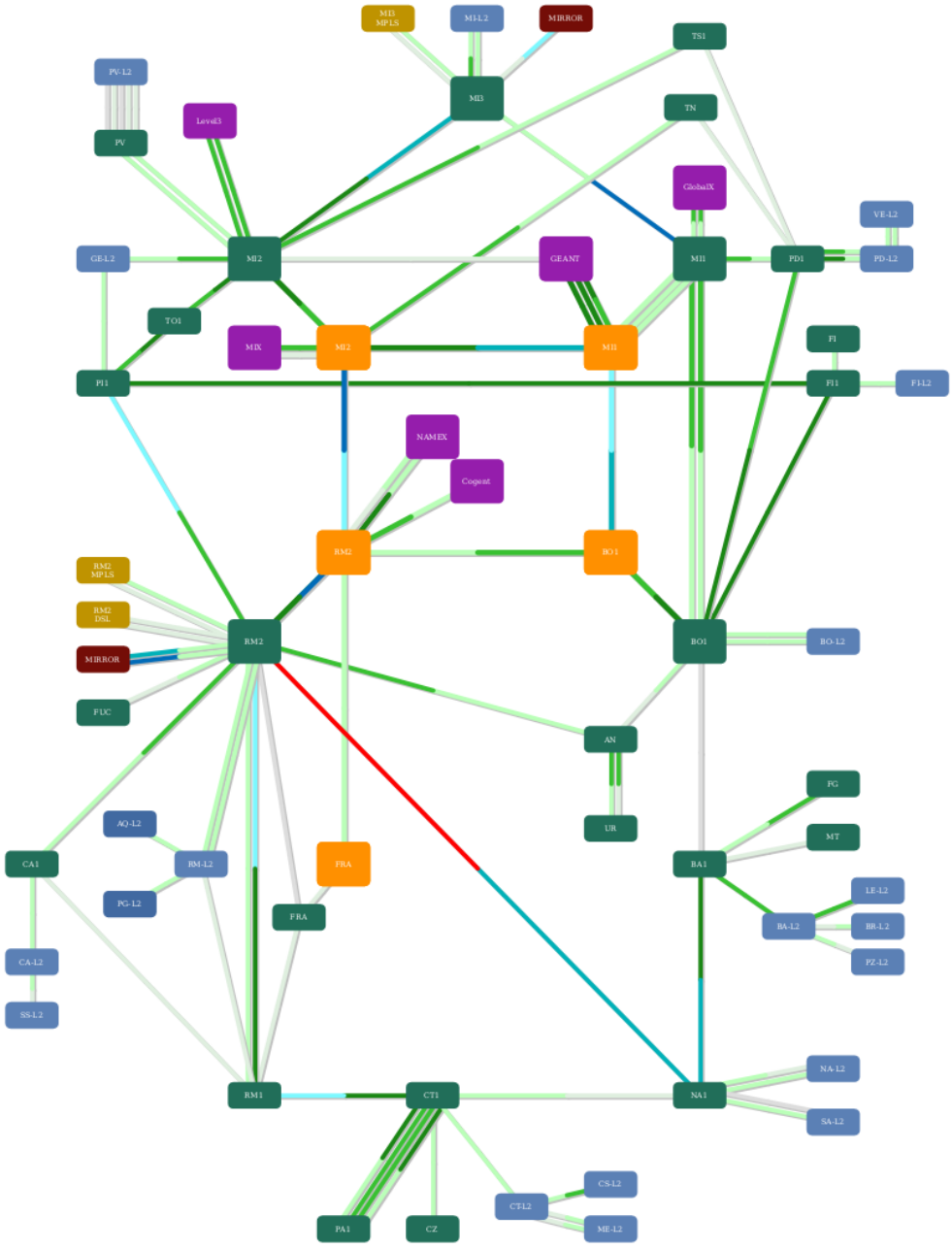


Figure 4.1: GARR network backbone weathermap via GINS

The following list contains the campaign's targets:

PoP	Domain name or description	Targeted IP address
CT1	rt-ct1-ru-unirc.ct1.garr.net	193.206.137.166
	rt-ct1-ru-infngrid.ct1.garr.net	193.206.137.186
	rt-ct1-ru-irccs-neurolesi-me.ct1.garr.net	193.206.137.182
TS1	rc-ts1-ru-cnrismar.ts1.garr.net	193.206.132.10
	rc-ts1-ru-units.ts1.garr.net	193.206.132.26
TN	rc1-tn-ru-unitn.tn.garr.net	193.206.143.98
	rc-tn-ru-infntn.tn.garr.net	193.206.143.94
PI1	rt-pi1-ru-iit-ge.pi1.garr.net	193.206.132.70
PV	rc-pv-ru-cnao-pv-bk.pv.garr.net	193.204.217.86
	rc-pv-ru-irccsmaug.pv.garr.net	193.206.142.178
	rc-pv-ru-unipv.pv.garr.net	193.206.129.50
MI3	rt-mi3-ru-abami.mi3.garr.net	193.206.129.26
	Archivio di Stato - Napoli	212.189.246.14
	IRCCS FBF Brescia	212.189.242.178
PD1	rt-pd1-ru-uniud-l1.pd1.garr.net	193.204.218.106
	rt-pd1-ru-cnrpd.pd1.garr.net	193.206.132.194
	rt-pd1-ru-corila.pd1.garr.net	193.206.140.146
BO1	ru-ababo-rt1-bo1.bo1.garr.net	193.206.128.78
	ns2.garr.net	193.206.141.41
	rt1-bo1-ru-lhcopn.bo1.garr.net	193.206.128.30
NA1	rt-na1-ru-abana.na1.garr.net	193.206.130.58
	rt-na1-ru-eneaportici-l1.na1.garr.net	193.204.218.130
	rt-na1-ru-infnsa.na1.garr.net	193.206.143.130
AN	rc-an-ru-cnran.an.garr.net	193.204.217.218
	rc-an-ru-itis.an.garr.net	193.206.140.110
UR	rc-an-rc-ur.ur.garr.net	193.206.134.238
	rc-an-rc-ur-l2.ur.garr.net	193.206.134.178

PoP	Domain name or description	Targeted IP address
AN	rc-an-ru-cnran.an.garr.net	193.204.217.218
	rc-an-ru-itis.an.garr.net	193.206.140.110
UR	rc-an-rc-ur.ur.garr.net	193.206.134.238
	rc-an-rc-ur-l2.ur.garr.net	193.206.134.178
CA	rc-ca1-ru-cybersarmons.ca1.garr.net	193.206.137.38
	rc-ca1-ru-uniss.ca1.garr.net	193.206.140.78
FUC	rc-fuc-ru-asifucino.fuc.garr.net	193.206.131.134
BA1	rt-ba1-ru-cnrba.ba1.garr.net	193.206.142.90
CZ	rc-cz-ru-uniczmg.cz.garr.net	193.206.142.246
MT	rc-mt-ru-asimt.mt.garr.net	193.206.137.122
	rc-mt-ru-emsamt.mt.garr.net	193.206.137.106
MI1	lhcopn-cnaf.cern.ch	192.16.166.18
PA1	rc-pa1-ru-abapa.pa1.garr.net	193.206.137.242
	rc-pa1-ru-cnr-iamc.pa1.garr.net	193.204.218.58
	rc-pa1-ru-unipa.pa1.garr.net	193.206.137.210
SS	rc-ss-ru-sarss.ss.garr.net	193.206.140.98
VE	rc-ve-ru-cnrve.ve.garr.net	193.206.140.154
UR	rc-an-rc-ur.ur.garr.net	193.206.134.238
	rc-an-rc-ur-l2.ur.garr.net	193.206.134.178
FG	rt-ba1-ru-izs-foggia.ba1.garr.net	193.206.142.106
	rc-fg-ru-unifg.fg.garr.net	193.206.143.214
FRA	re1-fra-ru-enea-frascati.fra.garr.net	193.206.136.54
	re2-fra-ru-kloe.fra.garr.net	193.206.136.198
	re1-fra-ru-lnf.fra.garr.net	193.206.136.206
Level3	www.bbc.co.uk	212.58.244.69
Cogent	www.kernel.org	149.20.4.69
NAMEX	www.telecom.it	62.149.130.234
MIX	www.clubnautilus.it	212.35.204.132
VSIX	VSIX peering link	95.140.128.11
GEANT	lhcopn-cnaf.cern.ch	192.16.166.18

Table 4.1: targets used in this campaigns and, where available, their domain name

The server used a static list to assign targets. Only one device was used (and allowed to receive jobs) during the experiment. The device used a

retry time of 1.5 seconds and fixed destination MDA mode. I didn't use varying destination mode since it's based on the assumption that two nodes are *topologically close* if their IP addresses are close too; this assumption does not hold in the GARR network. Since the goal of this experiment is to reconstruct the GARR backbone, a maximum TTL value of 11 has been used (with the exception of the analysis toward *lhcopn-cnaf.cern.ch,* which has been cut as soon as reached the CERN network. The server did not specify other parameters, so the default settings were used. A full job cycle took approximately 15 seconds, including communication[2].

[2]The server is multithreaded: since network communications are terminated as soon as possible, the client will reconnect to the server as soon as possible; meanwhile, the server will evaluate the received data.

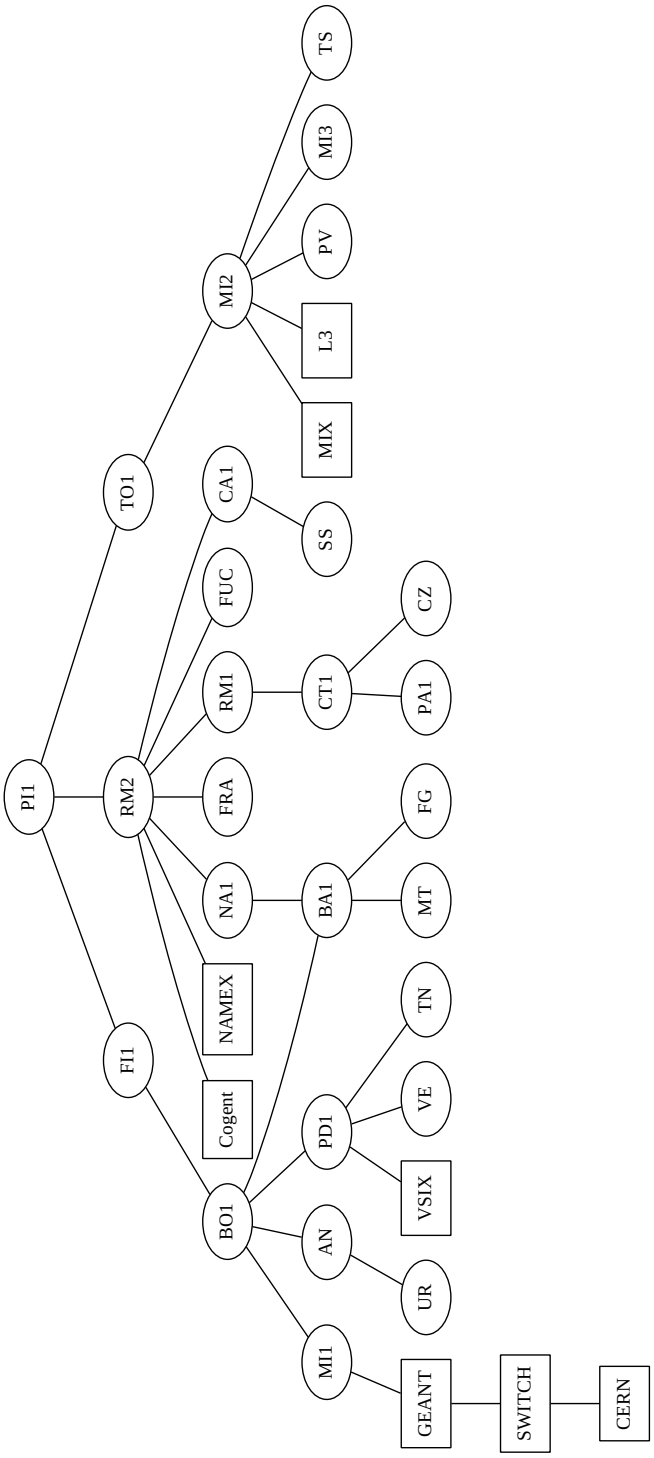


Figure 4.2: PoP level

The data validation is straightforward: the PoP-level map in figure 4.2 represents correctly the backbone. The client succeeded in revealing all the devices, and the server merged correctly each cluster of data[3]. Figures 4.3, 4.4, 4.5 and 4.6 shows the router-level map divided in four areas:

PI figure 4.3; Pisa, Firenze and Torino.

BO figure 4.4; Bologna, Ancora, Urbino, Padova, Venezia, Trento, Bari, Foggia and Matera, along with VISX.

MI figure 4.5; Milano, Pavia and Trieste, along with MIX, L3, GEANT, SWITCH and CERN.

RM figure 4.6; Roma, Frascati, Fucino, Napoli, Cagliari, Sassari, Catania, Palermo, Catanzaro, along with NAMEX and Cogent.

[3]The server has a quarantine mechanism that isolates two instances of a map location, discovered in two different analyses, that describe different situations although they both refer to the same network subset. These two instances are in quarantine until one of them reaches an acceptable confidence level. During this campaign no conflicts were found.

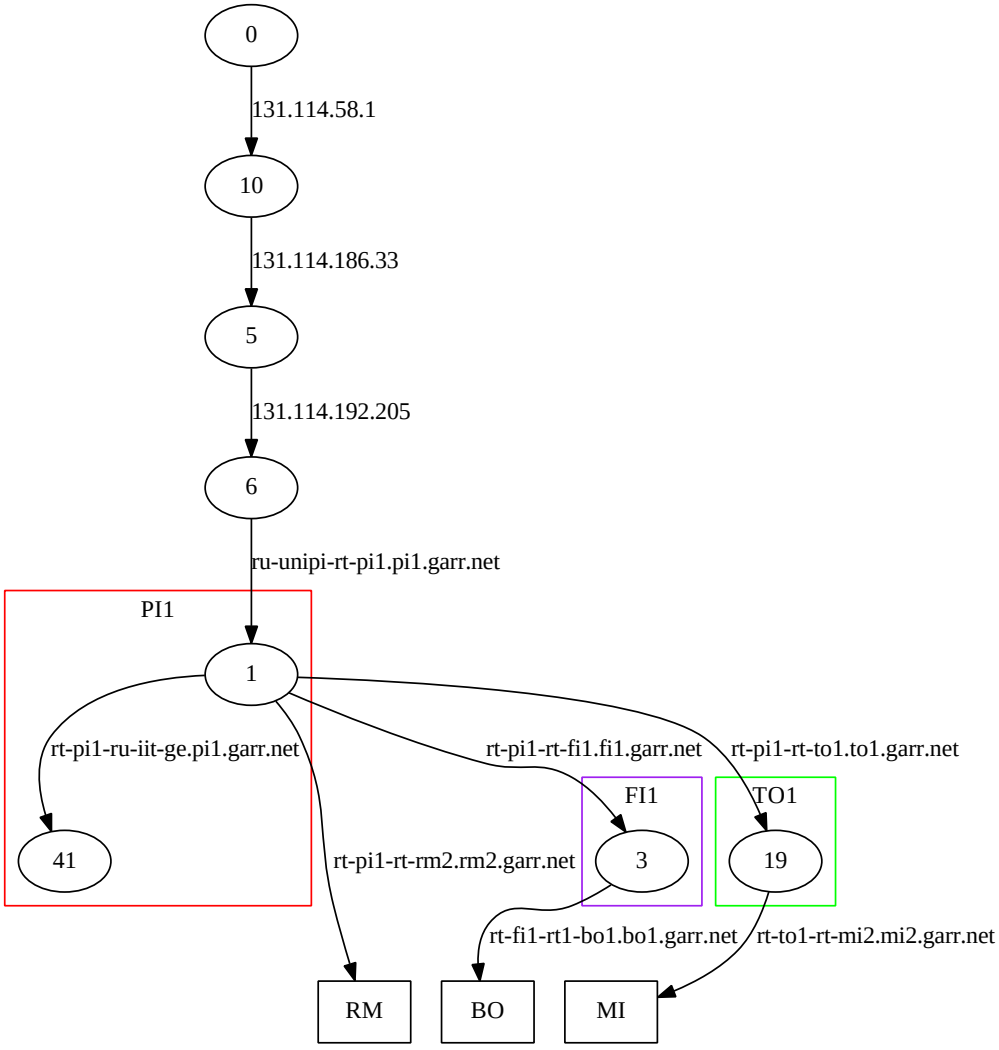


Figure 4.3: GARR network, PI area

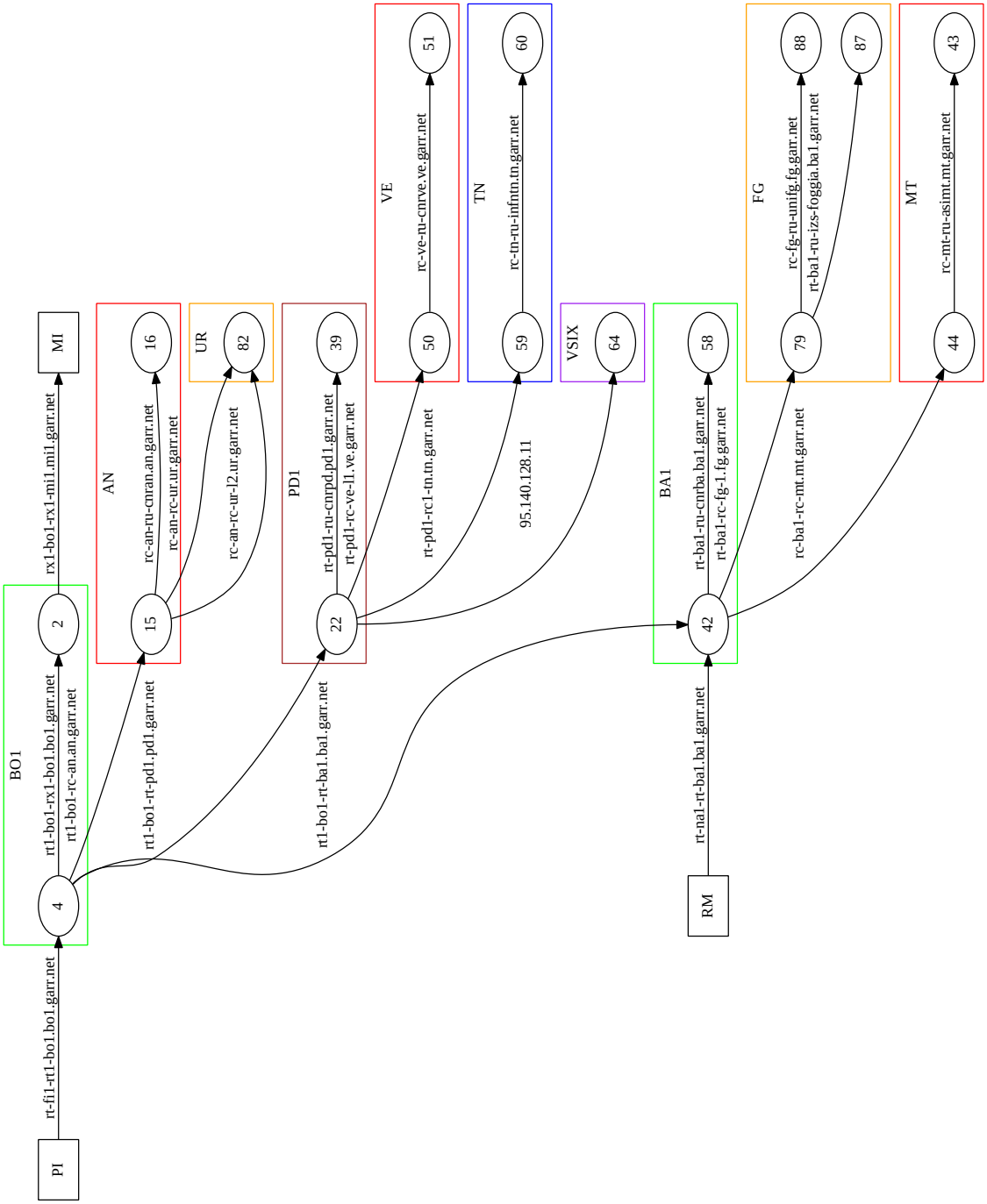


Figure 4.4: GARR network, BO area

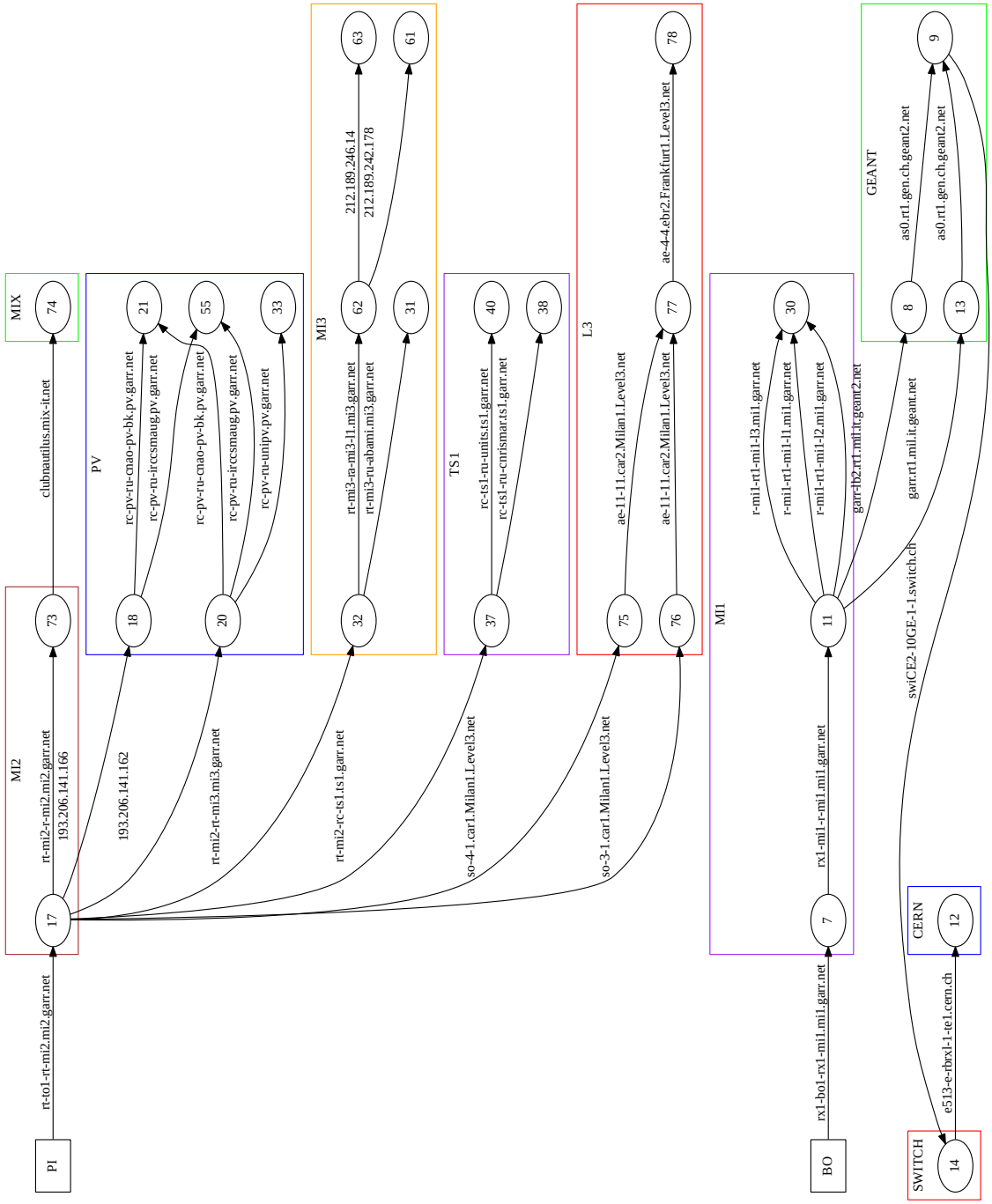


Figure 4.5: GARR network, MI area

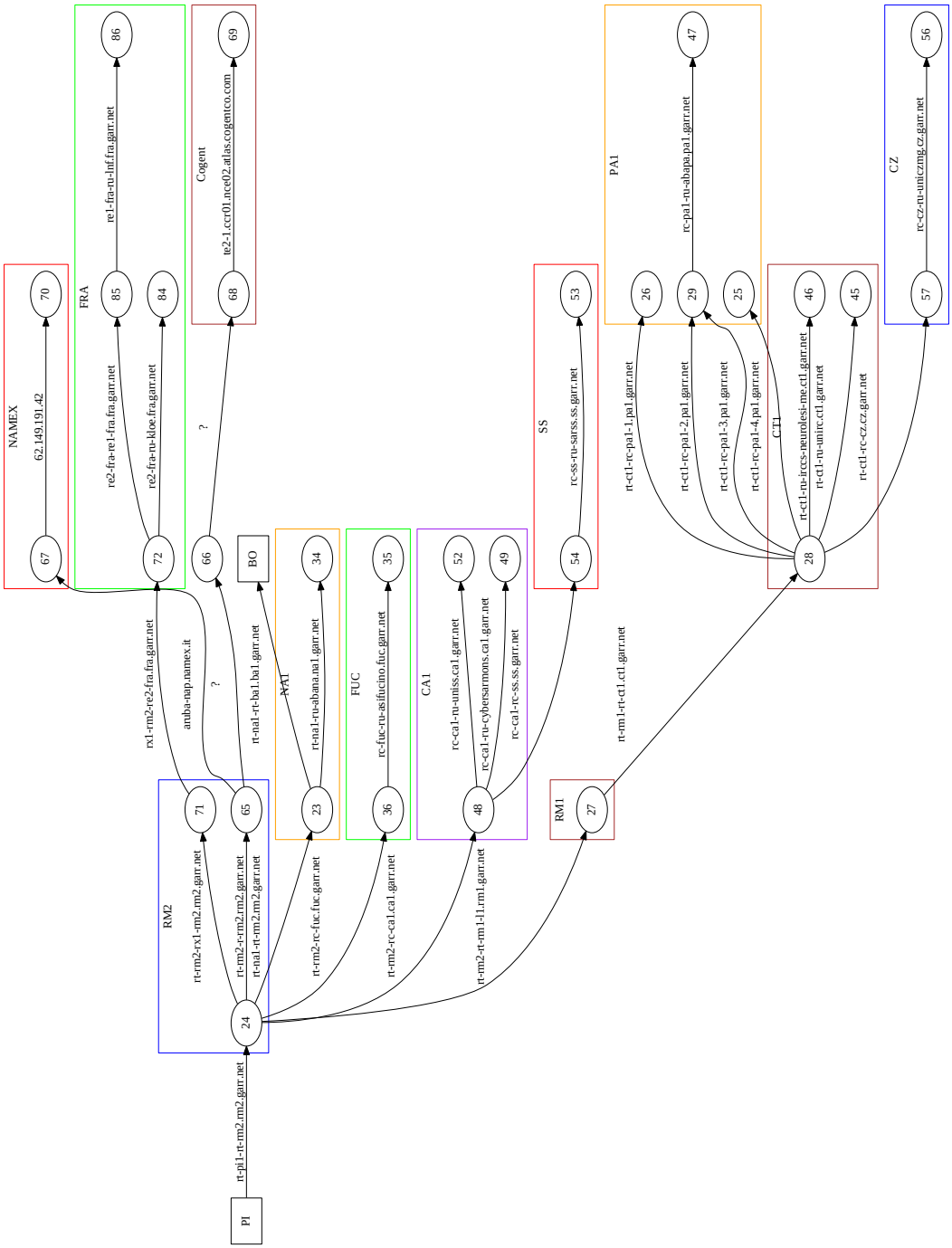


Figure 4.6: GARR network, RM area

A few considerations:

- BA1 (node 42) is usually reached through FI1 and BO1 (nodes 3 and 4, figure 4.4). A temporary traffic spike changed OSPF costs (which are visible in the live weathermap) so that the path with RM2 and NA1 (nodes 24 and 23, figure 4.6) was the best route. I exploited this situation to collect more data, that would've been hidden otherwise.
- Nodes 25, 26 and 29 (in PA1, see figure 4.6) are actually the same router, but the dealiasing process failed to group them; repeating the analysis multiple times always showed different configurations, so that the server will eventually merge them all each time a new analylis bring new dealiasing information.
- The two consecutive links between 65, 66 and 68 (in order, see figure 4.6) do not have a known IP address. The TTL skipping mechanism reached node 68 with a +2 bonus (the maximum allowed in this experiment), indicating that there are two non-responding routers. For this reason I was unable to insert node 66 in neither Cogent network nor RM2 PoP groups. Note that, however, if further analysis should not find these unknown nodes, the server will update its graph removing them, or replacing with a new node.
- Node 18 and 20 (in PV, see figure 4.5) are actually the same router, but the dealiasing process failed to group them. As per PA1, repeating the analysis multiple times yielded different groupings.

Thanks to parallel MDA we were able to correctly map the GARR network and its PoPs interconnections in less than 20 minutes with only one device. mYriadi client technology is fast and efficient as this experiment demonstrated.

4.2 Examples of analyses with different MDA modes

This chapter has the sole purpose of showing the differences between ICMP fixed destination (4.8), ICMP varying destination (4.7), UDP fixed destination (4.9) and UDP varying destination (4.10).

These tests were executed from a home network, where dsldevice.lan is the NAPT router, to the same GARR node: *rt-mi2-ru-infnge.mi2.garr.net*. The router is connected via ADSL to Tiscali, an Italian ISP. Be aware that these analysis are based on an unknown network, and that their only scope is to provide a basis to compare each mode. Nevertheless, the same network structure can be recognized in every scenario, although with very different levels of detail.

Figure 4.7: ICMP analysis, varying destination

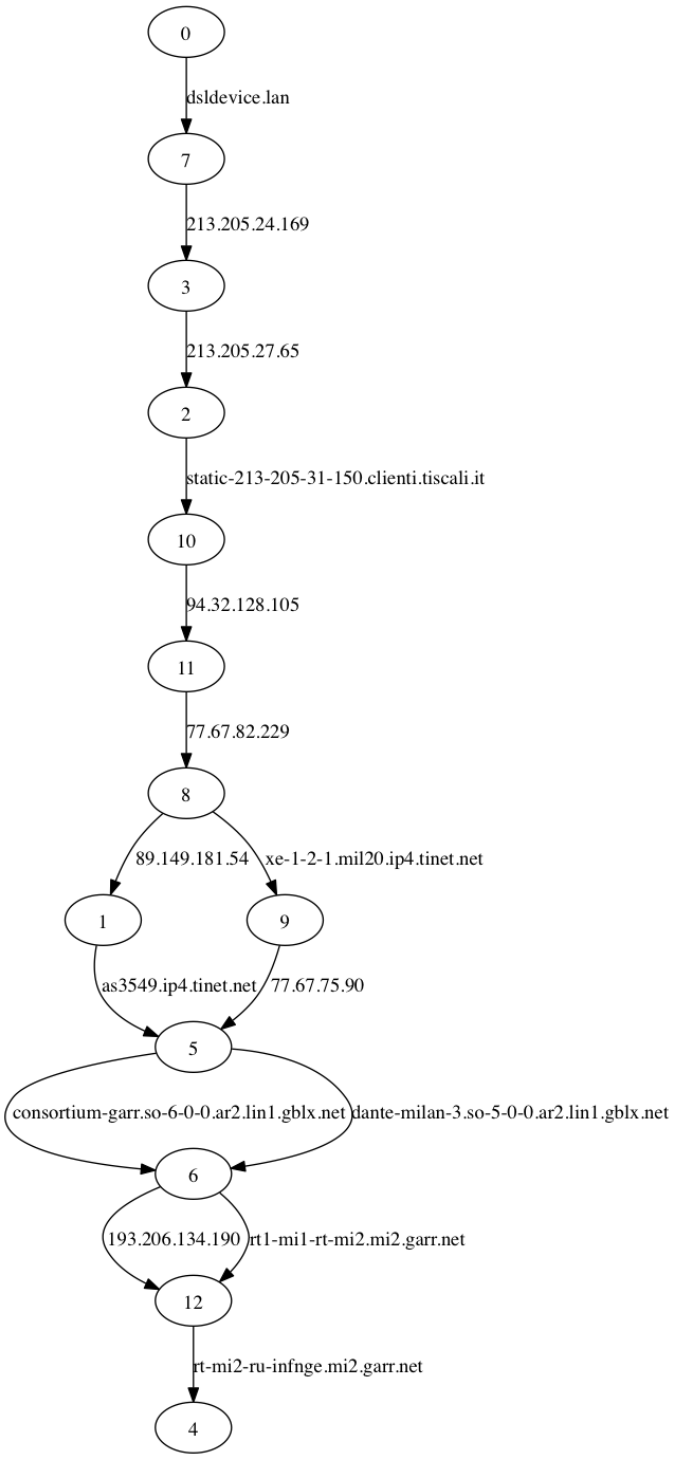


Figure 4.8: ICMP analysis, fixed destination

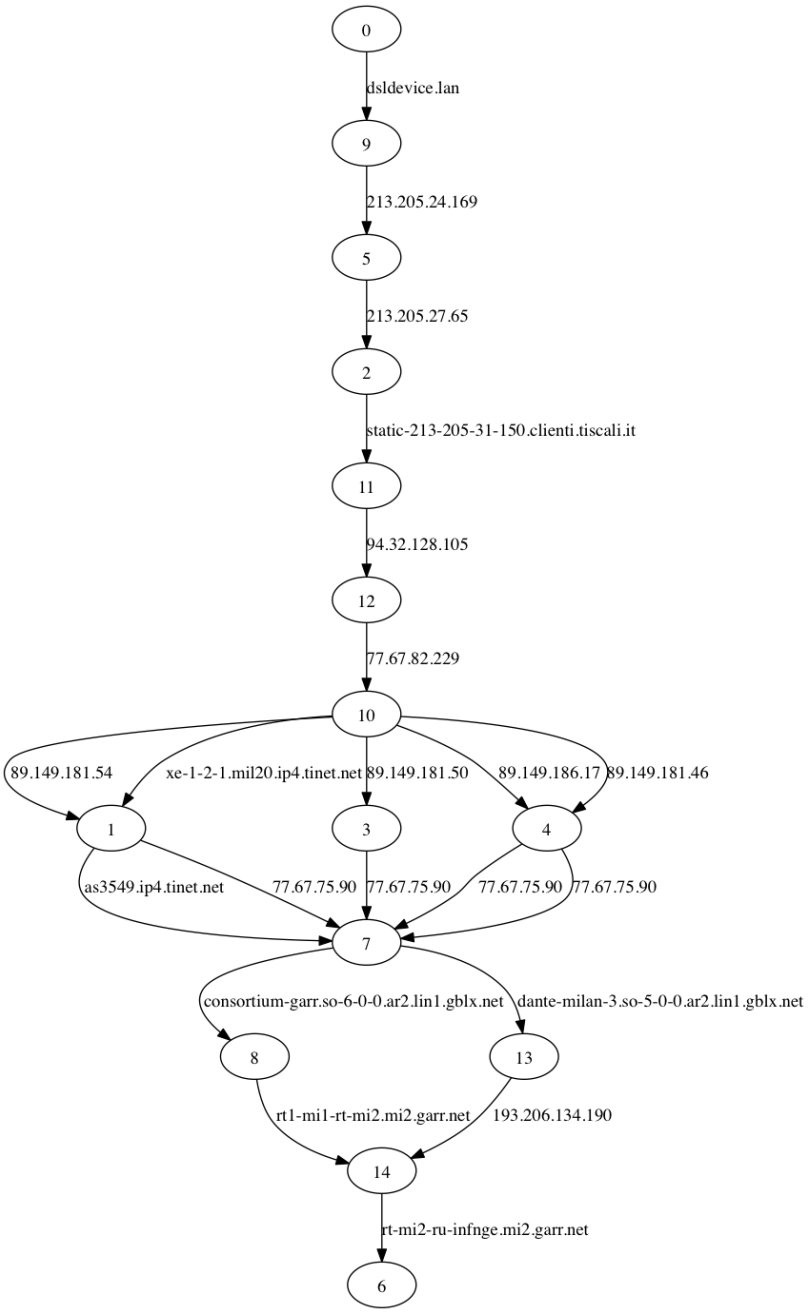


Figure 4.9: UDP analysis, fixed destination

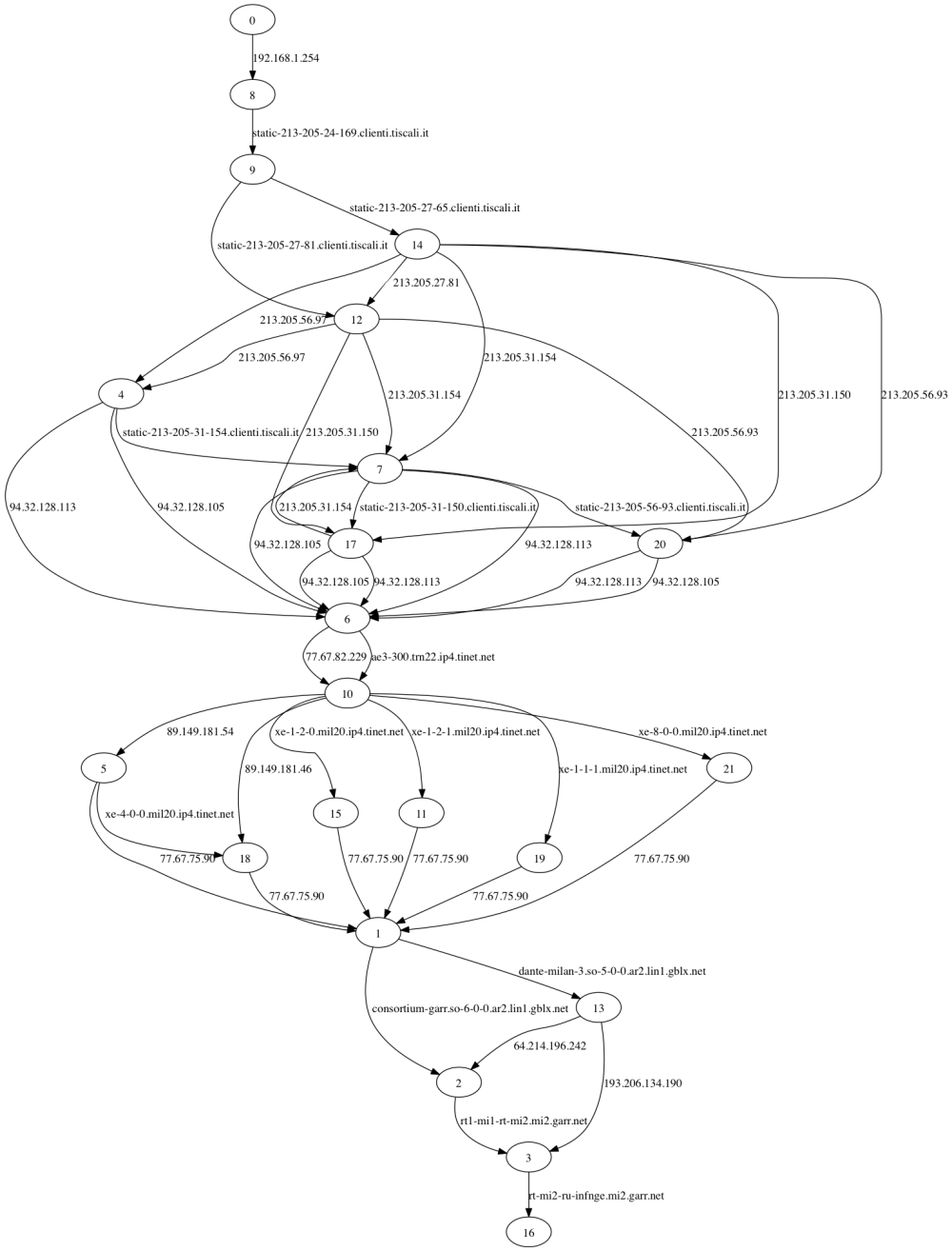
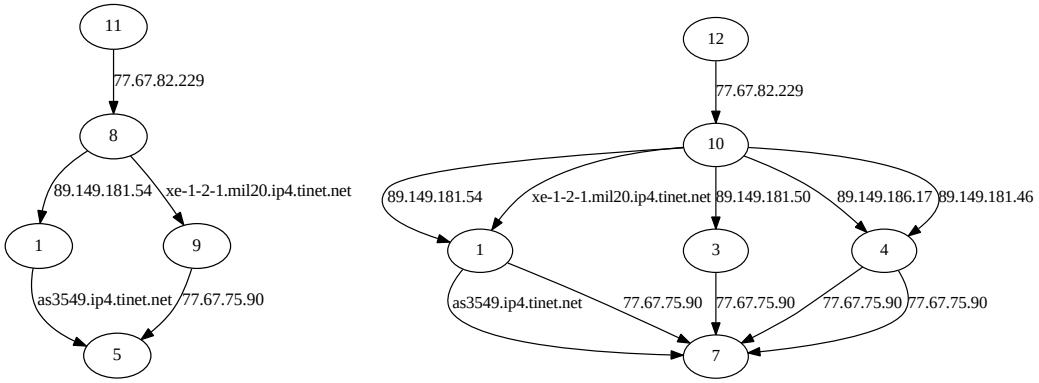


Figure 4.10: UDP analysis, varying destination

Let's start by comparing ICMP to UDP. In Chapter 2 I wrote that an ICMP traceroute would discover less node than a UDP traceroute. This statement holds true also for these analyses: it's likely that many ISP's routers do not respond to ICMP packets. Let's consider figure 4.8 and figure 4.9, which differs in the second half of the graphs. The first different block is showed in figure4.12, where UDP detected more nodes. In addition, ICMP did not merge routers 1 and 9 together, but UDP did. Since most of the new routers in the UDP block (subfigure 4.11b) reach node 7 with the same IP address, it's likely that they are the same router.

In addition, routers 1, 3 and 4 are actually the same router, as a standalone MIDAR execution would show. This is a minor error and it can be ignored since:

- multiple analyses from multiple client will eventually converge to the correct topology;
- the server keeps an AS map, completely hiding these details.

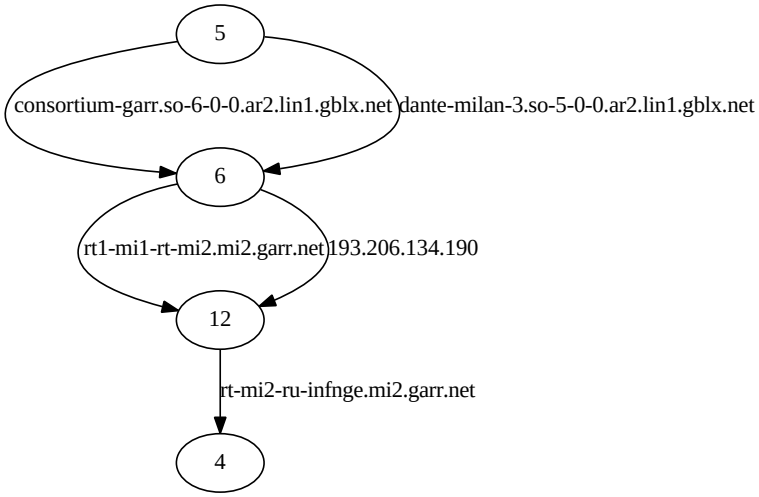


(a) first different block in the ICMP analysis

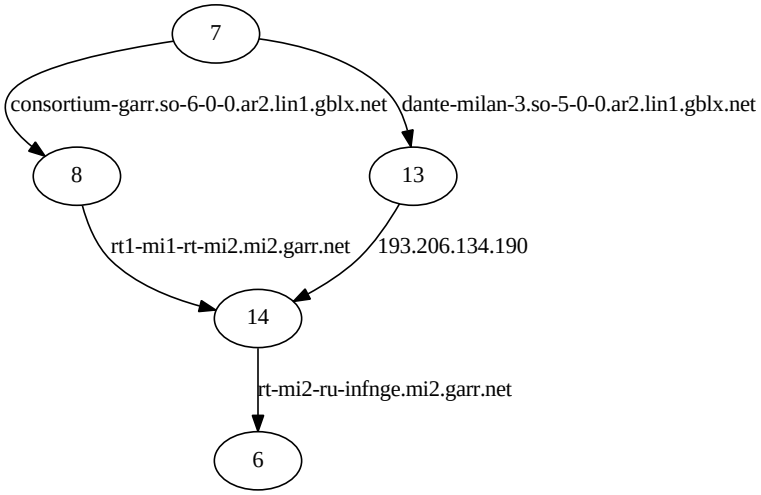
(b) first different block in the UDP analysis

Figure 4.11: analysis with fixed destination, first different block between ICMP (a) and UDP (b)

The second block is in figure 4.12. It's almost the same block, but the UDP analysis failed to merge nodes 8 and 13 in what is router 6 in ICMP's graph.



(a) second different block in the ICMP analysis



(b) second different block in the UDP analysis

Figure 4.12: analysis with fixed destination, second different block between ICMP (a) and UDP (b)

MIDAR, as already seen in section §2.4, is not a perfect, 100% correct dealiasing technique, especially in this implementation that sacrifices accuracy for data efficiency. Having a server which collects and merges data into a single graph greatly improves the dealiasing process.

Now let's compare fixed destination mode to varying destination mode. The latter discovers much more links than the former one. Consider nodes 2 and 13 in figure 4.10, which were discussed in a fixed destination context in figure 4.12. Using UDP with varying destination discovered a new link between nodes 2 and 13, so they cannot be the same router. This again stresses out the importance of having a server capable of merging clients data.

4.3 NAPT bypass validation

To validate the NAPT bypass mechanism I proposed in subsection 3.3.7 I developed a small Python script. This script, built upon the mYthon library[4], requires the definition of almost all parameters that are used in a probe, as in listing 4.1.

Listing 4.1: NAPT test usage

```
federico:mYthon federico$ python napt.py
Usage: napt.py privateIP target TTL TOS srcPort
    destPort SN newPort
```

It will send two different probes: the first one will be used to compute Δ, while the second one will use the computed value to correctly guess its answer's checksum. Let's add more details to the required parameters:

privateIP the IP of the device's network interface, which must be attached to a private network.

target the URL of the both probes' target.

TTL the Time to Live value that both probes will have.

ToS the Type of Service that must be set in both probes.

srcPort the UDP source port number used by both probes.

[4] mYthon is mYriadi for Python; it's a simple Python library that implements generic probes and graph mangling. It helped in many aspect, notably the development of the NAPT bypass technique itself and to understand how to manipulate ICMP and UDP checksums.

destPort the UDP destination port number used by the first probe.

SN the sequence number used in the first probe, while the second probe will use SN +1.

newPort the UDP destination port number used by the second probe.

The script executes the following steps:

1. parse all parameters;
2. resolve *target* parameter into an IP address;
3. find the public IP address of the router[5];
4. craft the first probe and create a payload that fixes its checksum to *sn*;
5. send the first probe and read its answer's checksum;
6. if the script runs in a device behind a NAPT it won't correctly guess the device, and will evaluate Δ;
7. craft the second probe with *newPort* and create a payload that fixes its checksum to *sn +1*;
8. the script will compute four possible checksums before sending the probe;
9. send the second probe and read its answer's checksum;
10. the test will be passed if one of the four proposed checksums is the same as the one read.

[5]There are three possible way to obtain the router's public IP address without mYriadi server:

- use a website that provides this service, like *http://canihazip.com/s/*, which is a two-liner in Python;
- use NAT-PMP, a simple yet elegant protocol to communicate with NAPTs and to request port mappings;
- use UPNP and its extension.

The test has been repeated many times, each time yielding the same result. Listing 4.2 show the output of one execution:

Listing 4.2: NAPT test output

```
1 federico:mYthon federico$ python napt.py 192.168.1.71
    www.kernel.org 2 0 2000 2020 54000 33333
2 First probe: got 0xf967, expected one of ['0xd2f0', '0
    xe926']
3 Delta is efbf
4
5 Second probe: got 0xf968 which is in ['0xd2f2', '0
    xe928', '0xf968', '0xf967']
6
7 www.kernel.org pass
```

To further validate this method, I implemented a simple automatic test tool, which repeated the test 250 times with three different targets; each instance of the NAPT test has random parameters. The results are in table 4.2.

Website	Pass	Skip	Fail	Quality
www.bbc.co.uk	250	0	0	100%
www.kernel.org	250	0	0	100%
www.unipi.it	249	1	0	99,8%

Table 4.2: NAPT bypass large scale test

www.unipi.it reported a skipped test due to packet loss; repeating the test gave different ratios, however reaching 100% from time to time.The quality is defined in equation 4.1:

$$quality = \frac{pass + 0.5 * skip}{pass + skip + fail} \tag{4.1}$$

Chapter 5

Conclusion

> "Software is like entropy. It is difficult to grasp, weighs nothing, and obeys the second law of thermodynamics; i.e. it always increases."
>
> Norman Ralph Augustine

In this work, I've described the current difficulties of creating an Internet topological map as they have been encountered by many projects. I've also explained the limits of the traditional traceroute and how to overcome them with Paris Traceroute and its evolution, MDA.

I've then presented **mYriadi**, a powerful and scalable mapping platform that executes an improved traceroute version based on MDA. I've described the overall client-server architecture and its macroscopic behavior, to then describe in detail the client structure. I've then introduced the Tracerouter analysis and its characteristics.

To demonstrate the correct behavior of the whole platform I've then proposed a traceroute campaign aimed at reconstructing the GARR network. A standalone test proved the validity of the NAPT bypass algorithm.

mYriadi shows a great potential, with its ability to handle a sheer quantity of monitors with minimum cost, its quality and its ability to adapt and extends through modules. The high diffusion of iOS device creates an enormous potential user base.

5.1 Future works

In this section I'll briefly introduce some topics that might deserve further investigation as a possible expansion of this work:

Complete IPv6 traceroute support the need to switch from IPv4 to v6 is strong, and since the year 2000 many systems started to support it. The client appliance partially supports IPv6, but IPv6 protocols and utilities, like ICMP6[27, 28], are still missing. At the time of the writing, the server appliance doesn't support IPv6, yet. Having full IPv6 support would increase corporate interest in this project, since this topic is gaining more and more attention each day. In addition, a new dealiasing technique that works with IPv6 is missing.

Secure and compressed communications mYriadi's client-server protocol should be updated to require secure data transmission; even though mYriadi does not collect any personal information, it's still high desirable that every information sent should be encrypted. Adding a compression method would allow the client to send more data using less bytes.

Net neutrality network operators apply limitations to specific network applications, i.e. VoIP. Our platform can be improved to estimate the so-called network neutrality, a measurement of the restrictions that a network imposes to a certain traffic category. Our distributed platform has the potential to benefit this kind of analysis, since we have an high quantity of clients available, and the quality of the measurement would be high: instead of using two fixed endpoints, that could easily be in privileged network area (as it happens with static traceroute monitors that are far from access networks), we would estimate the network neutrality to a certain service with monitors that would potentially be in the same spot where end users might use such service.

Other platforms support although this work is based on iPhone and, generally, iOS powered devices, the techniques shown here can be used also in other devices. Android, for example, is drawing more users to its side each day, and it's definitely a platform that requires

our attention. Work is needed to understand what are each platform's limitation and how to work around them as I did on iOS.

Bibliography

[1] "Internet usage statistics - the internet big picture (http://www.internetworldstats.com/stats.htm)." (document), 1, 1.1

[2] D. D. C. R. E. K. L. K. D. C. L. J. P. L. G. R. S. W. Barry M. Leiner, Vinton G. Cerf, "Brief history of the internet," 1

[3] "Today's road to e-commerce and global trade internet technology reports (http://www.internetworldstats.com/emarketing.htm)." 1

[4] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On power-law relationships of the internet topology," *SIGCOMM Comput. Commun. Rev.*, vol. 29, pp. 251–262, Aug. 1999. 1

[5] Z. Dezső and A.-L. Barabási, "Halting viruses in scale-free networks," *Phys. Rev. E*, vol. 65, p. 055103, May 2002. 1

[6] D. Alderson, H. Chang, M. Roughan, and S. Uhlig, "The many facets of internet topology and traffic," *Networks and Heterogeneous Media*, p. 2006. 1

[7] M. H. Gunes and K. Sarac, "Importance of ip alias resolution in sampling internet topologies," in *Proc. IEEE Global Internet Symp*, pp. 19–24, 2007. 1

[8] B. Huffaker, D. Plummer, D. Moore, and k. claffy, "Topology discovery by active probing," in *Symposium on Applications and the Internet (SAINT)*, (Nara, Japan), pp. 90–96, SAINT, Jan 2002. 1

[9] K. Lougheed and Y. Rekhter, "Border Gateway Protocol (BGP)." RFC 1105 (Experimental), June 1989. Obsoleted by RFC 1163. 1

[10] B. Augustin, T. Friedman, and R. Teixeira, "Measuring load-balanced paths in the internet," in *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, IMC '07, (New York, NY, USA), pp. 149–160, ACM, 2007. 1.1, 2.3

[11] A. Iaria, "Progettazione ed implementazione dell'infrastruttura server per un sistema distribuito di scansione e mappatura della rete internet.," Master's thesis, Università di Pisa, Dip. Ingegneria dell'Informazione, 2011. 1.1

[12] G. Malkin, "Traceroute Using an IP Option." RFC 1393 (Experimental), Jan. 1993. 2.1

[13] J. Postel, "Internet Control Message Protocol." RFC 792 (Standard), Sept. 1981. Updated by RFCs 950, 4884. 2.1

[14] D. Malone and M. Luckie, "Analysis of icmp quotations," in *Passive and Active Network Measurement* (S. Uhlig, K. Papagiannaki, and O. Bonaventure, eds.), vol. 4427 of *Lecture Notes in Computer Science*, pp. 228–232, Springer Berlin / Heidelberg, 2007. 2.1

[15] A. Broido, Y. Hyun, and k. claffy, "Spectroscopy of traceroute delays," in *Passive and Active Network Measurement* (C. Dovrolis, ed.), vol. 3431 of *Lecture Notes in Computer Science*, pp. 278–291, Springer Berlin / Heidelberg, 2005. 2.1.1

[16] Cisco, "How does load balancing work?," tech. rep., 2005. 2.1.2

[17] J. Moy, "OSPF Version 2." RFC 2328 (Standard), Apr. 1998. Updated by RFCs 5709, 6549. 2.1.2

[18] R. Callon, "Use of OSI IS-IS for routing in TCP/IP and dual environments." RFC 1195 (Proposed Standard), Dec. 1990. Updated by RFCs 1349, 5302, 5304. 2.1.2

[19] B. Quoitin, C. Pelsser, L. Swinnen, O. Bonaventure, and S. Uhlig, "Interdomain traffic engineering with bgp," vol. 41, no. 5, pp. 122–128, 2003. 2.1.2

[20] B. Augustin, X. Cuvellier, B. Orgogozo, F. Viger, T. Friedman, M. Latapy, C. Magnien, and R. Teixeira, "Avoiding traceroute anomalies with

paris traceroute," in *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, IMC '06, (New York, NY, USA), pp. 153–158, ACM, 2006. 2.2, 3.3.5

[21] K. Keys, Y. Hyun, M. Luckie, and k. claffy, "Internet-scale ipv4 alias resolution with midar: System architecture - technical report," tech. rep., Cooperative Association for Internet Data Analysis (CAIDA), May 2011. 2.4

[22] K. Keys, Y. Hyun, M. Luckie, and k. claffy, "Internet-scale ipv4 alias resolution with midar," *IEEE/ACM Transactions on Networking*, 2012. 2.4

[23] K. Keys, "Internet-scale ip alias resolution techniques," *ACM SIGCOMM Computer Communication Review (CCR)*, vol. 40, pp. 50–55, Jan 2010. 2.4

[24] R. Braden, D. Borman, and C. Partridge, "Computing the Internet checksum." RFC 1071, Sept. 1988. Updated by RFC 1141. 3.3.1

[25] K. Egevang and P. Francis, "The IP Network Address Translator (NAT)." RFC 1631 (Informational), May 1994. Obsoleted by RFC 3022. 3.3.7

[26] P. Srisuresh and M. Holdrege, "IP Network Address Translator (NAT) Terminology and Considerations." RFC 2663 (Informational), Aug. 1999. 3.3.7

[27] A. Conta and S. Deering, "Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6)." RFC 1885 (Proposed Standard), Dec. 1995. Obsoleted by RFC 2463. 5.1

[28] A. Conta and S. Deering, "Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification." RFC 2463 (Draft Standard), Dec. 1998. Obsoleted by RFC 4443. 5.1