FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	IPv6 support for Globalping
Student:	Valéria Frčková
Supervisor:	Ing. Martin Kolárik
Study program:	Informatics
Branch / specialization:	Web and Software Engineering, specialization Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2024/2025

Instructions

Globalping is an open-source platform that allows anyone to run networking commands (ping, traceroute, dig, curl, and mtr) on probes distributed around the world. It is currently IPv4-only. The goal of this work is to add support for targeting remote hosts over IPv6 and allow the probes to run in IPv6-only networks.

Proceed in the following steps:
1. Analyze the user requirements regarding the IPv6 functionality. Coordinate with the project's maintainers.
2. Propose a solution based on the requirements and analyze the necessary changes on the probes, the API, and related systems.
3. Implement the proposed solution.
4. Document and test all changes.

Electronically approved by Ing. Michal Valenta, Ph.D. on 28 January 2024 in Prague.

Bachelor's thesis

IPV6 SUPPORT FOR GLOBALPING

Valéria Frčková

Faculty of Information Technology
Department of Software Engineering
Supervisor: Ing. Martin Kolárik
May 17, 2024

Citation of this thesis: Frčková Valéria. *IPv6 support for Globalping.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Contents

List of Figures

List of Tables

List of code listings

I would like to thank my supervisor Ing. Martin Kolárik for his interest in this topic, his patience, and his great advice. I would like to thank my beloved mother, who supported me tirelessly. I would also like to thank my dear Kamil for his proficiency and willingness to help. Last but not least, I would like to thank myself for never giving up and making it through every minute.

Declaration

In Prague on May 17, 2024

Abstract

The thesis aims to extend the functionality of the Globalping platform to support IPv6 targets and allow probes running on dual-stack and IPv6-only networks. The need to support IPv6 addresses the gradual depletion of IPv4 address space, inevitably leading to the transition to IPv6 address space.

The platform enables its users to effectively monitor, benchmark, and optimize the performance of their global services. The thesis includes a comprehensive overview of the platform's current functionality, a detailed analysis of user requirements concerning IPv6 support, and design considerations. The process of implementation and automated testing is documented in great detail. Among others, it includes additional checks for private and malicious addresses.

The thesis successfully delivers the solution for incorporating IPv6 support into the Globalping platform.

Keywords Glopalping, IPv6 connectivity, IP address, probe, network performance

Abstrakt

Bakalárska práca sa zameriava na rozšírenie funkcionality platformy Globalping tak, aby podporovala IPv6 domény a povolila sondy bežiace v dual-stack sieťach a výlučne IPv6 sieťach. Potreba podpory IPv6 adresuje postupné vyčerpávanie adresného priestoru IPv4, čo nevyhnutne vedie k prechodu na adresný priestor IPv6.

Platforma umožňuje svojim používateľom efektívne monitorovať, porovnávať a optimalizovať výkon ich globálnych služieb. Práca obsahuje komplexný prehľad súčasnej funkcionality platformy, podrobnú analýzu užívateľských požiadaviek na podporu IPv6 a úvahy o dizajne. Proces implementácie a automatizovaného testovania je vrámci práce detailne zdokumentovaný. Okrem iného zahrňuje aj pridané overenie na privátne a škodlivé adresy.

Práca úspešne dodáva riešenie pre začlenenie podpory IPv6 do platformy Globalping.

Klíčová slova Glopalping, IPv6 konektivita, IP adresa, sonda, výkon siete

List of abbreviations

API	Application Programming Interface
CLI	Command Line Interface
DNS	Domain Name System
EDNS	Extension Mechanisms for DNS
HTTP	Hypertext Transfer Protocol
ID	Identification
IP	Internet Protocol
IPv6	Internet Protocol version 6
URI	Uniform Resource Identifier

Introduction

Digital connectivity is an inseparable part of our everyday interactions, whether it is work, communication, or commerce. As the demands on network infrastructure are growing, optimizing it and dealing with new challenges is necessary. One such challenge is the depletion of IPv4 address space. The rapid increase in the number of devices connected to the internet has overwhelmed the IPv4 system, which on its 32-bit architecture offers only about 4 billion addresses. Because of this shortage, the need to satisfy the burgeoning demand for new IP addresses unavoidably leads to the switch to IPv6 address space. With its 128-bit architecture, it offers exponentially more addresses. Apart from that, it also plays a role in improving the performance of the network, while making it more secure. For organizations to fully benefit from this system, they have to overcome compatibility issues, provide new infrastructure, and hire network administrators with specialized knowledge.

The Globalping platform with its network of globally distributed probes offers monitoring, debugging, and benchmarking of the network helping with optimizing the performance and reliability of the internet infrastructure across geographical boundaries. Currently, users of the platform are able to test IPv4 targets only. In order for Globalping to stay relevant in a new era of network visibility and resilience, it needs to face the pressing matter of IPv4 address space running out by extending its functionality to include IPv6 targets as well.

The main goal of this Bachelor's thesis is to design, implement, and test a solution for extending this open-source platform to support IPv6 addresses. The new extension will add support for targeting remote hosts over IPv6. Probes running in IPv6-only networks, as well as probes running on dual-stack will be allowed.

The Globalping chapter will break down the current functionality of the platform as well as its architecture and implementation details illustrated by diagrams.

The Analysis and Design chapter will contain the analysis of user requirements for the new extension, then it will discuss the options for how to approach individual tasks according to the user requirements. The options will then be proposed to the platform maintainers. The decision-making process behind the final choice will be explained. All the necessary changes to be made on probes, the API, and related systems in order to realize the approved solution will be listed to form an implementation plan.

The process of implementing the solution will be summarized in the Implementation chapter. Any challenges that may occur during this phase together with adjustments to the initial plan will be documented here as well.

The final chapter Testing and Documentation will first deal with the process of testing the solution thoroughly using unit tests and integration tests where needed and then it will present a list of edits and additions to the official platform documentation. Only functionality changes relevant to the user will be added.

Although similar platforms dealing with measuring Internet connectivity and reachability using a globally distributed network of probes, such as Ripe Atlas[1] or keycdn[2] tool exist, this thesis does not include their comparison to Globalping platform, as it is not part of the goals.

Apart from adding Ipv6 support, two other Globalping-related topics are currently being processed as Bachelor's theses. One focuses on the security aspect of Globalping and includes a comparison to other platforms in terms of security practices. The other evaluates the impact of the EDNS Client Subnet Extention on the performance of web services, such as Globalping.

[1]https://atlas.ripe.net/
[2]https://tools.keycdn.com/ipv6-ping

Chapter 1

Globalping

The following chapter's introduction is based on the source Globalping [1]. Globalping is a platform that allows anyone to run networking commands such as ping, traceroute, dig, curl, and mtr on probes distributed all around the world. The goal of the platform is to provide a free and simple service for everyone in order to make the internet faster. Users can use it to:

- optimize their anycast network,
- monitor their latency by running a global latency test,
- debug routing issues,
- check for censorship in different countries,
- benchmark their internet infrastructure and understand how the network performs,
- compare DNS providers to find the fastest one globally or in their region,
- run network tests globally or from specific locations and regions,
- troubleshoot their networking problems and share the test results with others.

Users can utilize a globally distributed network of probes. As of April 2024, Globalping has potentially as many as 858 connected probes in 252 different cities and 73 other countries at its disposal at any given time. Every component of Globalping is open source, including a simple-to-use Globalping CLI, from which network tests can be run and scripted from all over the world without ever leaving the terminal. Furthermore, the free Globalping REST API can be used to build new tools, automate tests and this way be integrated into the existing flow. Since the same measurement can be done on the same subset of probes for up to 7 days, users can benefit from using Globalping by comparing the measurements done before and after optimizing their global service. In practice it allows users to check whether making changes in the configuration makes the infrastructure faster by pinging the service, or if adding more servers reduces latency since in theory if more locations are added, it should be closer to the user of their service.

1.1 Globalping API

The information in the following sections is based on Globalping documentation [2]. The API offers three operations – creating a measurement, getting a measurement, and listing all available probes. The following sections will expand on each of them, including their purpose, their function, what they expect as a request, and how the responses they return look like.

1.1.1 Create a measurement

POST /v1/measurements

This endpoint creates a new measurement with parameters set in the request body. The measurement runs asynchronously and its current state can be retrieved at the URL returned in the `Location` header.

```json
{
  "type": "ping",
  "target": "cdn.jsdelivr.net",
  "locations": [
    {
      "country": "DE",
      "limit": 4
    },
    {
      "country": "PL",
      "limit": 2
    }
  ]
}
```

■ **Code listing 1.1** The request body to set the measurement parameters.

As can be seen in the request body Code listing 1.1, the test `type` that will be run on the probes can be specified. The types include:

Ping – this type is used to troubleshoot connectivity, reachability, and name resolution [3] by trying to send test packets to a particular device and waiting for its response. If the response is returned, there is a physical connection between the two devices. If no response is returned, this could indicate a problem with the physical connection [4],

Traceroute – identifies the route a packet takes between the probe and the destination computer specified in the target. It lists every hop along the path between the two computers. This can help identify whether communications take too many hops in the wrong direction or whether certain nodes are out of commission [5]. It will return the names or IP addresses of all routers between two devices. This can help to see where a packet may be misguided [4],

DNS – tools that allow users to query DNS servers to retrieve information about domain names, IP addresses, and DNS records. It is useful for troubleshooting DNS issues and performing DNS lookups. It supports various query types, including A, AAAA, CNAME, MX, NS, PTR, SOA, and TXT records [6],

MTR – is a robust network diagnostic tool combining the functionality of ping and traceroute commands. Its output contains information about the entire route a packet takes from the source server from which it originated to the destination server [7]. It is another common method for testing network connectivity and speed. In addition to the hops along the network path, MTR shows constantly updated information about the latency and packet loss along the route to the destination. This helps with troubleshooting network issues by presenting what's happening along the path in real-time [8],

HTTP – this test type transfers data over HTTP network protocol in order to simulate and examine HTTP traffic. Allowed methods are GET and HEAD.

Each measurement type has a different set of options. For example, a user can specify the number of packets to send, port number, preferred transport protocol…

A user has to specify the target in order to execute the measurement. Depending on the measurement type, the `Target` field accepts an input of hostname or IPv4 address. The target has to be publicly reachable, otherwise it would not pass the validation criteria. Target is also checked against the lists of known malicious addresses known as blacklists. The blacklists are updated periodically.

In the `Location` field, a user can specify from where the probes for the measurement are to be selected. The input can contain anything that makes sense in terms of location and the algorithm will do its best to select the most appropriate probes. It is said to use 'magic' for its feature to select the most appropriate location based on the input. Valid inputs include names of countries, continents, cities, data center tags, etc. It is possible to select multiple locations divided by a comma.

The `Limit` field specifies the maximum number of probes that should run the measurement. The result count might be lower if there are not enough probes available in the specified locations. It is mutually exclusive with the limit property that can be set for individual locations. The limit can be any number from 1 to 500.

```
{
    "id": "PY5fMsREMmIq45VR",
    "probesCount": 1
}
```

Code listing 1.2 Create a measurement response.

If the API accepts the request for processing, it returns a body containing the ID of the newly created measurement. The example response can be Code listing 1.2. The URL from the `Location` header can be used to retrieve the measurement status. The response contains information about the newly created measurement. The `id` field is the id of the measurement. It can be further used to create a new measurement request, reusing the same probes. The `probesCount` field refers to the actual number of probes that performed the measurement tests. It can be smaller or equal to the limit, depending on probe availability.

1.1.2 Get a measurement by ID

GET /v1/measurements/id

```
{
  "id": "nzGzfAGL7sZfUs3c",
  "type": "ping",
  "status": "finished",
  ...
  "target": "cdn.jsdelivr.net",
  "probesCount": 1,
  "measurementOptions": {
    "packets": 2
  },
  "results": [
    {
      "probe": {
        "continent": "OC",
        "region": "Australia and New Zealand",
        "country": "NZ",
        "state": null,
        "city": "Auckland",
        ...
        "tags": [
          "datacenter-network"
        ],
        "resolvers": [
          "1.1.1.1",
          "8.8.8.8"
        ]
      },
      "result": {
        "status": "finished",
        "rawOutput": "PING jsdelivr.map.fastly.net (151.101.129.229) 56(84) bytes of data..."
      }
    }
  ]
}
```

Code listing 1.3 The response Get a measurement by ID.

The endpoint is used to retrieve the status and results of an existing measurement. Measurements are typically available for up to 7 days after creation. A link to this endpoint is returned in the `Location` response header when creating the measurement. The request consists of a single parameter `id` – the ID of the measurement that is to be retrieved. It can be obtained from the attribute `id` from the create measurement response. The request itself may take a few hundred milliseconds to complete. A successful request returns status `200 OK` and a body containing the requested measurement results. As can be seen in the response Code listing 1.3 the results include `id` and all the options set by the user, such as `type` of the measurement or `target`. Furthermore, it contains the current measurement `status`, where any value other than `in-progress` is final (the measurement is no longer running). `Locations` is either an `id` of a previous measurement whose probes would be reused or an array of locations from which the measurement will be run.

1.1.3 Probes

Probes are end devices that process requests from users. They are system components that monitor and analyze network activity and may also perform prevention actions [9]. They are a critical tool for IT and network administrators to monitor network performance in real-time. They act as messengers, delivering questions to network devices and retrieving data to be analyzed by network monitoring software. Probes help prevent bottlenecks, slowdowns, and downtime by providing nearly real-time information about the network, allowing administrators to take quick action. By using probes, administrators can focus on other IT priorities while ensuring that their network is functioning optimally [10]. A probe can be any device e.g. a Raspberry Pi anywhere in the world can become a probe, if a vendor chooses to connect it to the Globalping network by running a docker container. The container will work on both x86 and ARM architectures. Emphasis is being put on the security and privacy of the system, which is why the public information exposed by probes is minimized and harmful or abusive domains and IPs are blocked [1].

GET /v1/probes

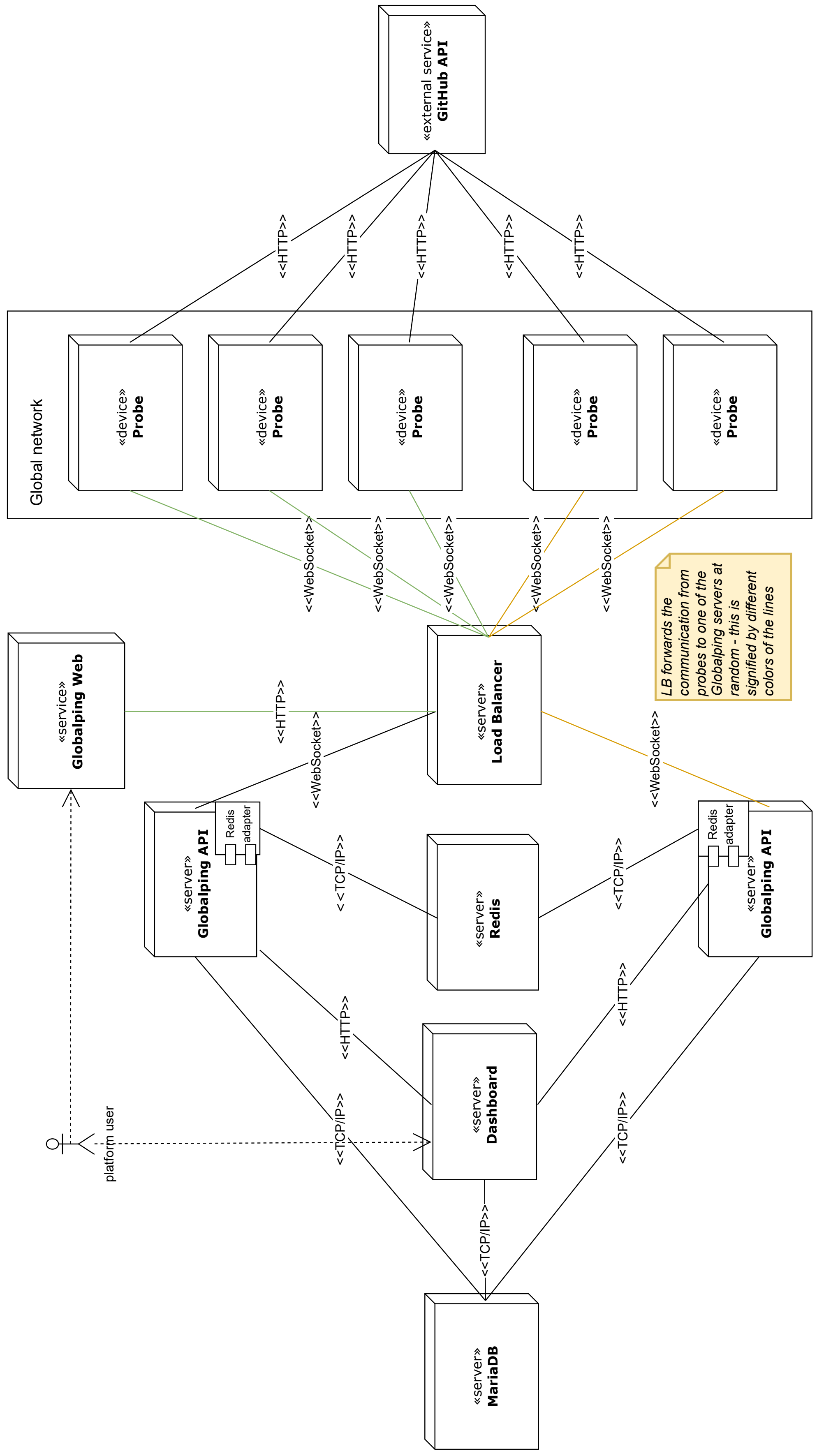
```
[
  {
    "version": "0.20.0",
    "location": {
      "continent": "NA",
      "region": "Northern America",
      "country": "US",
      "state": "VA",
      "city": "Ashburn",
      "asn": 14618,
      "network": "Amazon.com, Inc.",
      "latitude": 39.0437,
      "longitude": -77.4875
    },
    "tags": [
      "aws-us-east-1",
      "datacenter-network"
    ],
    "resolvers": [
      "private"
    ]
  }
]
```

Code listing 1.4 List probes currently online response.

This endpoint returns a list of all probes currently online and their metadata as in example response body Code listing 1.4. Probes are specified by the requested location or an ID of an existing measurement when creating new measurements. The metadata includes for example `location`, assigned `tags` that are additional values to fine-tune probe selection such as `aws-eu-west-1` or `datacenter-network` and a list of the default `resolvers` configured on the probe. These can be either IPv4 addresses or constants, which indicate that the resolver points to a private IP address.

1.2 Architecture

Since it is essential to understand the internal processes of the system in order to implement new functionalities, the purpose of this section is to create an overview of the system architecture, its internal structure, its components, and their interaction with each other and other external elements. The following subsections explain the deployment diagram shown in Figure 1.1, including the event-driven nature of the platform, the role of Socket.IO library in the real-time communication between its components, the use of Redis for data storage, containerization with Docker, CI/CD pipelines and load balancing.



■ **Figure 1.1** Globalping API deployment diagram

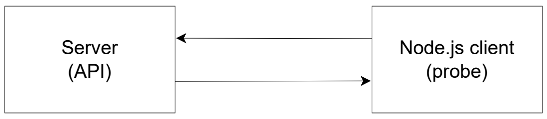


Figure 1.2 Bidirectional communication between server and client

1.2.1 Event-driven architecture

Apart from the website and CLI repository, the platform consists of two code repositories – `globalping` representing an API and `globalping-probe` running on every probe in the network. Unusually, the communication of these separate parts is based on producing and subscribing to events. This design pattern is called event-driven architecture – it is the foundational design of an application's communication of state changes around an asynchronous exchange of messages called events. The architecture allows the application to be developed as a highly distributed and loosely coupled organization of components [11]. The communication is bidirectional, which means that both API and probes both produce and subscribe to events.

1.2.2 Socket.IO

The exchange of events between probes and API is realized via WebSockets using `Socket.IO` – it is a library that enables real-time, bidirectional, and event-based communication between the browser and the server [12]. It consists of a server and client as shown in Figure 1.2. The client (in this case the probe) will try to establish a WebSocket connection if possible and will fall back on HTTP long polling if not. WebSocket is a communication protocol that provides a full-duplex and low-latency channel between the server and the browser [12].

1.2.3 Redis

Redis is an in-memory database that provides solutions for caching, vector search, and NoSQL databases making it simple for digital customers to build, scale, and deploy applications. It offers different levels of on-disk persistence and provides high availability and automatic partitioning [13].

Redis works with an in-memory dataset. The data can be persisted either by periodically dumping the dataset to disk or by appending each command to a disk-based log. The persistence can also be disabled if only an in-memory cache is needed.

Redis supports asynchronous replication, with fast non-blocking synchronization and auto-reconnection with partial resynchronization on net split.

In Globalping, Redis is used to cache GeoIP information from its 3 IP databases, to store all measurement results for a period of up to 7 days, and to sync connected probes between multiple API instances [14]. The measurement results can be later accessed by id. They cannot be stored in memory, since the API runs in multiple instances. The data consists of everything the API gathered about a particular measurement – id, limit, location, probe count, collection of results from probes, status, target, type, …It is configured to persist data periodically, so there is a risk of losing some data.

1.2.3.1 Pub/Sub channels in Redis

The messaging is implemented according to the Publish/Subscribe paradigm, where messages sent by the publishers are characterized into channels without knowing which subscribers there may be. Subscribers only receive messages from the channels they expressed an interest in, also without knowing which publishers there are. This decoupling of publishers and subscribers allows for greater scalability and a more dynamic network topology [15].

1.2.4 Redis adapter of Socket.IO

Scaling to multiple servers gives rise to the problem of one server having a connection to a probe and at the same time another server wanting to communicate with the same probe without the connection to it. In order to properly route events to all clients, the default in-memory adapter needs to be replaced by another implementation.

The Redis adapter is a server-side component, that is responsible for broadcasting events to all or a subset of clients(subscribers) [16]. It relies on the Redis Pub/Sub mechanism. Every packet that is sent to multiple clients (probes) is sent to all matching clients connected to the current Globalping API server and published in a Redis channel while being received by the other Globalping API servers of the cluster [17] as can be seen in Figure 1.1.

1.2.5 MariaDB

MariaDB is an open-source relational database management system. It ensures high compatibility with MySQL, allowing applications to transition to MariaDB with minimal modifications. It plays a role in maintaining data integrity and offering powerful querying possibilities. It is designated as a solid, scalable, and secure data storage solution for modern applications [18] like Globalping.

In the Globaping architecture, MariaDB serves as a database for storage of transactional data, especially information about adopted probes, credits, and tokens [14]. The API communicates with it to retrieve this data, for example, to determine if a probe is assigned to any user (if it is adopted). The users can then manage their adopted probes through Dashboard, for example by modifying the probe's location or assigning tags to the probe. The information about credits shows how many requests a user with adopted probes can make after exceeding the free quota.

1.2.6 Deployment and Containerization

Docker serves as an essential tool in the Globalping deployment and containerization strategy [14]. It enables automation of deployment using continuous integration and continuous deployment (CI/CD) pipelines – it consists of a series of steps that must be performed in order to deliver a new version of the software, such as development, testing, production, and monitoring of an application. The true value of the pipelines lies in these steps being automated, so that the code of the applications can be developed faster, more securely, and with higher quality [19].

The Globalping master branch undergoes automatic compilation into a Docker container via Docker Hub, making the latest version of the application available as a Docker image.

The platform relies on Docker Swarm [14] in a Network host mode for triggering manual deployment and this way ensures efficient resource allocation and effortless container orchestration across multiple hosts. Through Docker Swarm, containers can connect to multiple hosts, enhancing application availability throughout the system [20].

1.2.7 Load balancing

The Globalping platform uses a single load balancer [14] to ensure that its servers are used equally. It performs periodical health checks to prevent issues that can cause application downtime, making the application unavailable to visitors [21].

In order to speed up the decryption process and reduce the processing burden on backend servers, the load balancer utilizes the process of TLS termination [14], which refers to decrypting encrypted traffic (HTTPS). In this process, the SSL/TLS encryption is terminated, and the communication between the client and the server/application happens over unencrypted HTTP [22]. The communication is secured by TLS encryption of the data through automated Let's Encrypt certificates [14].

1.3 Implementation details

The crucial parts of the whole system are Globalping API servers and probes distributed around the world. In order for these separate components to function as a unit, a connection needs to be established.

The whole process is visualized in the sequence diagram Figure 1.3. As shown in the diagram, the communication is initialized on the client side – the probe. Each probe calls function `io(server URI)` with URI of the server as an argument, which establishes a connection with the server by sending a request to it. In the process, it also emits a special event called `'connect'`. Upon this new connection, the server side – the API – emits `'connect'` event as well. The probe initializes an instance of `StatusManager` class, which tracks the current status of an IPv4 communication. This status is initially set to `'initializing'`. As the first thing after establishing a connection with the API, it sends a status update back to the API in an event `'probe:status:update'`. The API handles it by setting the value `status` stored in `Probe` object to the value sent within an event data. The probe would not be used for processing measurements unless its status equals to `'ready'`. In order to do so, first the API determines the approximate geographical location of the probe using its IP address. This is done using algorithm called Geo IP in the following steps:

1. the IP address of a probe is passed to 5 different geo IP info providers – they have databases that map IP addresses for location with low precision since there are so many addresses – the data from each provider is then collected,
2. for data from every provider a city approximation is calculated, so that the most relevant city based on the provider's country, population size, and proximity to the original location is provided for the final approximation,
3. the providers are grouped together based on the city they provided – the city from the largest group or group with the highest priority is then chosen as the city approximation that will be used to describe the probe.

Now the API is ready to connect the location to the probe via an event `'api:connect:location'`. The probe starts the status manager and sends available public IPv4 DNS resolvers in an event `'probe:dns:update'`, As the status manager starts, it runs a recursive function `pingTest()`, which tries to ping 4 reliable domains via IPv4. If more than half of the pings were successful, the status is updated to `'ready'` in an event `'probe:statusIPv4:update'`, if not, it is updated to status `'ping-test-failed'`. The function `pingTest()` then calls itself in 10-minute intervals until the connection is interrupted. The probe can be connected to the API for weeks until it disconnects from the WiFi, the socket is forcefully disconnected or there is a connection error. Upon disconnection, the Socket instance fires event `'disconnect'`, and then the probe tries to reconnect itself. [23]

Now that the communication is taken care of by the events happening in the background, the measurements themselves can be processed.

The sequence diagram in Figure 1.4 illustrates how the measurement is carried out. It begins with the user's click on the button `Run test`, then the API selects the number of probes equal to the`limit` set to the measurement by the user. It always filters through currently online probes only, which means probes that are connected to the WiFi in real time. If the location is set to 'world', probes are selected randomly, if not, the probes matching the location are selected in the function `findMatchingProbes()` – the output is `onlineProbesMap` containing filtered probes. The

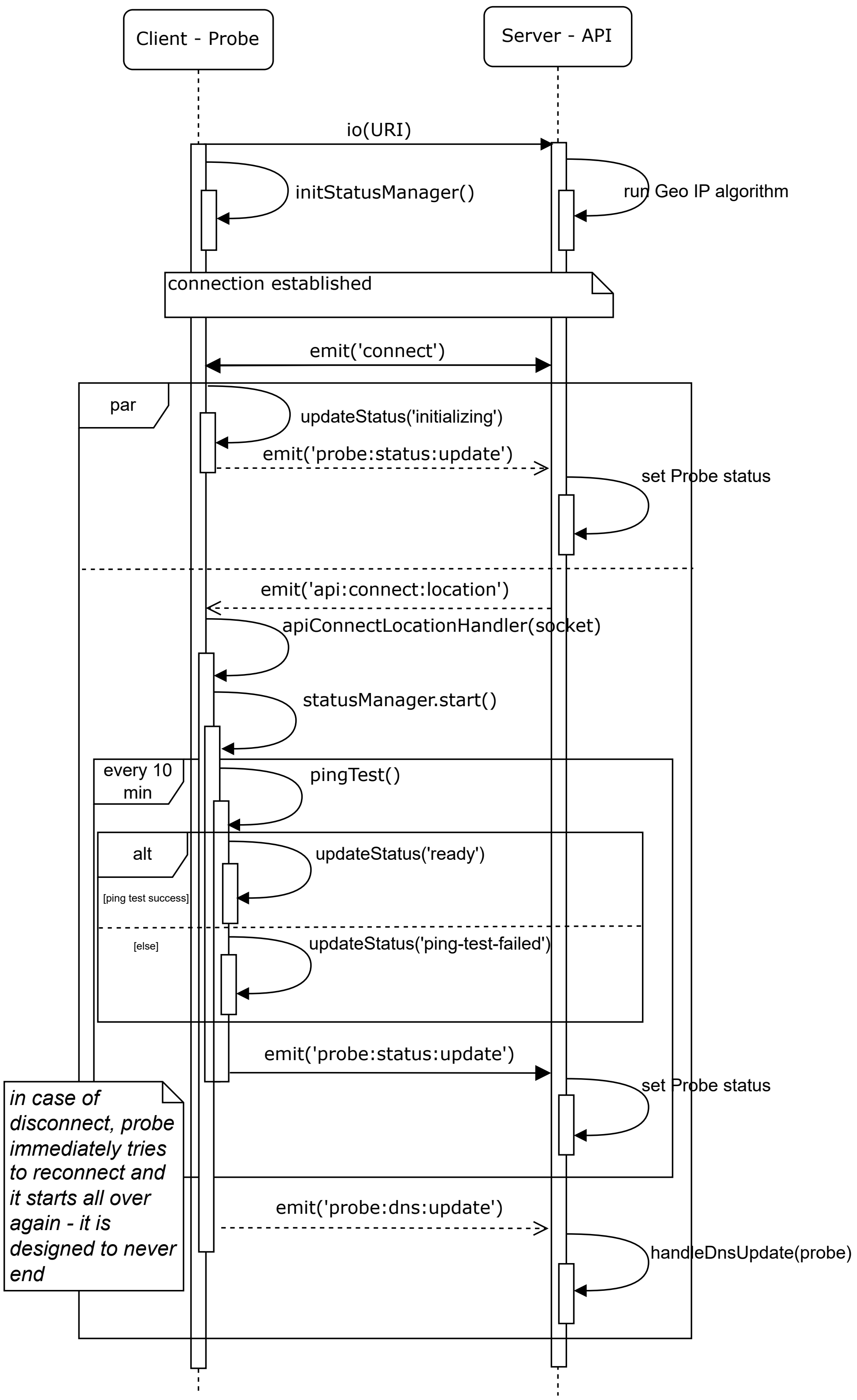

Client - Probe
Server - API
io(URI)
initStatusManager()
run Geo IP algorithm
connection established
emit('connect')
par
updateStatus('initializing')
emit('probe:status:update')
set Probe status
emit('api:connect:location')
apiConnectLocationHandler(socket)
statusManager.start()
every 10 min
pingTest()
alt
updateStatus('ready')
[ping test success]
[else]
updateStatus('ping-test-failed')
emit('probe:status:update')
set Probe status
in case of disconnect, probe immediately tries to reconnect and it starts all over again - it is designed to never end
emit('probe:dns:update')
handleDnsUpdate(probe)

Figure 1.3 Sequence diagram of the connection between API and Probe

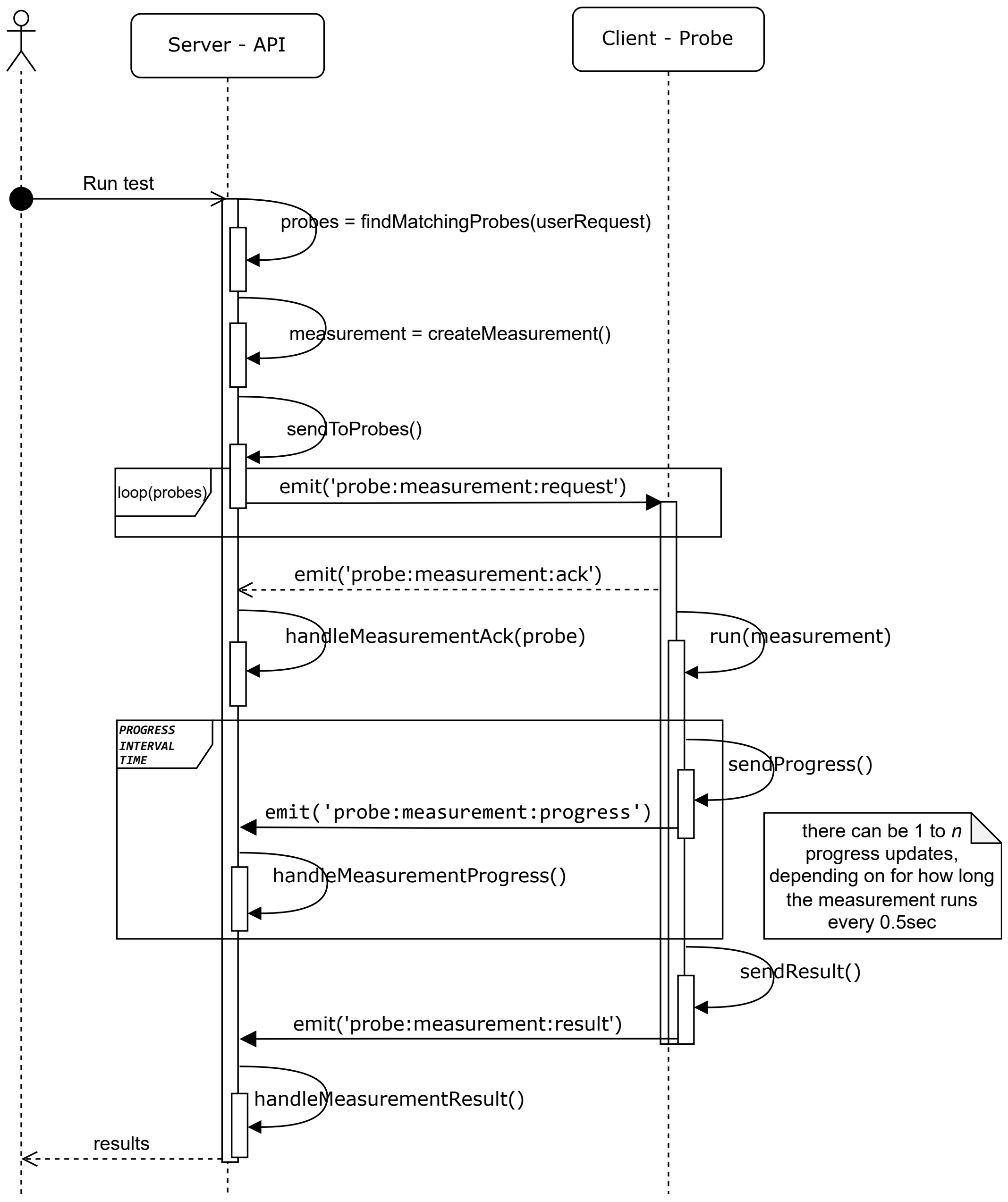

Server - API
Client - Probe
Run test
probes = findMatchingProbes(userRequest)
measurement = createMeasurement()
sendToProbes()
loop(probes)
emit('probe:measurement:request')
emit('probe:measurement:ack')
handleMeasurementAck(probe)
run(measurement)
PROGRESS INTERVAL TIME
sendProgress()
emit('probe:measurement:progress')
handleMeasurementProgress()
there can be 1 to *n* progress updates, depending on for how long the measurement runs every 0.5sec
sendResult()
emit('probe:measurement:result')
handleMeasurementResult()
results

Figure 1.4 Sequence diagram of processing a measurement

API sends a measurement request to each probe in an event `'probe:measurement:request'` and then asynchronously collects results and stores them in Redis. The API does not wait for the results after assigning the measurement – the probes will report them back once they are ready. As soon as the status of the probe is `'ready'`, it can receive a measurement request provided it was selected by the API. The probe then runs the handler according to the measurement type. In each handler first, the options set by the user are validated according to the command schema using Joi – the most powerful schema description language and data validator for JavaScript. Then the command itself is executed and the result is returned [24]. It is possible for a user to inquire about the state of the measurement in the meantime. If the results are not ready, the state is `'in-progress'`. The user can inquire repeatedly since neither side participating in the communication is ever waiting. As the probes are finalizing their partial results, the measurement results are being completed until the state of the measurement is reported to be `'finished'`. Until then, the incomplete contents of the results provided by the API are undefined. If an error occurs during the carrying out of the measurement, it ends up in the `timeout` status.

1.3.1 Joi

As mentioned earlier, the user input is validated before assigning measurements to the individual probes. The validation is managed by utilizing a JavaScript data validation library called Joi. It offers intuitive and readable language for describing data. It functions by first constructing a schema using the types and constraints provided by the user of the library (a developer) and then validating the values against the defined schema. In Globalping, the user input is validated to ensure only valid and expected input is accepted thus enhancing security and eliminating potential errors and vulnerabilities [25].

1.4 Limitations

While Globalping is robust in its capabilities for testing network performance across geographical boundaries, its potential is limited by the absence of support for testing IPv6 targets and utilizing dual-stack and IPv6-only probes. As the IPv4 address space is running out, the transition to IPv6 address space is inevitable. Naturally, this would lead to a higher demand on network infrastructures to provide compatibility with IPv6 devices. It is certainly beneficial for Globalping to leverage the vast number of new devices connected to the internet in the IPv6 address space and scale efficiently.

The diagram in Figure 1.1 shows that the user communicates with the Globalpig API and the API communicates with probes. If the probes are running in the networks supporting only IPv6, in order to communicate, the API also has to run on IPv6. Adding the IPv6 support is therefore crucial for Globalping's future relevance.

Chapter 2

Analysis and Design

The following chapter explores the necessary modifications to extend Globalping's support for IPv6. Firstly the user requirements are presented and then analyzed. Several approaches how to integrate the changes are suggested and then discussed. At the end of each section, the final solution design is decided.

2.1 User requirements analysis

The information in this section is based on the GitHub issue [26]. There are two main user requirements for the new extension. The first is for Globalping to be able to target remote hosts over IPv6, the second is to allow probes to run in IPv6-only networks. That means that the user will be able to specify an IPv6 address in the `Target` field. Apart from the IPv6 address as a target, if the user specifies a domain, there must be a way to choose a connection type: auto/ipv4/ipv6. Furthermore, the anti-abuse system must also support IPv6. Probes themselves will have to detect what IP versions they support and let it be known. All the partial requirements are summarized below.

R1: Targeting IPv6-enabled hosts in tests.

- **R1.1:** Being able to specify IPv6 address in all relevant inputs.
  - **R1.1.1:** This should automatically filter the probes to only those that support IPv6. Similarly, IPv4 addresses should be filtered to those that support IPv4
  - **R1.1.2:** On the probe side, we need to test the output parsing for each test type.
- **R1.2:** There should be an option to specify the connection type (auto/IPv4/IPv6) when a domain is specified
- **R1.3:** The anti-abuse system should support IPv6.

R2: Probes running in dual-stack and IPv6-only networks.

- **R2.1:** Probes detect which connection types they support for tests and report this to the API.
- **R2.2:** There should be only one connection to the API, either via IPv4 or IPv6, determined by the probe. However, this choice does not affect the user's ability to run tests of the other type.

For successful deployment, adjustments to Globalping's infrastructure need to be made as well. These adjustments are out of the scope of this thesis since they are not directly linked to providing support for targeting IPv6 hosts. The changes are on the network level. They involve adjusting both the probe update system and a load balancer to accept IPv6 connections.

As can be seen in the deployment diagram in Figure 1.1, the probe is connected to the GitHub API. It requests GitHub every 10 minutes to determine if a new version of the code has been released, so it can be updated. If there is a new version, the probe downloads it and restarts itself. The complication arises from the fact that GitHub on the endpoint where the code is stored does not yet support IPv6. This presents a significant challenge, as the new extension’s objective is to enable probes to run on an IPv6-only network. Consequently, this limitation would prevent probes from updating themselves.

When the IPv6-only probe updates the docker image, it has to be downloaded via IPv6, so the repository containing the docker image must also support IPv6. The current GitHub repository hosting the docker images does not support IPv6.

The probe is also connected to the load balancer, through which it communicates with the Globalping server. This means that on the network level if a probe initiates a connection, the API needs to have an IPv6 address so that it can accept the connection.

2.2 User’s choice of IP version

The following section focuses on the user requirement R1.2.1 dealing with how a user can choose which IP version the measurement should use. It will bring insight into the discussion on where to integrate a new field `'ipVersion'`. It will also discuss the default IP version and the reasons behind the choice.

It is important to mention that the user can specify the IP version only if the target is a domain. If the target is an IP address, the IP protocol will be decided automatically. If the target is a domain and the user specifies an IP version non-corresponding to the IP version of the target, the API will send the measurement regardless, and it will fail on the probe side, propagating the error back to the API.

2.2.1 Selecting IP version

The following information is based on private communication [27]. The project maintainers required that for measurement types ping, traceroute, mtr, and HTTP, users should be able to set a preferred IP protocol that will be used for the measurement, provided the target is a domain. It was supposed to have values `'IPv4'`, `'IPv6'` and `'auto'`.

There were two approaches to enabling the selection of an IP version that were considered – either to add a new measurement option `'ipVersion'` to the relevant types of the measurements as shown in Code listing 2.1, or to add it as another tag within `'locations'` as shown in Code listing 2.2. After further evaluation of the advantages and disadvantages of both approaches discussed below, the first approach has been chosen for its simplicity and straightforwardness. This was also the preferred approach of the project maintainers.

2.2.1.1 Selecting IP version within measurement options

The first approach with measurement options at first sight seems to be consistent with other similar cases, for example for command `http`, there is an option protocol with values `'HTTP'`, `'HTTPS'` and `'HTTP2'`. Adding an option would be easy to document and easy to discover for users on all platforms.

On the contrary, not all probes support both protocols, so the option implies an automatically applied filter (to select only those probes that support the protocol if specified). Furthermore, the automatically applied filter may result in a “no suitable probes found” error if there are no probes supporting the selected IP version in the specified location. The reason for the error may not be immediately clear. This would affect mainly users setting `'IPv6'` together with specific locations. However, the same thing would happen if the user specified an IP instead of a domain as the `Target`. The confusion could be avoided by setting a more specific error message, which would hint to the user that the choice of IP version is responsible for no probes found.

2.2.1.2 Selecting IP version within tags

The second approach involving tags is based on the idea that the supported IP version is a "property" of the probe's connection, similar to, e.g., datacenter/eyeball, or 4g/5g/fiber so users are not "configuring" the measurement, they simply select the probes that have the "right" connection type. This concept lacks substantive coherence because a single connection may support both versions and does so in most cases. Most ISPs that support IPv6 do so in addition to IPv4 over the same, single connection, so the supported version is not the connection's "property".

However, this approach would make the error "no suitable probes found" clearer for users to see that it affects available probes since the IP version would be a part of the `location` filter.

It would also allow users to combine IPv4 and IPv6 results in a single measurement, even though this feature is not supported for any other feature. If it was needed, it could be implemented in a more generic approach. It would allow combining results for all cases where applicable, for example querying multiple DNS record types at once.

One of the disadvantages of this approach is that since a single probe may support both versions, using a tag implies an automatic "setting" being applied in addition to standard filtering. For example, using "IPv6" does not mean only to select probes that support IPv6, it means to select probes that support IPv6 *and force the measurement to use IPv6*. This is an inconsistency compared to how the tag system normally works. It would also create scenarios with unclear behavior.

Another disadvantage is that tags are more difficult to document and discover for users.

```
{
  "locations": [
    {
      "country": "PL"
    }
  ],
  "measurementOptions": {
      "ipVersion": "IPv6"
  }
}
```

Code listing 2.1 Selecting IP version – measurement options

Table 2.1 Measurement results if IPv4 is the default, and the user did not specify anything

Target \ Probe	IPv4 only	IPv6 only	IPv4 + IPv6
IPv4 only	✓	won't be selected	✓
IPv6 only	✗	won't be selected	✗
IPv4 + IPv6	✓	won't be selected	✓

```
{
  "locations": [
    {
      "country": "PL",
      "tags": [ "IPv6" ]
    }
  ]
}

// OR

{
  "locations": [
    {
      "magic": "PL+ipv6"
    }
  ]
}
```

Code listing 2.2 Selecting IP version – tags

2.2.2 Default IP version

Several options were considered for the default IP version of the measurement in case the `Target` is a hostname and the user does not choose a preferred IP version. The following tables illustrate the behavior that each option would trigger. The probe can support IPv4, IPv6, or both versions – this is represented in the columns. If the target supports a different version from the probe executing the measurement, the measurement will fail – this is signified by a red cross. If they support the version used for the measurement, it will succeed – this is signified by a green check mark. Each target can also support IPv4, IPv6, or both – that is represented in the rows.

Table 2.1 shows what it would look like if the default version were IPv4. In this case, unless the user explicitly states that they want to use IPv6, the IPv6 targets will always fail since the IPv6-only probes will not be selected, and with dual-stack probes, the IPv4 version of the command will be used. If the user sets a preferred IP version to be IPv6, IPv6-only and dual-stack probes will be selected for the measurement all the while the IPv6 version of the command will be used. This is the only way for IPv6-only targets to succeed.

Table 2.2 represents the state if "auto" were the default. It would mean that all probes would always be selected regardless of their supported IP version. If the user chose a target that does not support the IP version corresponding to that of the probe, the measurement would fail. The disadvantage of this approach is that as the majority of targets are IPv4-only and in the current Globalping version measurements for these targets regularly pass, this change would cause measurements with IPv6-only probes to fail. It would have an undesirable effect of measurements failing randomly since probes are being selected randomly. This nuisance is being taken care of by the first option with IPv4 as the default, where the user can simply choose not to choose IPv6-only probes for the measurements. The first option ensures that IPv4 targets will remain working without a change and regardless of the added support for IPv6.

Table 2.2 Measurement results if "auto" is the default, and the user did not specify anything

Target \ Probe	IPv4 only	IPv6 only	IPv4 + IPv6
IPv4 only	✓	✗	✓
IPv6 only	✗	✓	✓
IPv4 + IPv6	✓	✓	✓

Table 2.3 Measurement results if "auto" is the default, but we don't select IPv6-only probes unless explicitly enabled

Target \ Probe	IPv4 only	IPv6 only	IPv4 + IPv6
IPv4 only	✓	won't be selected	✓
IPv6 only	✗	won't be selected	✓
IPv4 + IPv6	✓	won't be selected	✓

The third option "auto" is being used as the default in Table 2.3, but unlike the second option, the IPv6-only probes would not be selected for measurements unless explicitly enabled. In the case of dual-stack probes, both IP versions would be tested, which conveniently eliminates one red cross from the table, compared to the table of the first option, where IPv4 is forced in the case of dual-stack probes. The disadvantage lies in the unpredictability that would arise if the users chose an IPv6-only target. The results would consist of a mix of failed measurements from IPv4-only probes and successful measurements from dual-stack probes.

After thorough consideration of the pros and cons of all three options, the first one emerged as the best choice. Having IPv4 as the default choice would ensure continuity of functionality for existing users without change. This consistency of behavior was also a priority for the project maintainers.

2.3 Probe's supported connection types

As was expressed in user requirement R2.1, a probe has to be able to detect its supported IP versions and report it back to the API. The API then knows to which probe to assign a measurement, so that its IP version corresponds to the target and ideally also to the IP version chosen by the user.

After researching how to detect whether a probe runs on IPv4, IPv6, or dual-stack, the first possible solution was formed. It would utilize an operating system module's method `os.networkInterfaces()` from Node.js. It returns an object containing network interfaces assigned a network address [28]. After retrieving the network interfaces available for the probe, the presence of both IPv4 and IPv6 would be checked. If at least one IP address associated with any network interface present in the system is IPv4, and at least one address is IPv6, the probe runs on dual-stack. If all the IP addresses are IPv4, it runs on IPv4 only, analogically for IPv6.

The challenge presented by this approach was, that the network interfaces also contained loopback interfaces. These interfaces are considered to be internal to the network and therefore are not relevant to a probe's IP mode determination. For this reason, the algorithm would need to be altered to accommodate this. It could be done by filtering out the loop-back interfaces before checking their IP addresses' family (IP version).

The second approach manifested itself after further familiarization with the code base. Currently, each probe has a status, referring to the availability of its communication via IPv4. The status manager directs status checks. They consist of periodical pinging of three reliable IP addresses. The test is successful if more than half of the targets are successfully pinged. The status, in this case, is ’ready’, otherwise it is ’ping-test-failed’. The API assigns measurements only to the probes with the status ’ready’. By simply adding a similar periodical ping test for reliable IPv6 targets, the status of both, IPv4 and IPv6 statuses could be tracked easily.

The first approach turned out not to be failproof because it does not guarantee the network’s ability to reach IPv6/IPv4 address even if it’s available within network interfaces (e.g. given IP version is not enabled on the router). The probe could then for example have IPv6 addresses within its interfaces, however, if the router that the probe is connected to was unable to communicate via IPv6, the communication would be effectively unfeasible. For this reason, the second approach will be used for the implementation.

2.3.1 Analysis of chosen approach

Accommodating status checks could be done in two ways. The first is to track IPv4 and IPv6 status as separate attributes within the status manager. The second is to keep status to represent the fact that the probe can communicate in any way, and add a whole new field called mode that would hold the information about the supported IP version. Its values would contain `'both'`, `'ipV4Only'`, `'ipV6Only'` and `'none'`.

It was decided to mix both options and keep the status field as is and instead of the mode field add two new fields – booleans `'supportsIPv4'` and `'supportsIPv6'`. The status is `'ready'` if at least one of the new fields is true. This approach was chosen because it would keep the simplicity of the code. This approach also maintains backward compatibility by keeping the event ’probe:status:update’ intact, and just adding new handlers for updating fields `'supportsIPv4'` and `'supportsIPv6'` – `'probe:supportsIPv4:update'` and `'probe:supportsIPv6:update'`. This way Globalping will work with older probes without a change while extending its functionality to IPv6.

2.3.2 Reliable IPv6 targets for ping test

In order to check the probe’s reachability via IPv6, reliable, globally reachable IPv6 targets are needed. They will be pinged periodically and the result of this test will be then reported to the API.

Currently, there are three IPv4 targets used for IPv4 ping test: `'ns1.registry.in'`, `'k.root-servers.net'` and `'ns1.dns.nl'`. After checking them, each of them supports IPv6 as well, the ping command will just use option `'-6'` instead of `'-4'`. For this reason, no new IPv6 targets are necessary.

2.4 Single connection to the API

The requirement R2.2 says that there should be only one connection to the API via either IPv4 or IPv6 and that the probe itself can select the protocol. It also does not affect the user’s ability to run tests of the other type.

Originally, there were two options regarding communication between the API and the probe that would add IPv6 support. They would have either one connection or two at the same time.

If there was only one connection, its protocol would be decided by the networking stack of the operating system of the probe. Via this connection, the probe would report what IP versions it supports. For example, the probe running on dual-stack would connect to the API via IPv4. The probe would report its supported IP versions also via IPv4.

If the probe had two independent connections to the API, one socket would be created through an IPv4 connection and the other through an IPv6 connection. It would eliminate the need to detect the supported IP version since the existence of the connection means the probe supports the given IP version. However, two independent connections to the probe would be too complicated to maintain. For this reason, the project maintainers decided to only keep one connection between the probe and the API. This requirement is therefore considered to be met prior to my engagement in the project.

2.5 Blacklist matching

The requirement R1.3 states that the anti-abuse system must support IPv6, ensuring that Globalping stays secure for the probes' vendors. Currently, a target is being looked up in three IPv4 blacklists and three domain blacklists. If it matches any of the addresses listed in the malware blacklists, an error message is displayed to the user, and the measurement process is halted. The blacklists are being constantly updated. Matching of the IPv6 targets should function in a similar way.

Since the domains are not resolved before blocking, there is no need to add a new domain blacklist for IPv6 domains specifically. Only the list of malicious IPv6 addresses is needed. After researching reputable sources, three blacklists were found – spamhaus, bogons, and blocklist.de [29]. All of these have been chosen to be added for preventive security checks in Globalping.

Spamhaus blacklist includes IPv6 netblocks, that are "hijacked" or leased by professional spam or cyber-crime operations, directly allocated by an established Regional Internet Registry (RIR) or National Internet Registry (NIR) [30].

Bogons list includes IPv6 bogons, which are invalid addresses for public internet traffic [31]. They are legitimate addresses, however, they have not been allocated or delegated by the Internet Assigned Numbers Authority (IANA) or a delegated Regional Internet Registry (RIR) [32].

Blocklist.de list `'all.txt'`, contains mainly IPv4 but also IPv6 addresses that have been sending spam specifically to a fraud and abuse specialist, whose servers are often attacked on SSH, Mail-Login, File Transfer Protocol (FTP), Web server and other services [33].

2.6 Target validation and probe filtering

Requirement R1.1 states that a user must be able to specify a IPv6 address in all relevant inputs. In order to target IPv6-enabled hosts, the validation criteria need to be relaxed. Currently, the target is being validated against a rigid schema allowing public IPv4 addresses only. The IPv addresses need to be allowed by changing the schema in every type of test where necessary – traceroute, mtr, HTTP, and ping. In order to satisfy the requirement R1.1.1, the current filtering of probes for measurements will need to be adjusted as well.

Private IP addresses are currently validated against private IPv4 address blocks. Therefore private or reserved IPv6 address blocks must be included in the validation process.

2.6.1 Private IPv6 addresses

Checking if an address is private is up to the API side as well as up to the probe side. If a user inputs an IP address as the target, it is only checked on the API side, however, if a user inputs a domain, it is sent to a probe and it is checked only after the DNS is resolved. This check prevents users from testing targets in the local network since it is possible to set a private IP address for a public domain. Private IPv6 addresses are listed in IANA IPv6 Special-Purpose Address Registry [34] together with their RFC files. These files expand on why these addresses are included in the list.

Not all addresses listed in IANA are private. Some are globally reachable, however, they are reserved for a special purpose. Some of these should be allowed because it is beneficial for users to be able to test them. Generally, unless the usefulness of allowing the address is obvious, it is blocked.

Address block IPv4-mapped Address is listed as a special purpose address registry, and it is not globally reachable, however, it will not be blocked. It allows an IPv4 address to be represented as an IPv6 address. It serves as a middleman for communication between IPv6-only environments and IPv4 addresses [35]. Therefore, these addresses do not exist directly on the internet. It is not desirable to block this compatibility technology because it benefits users to be allowed to test these addresses. Similarly, address blocks IETF Protocol Assignments and Benchmarking will be allowed because they serve as mapping and translating protocols.

The only blocked address block not from the IANA list is ff00::/8. It is blocked because these are multicast addresses [36]. Multicast is a communication technology where one host sends data to multiple hosts at once by sending an IP packet to a special IP address that identifies the group[37]. For this reason, allowing it does not appear justified.

2.6.2 Filtering probes

It was decided that if the target is an IP address, probes not supporting the version of this address will be automatically filtered out. If the target is a domain, the user will have a choice of IP version, in which the measurement will be carried out. If the user does not choose a preferred IP version, IPv4 will be used as a default. If the user specifies a different IP version than the IP version of the target, the measurement will be sent to the probes supporting the preferred IP version and it will fail.

Chapter 3
Implementation

The following chapter delves into the technical realization of the user requirements based on the analysis included in the previous chapter. It is divided into sections according to the logical units. There is a section dealing with the integration of field `ipVersion`, another with creating fields `isIPv4Supported` and `isIPv4Supported` together with the events used for reporting their updates from probes to the API. Then the separate section describes the validation of fields `target` and `resolver` including the checks for malicious or private addresses. The next section focuses on probe filtration by `ipVersion` and the last focuses on the changes in Globalping demo.

3.1 User's choice of IP version

As explained in the chapter Analysis and Design, the `ipVersion` field was supposed to have values `'IPv4'`, `'IPv6'`, and `'auto'`. However, in the end, it has values 4 and 6. If nothing is provided, it defaults to 4. If the `target` is an IP address, it is forbidden for a user to provide an `ipVersion`, however internally it is automatically set to the IP version of the `target`.

In the course of the implementation process, it became obvious that the type incompatibility of the DNS measurement options and measurement options of other commands caused by adding an option `ipVersion` to all commands but DNS would disrupt the current design. After further research, it was found that the dig command offers a choice of IP version to use. After consulting with the project maintainers, it was agreed upon to add the `ipVersion` option to DNS as well.

```
type TracerouteTest = {
    protocol: 'ICMP' | 'TCP' | 'UDP';
    port: number;
    ipVersion?: 4 | 6;
};
```

Code listing 3.1 Measurement options type containing ipVersion

Firstly, the `ipVersion` field was added to each test type. It is shown in Code listing 3.1 – it can be either 4, 6, or none.

```
1  const allowedIpVersions = [ 4, 6 ];
2
3  export const ipVersionSchema = Joi.number().when(Joi.ref('...target'), {
4      is: Joi.custom(joiValidateDomain()),
5      then: Joi.valid(...allowedIpVersions).optional().default(DEFAULT_IP_VERSION),
6      otherwise: Joi.when(Joi.ref('...target'), {
7          is: Joi.string().ip({ version: [ 'ipv6' ], cidr: 'forbidden' }),
8          then: Joi.forbidden().default(6),
9          otherwise: Joi.forbidden().default(DEFAULT_IP_VERSION),
10     }),
11 }).messages({
12     'any.only': 'ipVersion must be either 4 or 6',
13     'any.unknown': 'ipVersion is not allowed when target is not a domain',
14 });
15
16 export const ipVersionDnsSchema = Joi.number().when(Joi.ref('resolver'), {
17     is: Joi.custom(joiValidateDomain()),
18     then: Joi.valid(...allowedIpVersions).optional().default(DEFAULT_IP_VERSION),
19     otherwise: Joi.when(Joi.ref('resolver'), {
20         is: Joi.string().ip({ version: [ 'ipv6' ], cidr: 'forbidden' }),
21         then: Joi.forbidden().default(6),
22         otherwise: Joi.forbidden().default(DEFAULT_IP_VERSION),
23     }),
24 }).messages({
25     'any.only': 'ipVersion must be either 4 or 6',
26     'any.unknown': 'ipVersion is not allowed when resolver is not a domain',
27 });
28 ...
29 export const pingSchema = Joi.object({
30     packets: Joi.number().min(1).max(16).default(COMMAND_DEFAULTS.ping.packets),
31     ipVersion: ipVersionSchema,
32 }).default().messages(schemaErrorMessages);
```

Code listing 3.2 Joi validation of ipVersion

In Code listing 3.2 can be seen an added validation for field `ipVersion`. On line 31 it is included in the schema of the options for test type ping, but it was added to the options of each test type similarly. Above there is the detailed schema described in terms of Joi validation library. It says that if the `target` is a domain, the value provided by the user can have any of the allowed IP versions, that is 4 or 6, defaulting to 4. Otherwise, if the `target` is an IP address, it is forbidden for a user to provide an `ipVersion`. It either raises an error or if the user has not provided an `ipVersion`, it is set automatically according to the IP version of the `target`. For DNS test type it works similarly, however instead of the `target` it depends on the `resolver` field.

```
export const argBuilder = (options: PingOptions): string[] => {
    const args = [
-       '-4',
+       `-${options.ipVersion}`,
        '-O',
        [ '-c', options.packets.toString() ],
        [ '-i', '0.5' ],
        [ '-w', '10' ],
        options.target,
    ].flat();
    return args;
};
```

Code listing 3.3 Ping measurement arguments with integrated choice of IP version

On the probe side, where the arguments for each command are built, is the value of `ipVersion` field used instead of the fixed `'-4'` option for each test type. In the example Code listing 3.3 is the argument builder that returns a collection of options. These will be then used as options for command `ping`.

3.2 Target validation

```
+   const DEFAULT_IP_VERSION = 4;

-   export const globalIpOptions = { version: [ 'ipv4' ], cidr: 'forbidden' };
+   export const globalIpOptions = { version: [ 'ipv4', 'ipv6' ], cidr: 'forbidden' };
```

Code listing 3.4 Joi validation constants allowing IPv6 addresses and setting a default IP version

In order to allow IPv6 addresses as `target`, it was sufficient to add `'ipv6'` into constant `globalIpOptions` that was already used for `target` validation. In the same file was also added a constant for the default IP version, which could be easily changed in the future in case IPv6 gains traction over the IPv4 protocol.

3.3 Private IPs

```
+   privateBlockList.addSubnet('::', 128, 'ipv6');
+   privateBlockList.addSubnet('::1', 128, 'ipv6');
+   privateBlockList.addSubnet('64:ff9b:1::', 48, 'ipv6');
+   privateBlockList.addSubnet('100::', 64, 'ipv6');
+   privateBlockList.addSubnet('2001::', 32, 'ipv6');
+   privateBlockList.addSubnet('2001:10::', 28, 'ipv6');
+   privateBlockList.addSubnet('2001:20::', 28, 'ipv6');
+   privateBlockList.addSubnet('2001:db8::', 32, 'ipv6');
+   privateBlockList.addSubnet('2002::', 16, 'ipv6');
+   privateBlockList.addSubnet('fc00::', 7, 'ipv6');
+   privateBlockList.addSubnet('fe80::', 10, 'ipv6');
+   privateBlockList.addSubnet('ff00::', 8, 'ipv6');
...
if (ipVersion === 6) {
-   throw new Error('IPv6 not supported');
+   return privateBlockList.check(ip, 'ipv6');
}
```

Code listing 3.5 Checks for private address blocks

For both the API and probe repository the checks for private IP addresses were extended. It is shown in the Code listing 3.5. The subnets are private blocks for which there is no justification to allow them. On the probe side in the function that returns default DNS servers shown in Code listing 3.6 the IPv6 addresses are also allowed and checked if they are private. Because of the difference in the syntax the given address is first stripped of a port number by matching an IPv6 address with a port number and then by matching an IPv4 address with a port number. These servers are then reported to the API using an event designated specially for this purpose.

```
export const getDnsServers = (getServers: () => string[] = dns.getServers): string[] => {
    const servers = getServers();

    return servers
-   // Filter out ipv6
-   .filter((addr: string) => {
-       const ipv6Match = /^\[(.*)]/g.exec(addr); // Nested with port
-       return !isIPv6(addr) && !isIPv6(ipv6Match?.[1] ?? '');
-   })
    // Hide private ips
    .map((addr: string) => {
-       const ip = addr.replace(/:\d{1,5}$/, '');
+       const ip_ = addr.replace('[', '').replace(/]:\d{1,5}$/, ''); // removes port ipv6
+       const ip = isIPv6(ip_) ? ip_ : ip_.replace(/:\d{1,5}$/, ''); // removes port not ipv6
        return isIpPrivate(ip) ? 'private' : addr;
    });
};
```

Code listing 3.6 Function getDnsServers allowing IPv6 servers

3.4 Blacklists, malicious IPv6 addresses and domains

One of the requirements states that the checks for malicious addresses must also be able to validate IPv6 addresses. The blacklists containing either malicious IPv6 addresses or address blocks were added as source lists in the Code listing 3.7 so that they can be automatically updated. The address provided by the user is then looked up in these blacklists.

```
export const sourceList = [
    'https://osint.digitalside.it/Threat-Intel/lists/latestips.txt',
    'https://raw.githubusercontent.com/firehol/blocklist-ipsets/master/firehol_level2.netset',
    'https://raw.githubusercontent.com/stamparm/ipsum/master/levels/2.txt',
+   'https://www.spamhaus.org/drop/dropv6.txt',
+   'https://lists.blocklist.de/lists/all.txt',
+   'https://www.team-cymru.org/Services/Bogons/fullbogons-ipv6.txt',
];
```

Code listing 3.7 IPv6 blacklists

Since some of the blacklists contain address blocks in CIDR format, the parsing for these needed to be added. It is shown in the Code listing 3.8. The body, which is the content of one of the blacklists, is split into lines. Any additional information after the first space character is removed, and then it is checked if the network prefix part of the CIDR notation is a valid IP address, and if it is, the address block in CIDR notation is added to the accumulative list of all malicious addresses and address blocks.

The address provided by the user is then matched against this list. The list is iterated and checked for a match. In case of matching against a malicious address block, the function `isContainedWithinSubnet` is used. It utilizes the methods `match` and `parseCIDR` from ipaddr.js, as can be seen in Code listing 3.9.

```
-   const result = body.split(/r?\n/).filter(l => validator.isIP(l));
+   const result = body.split(/\r?\n/).map(l => l.split(' ')[0] ?? '')
+               .filter(l => validator.isIP(l.split('/')[0] ?? ''));
```

Code listing 3.8 Parsing of blacklists in CIDR format

```
function isContainedWithinSubnet (target: string, ipListArray: Set<string>): boolean {
    if (!validator.isIP(target)) {
        return false;
    }

    const addr = ipaddr.parse(target);

    for (const ip of ipListArray) {
        if (ip.includes('/') && addr.kind() === ipaddr.parse(ip.split('/')[0] ?? '').kind()) {
            if (addr.match(ipaddr.parseCIDR(ip))) {
                return true;
            }
        }
    }

    return false;
}
```

■ **Code listing 3.9** Check for the address being contained within the subnet

3.5 StatusManager

Previously, the status manager of the probe managed the status of the IPv4 connection. When adding support for probes supporting IPv6, part of the job of the `status` field was taken by two new fields – `isIPv4Supported` and `isIPv6Supported` – that can be seen in Code listing 3.10. They indicate the probe's status of IPv4 and IPv6 connection respectively. The `status` remained unchanged, however now it only holds the information about whether a probe supports at least one of IPv4 or IPv6.

```
export class StatusManager {
    private status: 'initializing' | 'ready' | 'ping-test-failed' | ...;
+   private isIPv4Supported: boolean = false;
+   private isIPv6Supported: boolean = false;
...
}
```

■ **Code listing 3.10** Status manager's new fields isIPv4Supported and isIPv6Supported

The fields `isIPv4Supported` and `isIPv6Supported` were also added to the database into the table `gp_adopted_probes` that contains information about the probes, as can be seen in Code listing 3.11.

```
CREATE TABLE IF NOT EXISTS gp_adopted_probes (
  id INT(10) UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  ip VARCHAR(255) NOT NULL,
  lastSyncDate DATE NOT NULL,
  status VARCHAR(255) NOT NULL,
+     isIPv4Supported BOOLEAN,
+     isIPv6Supported BOOLEAN,
  country VARCHAR(255) NOT NULL,
  ...
```

■ **Code listing 3.11** Database table containing information about probes has new fields

3.6 Events

The values of fields `isIPv4Supported` and `isIPv6Supported` are being periodically updated and then reported back to the API. For this reason, two new events were created – `'probe:isIPv4Supported:update'` and `'probe:isIPv6Supported:update'`. Probes emit these events after retrieving results of periodical ping tests shown in Code listing 3.12. Each round, ping test for IPv4 and IPv6 is carried out. On line 14 function `'runTest` updates `status` to `'ready'` if at least one of the ping tests was successful, otherwise it updates it to `'ping-test-failed'`. After updating the `status`, it updates both – `isIPv4Supported` and `isIPv6Supported`, resulting in emitting of the events `'probe:isIPv4Supported:update'` and `'probe:isIPv6Supported:update'`.

The API handles these events by updating the values of `isIPv4Supported` and `isIPv6Supported` fields as shown in Code listing 3.13.

```
1  public updateIsIPv6Supported (isIPv6Supported : boolean) {
2      this.isIPv6Supported = isIPv6Supported;
3      this.sendIsIPv6Supported();
4  }
5  ...
6  public sendIsIPv6Supported () {
7      this.socket.emit('probe:isIPv6Supported:update', this.isIPv6Supported);
8  }
9  ...
10 private async runTest () {
11     const resultIPv4 = await this.pingTest(4);
12     const resultIPv6 = await this.pingTest(6);
13
14     if (resultIPv4 || resultIPv6) {
15         this.updateStatus('ready');
16     } else {
17         this.updateStatus('ping-test-failed');
18     }
19
20     this.updateIsIPv4Supported(resultIPv4);
21     this.updateIsIPv6Supported(resultIPv6);
22 ...
23 }
```

Code listing 3.12 New events for updating of new fields isIPv4Supported and isIPv6Supported

```
export const handleIsIPv4SupportedUpdate = (probe: Probe) => {
    probe.isIPv4Supported = isIPv4Supported;
};

export const handleIsIPv6SupportedUpdate = (probe: Probe) => {
    probe.isIPv6Supported = isIPv6Supported;
};
```

Code listing 3.13 Handlers for updating of new fields isIPv4Supported and isIPv6Supported

3.7 Probe filtration

The whole request body of the created measurement is sent to the probes chosen to carry out the measurement. Before that, the API has to filter the probes to which it will assign the measurement. Originally, it filtered all the currently online probes, meaning that they had `'ready'` status, and then it further filtered the online probes according to the user, choice of `location`, and `limit`. With the new user choice of preferred IP version of the measurement, it became insufficient to filter probes only according to the `'ready'` status, which indicates that the probe supports at least one IP version. In the Code listing 3.14 on line 18 contains code, which further narrows down the online probes into those that support the preferred IP version. After that, filtration according to `location` and `limit` proceeds without change.

```
 1  public filterByIpVersion (probes: Probe[], ipVersion: 4 | 6) : Probe[] {
 2      let filteredProbes = probes;
 3
 4      if (ipVersion === 4) {
 5          filteredProbes = probes.filter(probe => probe.isIPv4Supported);
 6      } else if (ipVersion === 6) {
 7          filteredProbes = probes.filter(probe => probe.isIPv6Supported);
 8      }
 9
10      return filteredProbes;
11  }
12  ...
13  public async findMatchingProbes (userRequest: UserRequest): Promise<{...}> {
14  ...
15  +   const preferredIpVersion = userRequest.measurementOptions?.ipVersion ?? 4;
16      const connectedProbes = (await this.fetchProbes()).filter(probe => probe.status === 'ready');
17  ...
18  +   const filtered = this.probesFilter.filterByIpVersion(connectedProbes, preferredIpVersion);
19  ...
20  // other filters further narrowing the "filtered" collection
21  }
```

Code listing 3.14 Filtration of probes by ipVersion

3.8 Demo

The Globalping platform has its demo version that simulates the creation of the measurements and sending it to the probes. The front-end of the demo needed to include the field `ipVersion` too. It was added to the HTML for each measurement type, as shown in Code listing 3.15. It offers a combobox with values of allowed IP versions. The values are obtained from the `getIpVersionArray` function that returns values 4 and 6. The fields `isIPv4Supported` and `isIPv6Supported` are also displayed in the probe information section, provided the user has an admin or a system key.

```
<div v-if="['ping', 'mtr', 'dns', 'traceroute', 'http'].includes(query.type)" ...>
    <label for="query_ipVersion" class="col-sm-2 col-form-label">ipVersion</label>
...
        <select v-model="query.ipVersion" name="query_ipVersion" id="query_ipVersion" ...>
            <option v-for="ipVersion in getIpVersionArray()" :value="ipVersion">
                {{ ipVersion }}
...
```

Code listing 3.15 Adding of field for ipVersion to demo

Chapter 4

Testing and Documentation

The following chapter looks into the automated tests added to the API as well as to the probe repository. A JavaScript framework called Mocha is used for asynchronous testing and mocking of HTTP responses. It refers to a single test by “it” followed by a description of its expected behavior in natural language. In the sections below, the testing process for each implemented logical unit will be described and visualized by code snippets of the test examples.

4.1 User’s choice of IP version

Testing the field `ipVersion` that was added to the measurement options of each measurement type included checking that the `ipVersion` could be provided by the user only if the `target` was a domain. In the Code listing 4.1 is shown a test for ping measurement where the provided `target` is an IPv4 address the `ipVersion` is provided as well. This test should therefore fail.

```
it('should fail ip version provided when target is an ip', () => {
    const input = {
        type: 'ping',
        target: '1.1.1.1',
        measurementOptions: {
            packets: 1,
            ipVersion: 4,
        },
    };

    const valid = globalSchema.validate(input, { convert: true });

    expect(valid.error).to.exist;
    expect(valid.error!.message).to.equal('ipVersion is not allowed when target is not a domain');
});
```

Code listing 4.1 Test of ipVersion validation – forbidden option when the target is not a domain

Furthermore, if the `target` (or `resolver` in case of DNS measurement) is an IP address, the `ipVersion` is automatically filled by the API’s validation schema. In Code listing 4.2 is displayed a test case where the provided `resolver` for the DNS measurement is an IPv6 address and the expected `ipVersion` after validation should equal to 6. There are similar unit tests for each measurement type and each combination of `target` values and `ipVersion` values.

```
it('should pass (deep equal) ip version 6 automatically selected when resolver is ipv6', () => {
    const input = {
        type: 'dns',
        target: 'abc.com',
        measurementOptions: {
            trace: false,
            resolver: '2001:41f0:4060::',
            protocol: 'UDP',
            port: 53,
            query: {
                type: 'A',
            },
        },
    };

    const desiredOutput = {
        type: 'dns',
        target: 'abc.com',
...
        measurementOptions: {
            resolver: '2001:41f0:4060::',
            ipVersion: 6,
...
        },
    };

    const valid = globalSchema.validate(input, { convert: true });
    expect(valid.error).to.not.exist;
    expect(valid.value).to.deep.equal(desiredOutput);
});
```

■ **Code listing 4.2** Test ipVersion validation – should be automatically detected when the resolver is an IP address

On the probe's side the unit tests consisted mainly of the command argument checks. In the Code listing 4.3 can be seen that if the `ipVersion` value is 6, the argument builder should add option `'-6'` to the traceroute command that will be then executed.

```
it('should set -6 flag', () => {
    const options = {
        type: 'traceroute' as TraceOptions['type'],
        target: 'google.com',
        port: 80,
        protocol: 'TCP',
        inProgressUpdates: false,
        ipVersion: 6,
    };

    const args = argBuilder(options);
    expect(args[0]).to.equal('-6');
});
```

■ **Code listing 4.3** Traceroute measurement arguments test

4.2 Target validation

Validating the `target` in case of IPv6 address was provided required unit tests for invalid format, checks for private addresses and malware matching. In the Code listing 4.4 is provided target containing sequence ":::", which is invalid, and therefore the test expects an error. The similar tests were done for the DNS's `resolver`.

```
it('should fail (target: invalid ipv6 format)', () => {
    const input = {
        type: 'ping',
        target: '2a06:e80:::3000:abcd:1234:5678:9abc:def0',
    };

    const valid = globalSchema.validate(input, { convert: true });

    expect(valid.error).to.exist;
    expect(valid.error!.message).to.equal('"target" does not match any of the allowed types');
});
```

Code listing 4.4 Test of target validation – invalid IPv6 format

The Code listing 4.5 shows test covering the function `joiValidateTarget` that should throw an error since the provided IP address is private.

```
it('should fail (ip type) (private ipv6)', () => {
    const input = '64:ff9b:1::1a2b:3c4d';

    let result: string | Error = '';

    try {
        result = joiValidateTarget('ip')(input);
    } catch (error: unknown) {
        if (error instanceof Error) {
            result = error;
        }
    }

    expect(result).to.be.instanceof(Error);
});
```

Code listing 4.5 Testing private IPv6 address check

The Code listing 4.6 covers testing of the function `validateIp` that detects that the provided IP address is contained within one of the IPv6 blacklists.

```
it('should fail (ipv6 present on the list)', () => {
    const isValid = validateIp('2803:5380:ffff::386');

    expect(isValid).to.be.false;
});
```

Code listing 4.6 Test for presence of IPv6 address on blacklist

The function `validateIp` from the previous test is used as a check in the schema validation. If it says that the address is blacklisted, the schema should throw an error, as can be seen in Code listing 4.7 where the measurement option `resolver` caused the validation to fail.

```
it('should fail (blacklisted ipv6 resolver)', () => {
    const input = {
        type: 'dns',
        target: 'abc.com',
        measurementOptions: {
            resolver: '2803:5380:ffff::386',
        },
    };

    const valid = globalSchema.validate(input, { convert: true });

    expect(valid.error).to.exist;
    expect(valid.error!.message).to.equal('Provided address is blacklisted.');
});
```

Code listing 4.7 Test for measurement schema – blacklisted IPv6 resolver

4.3 Private IPs

Apart from validation, the addresses were checked for being private in the function `getDnsServers` as well. The Code listing 4.8 shows that one of the addresses provided in the `input` on line 2 is private, and therefore should be masked.

```
1  it('should mask private ipv6', () => {
2      const input = [
3          '2001:db8:fa34::',
4          '1.1.1.1',
5      ];
6
7      const servers = getDnsServers(client(input));
8
9      expect(servers.length).to.equal(2);
10
11     expect(servers).to.deep.equal([
12         'private',
13         '1.1.1.1',
14     ]);
15 });
```

Code listing 4.8 Test for masking private IPv6 addresses

4.4 Probe filtration

For example, the probe filtration's proper functioning was tested using the test shown in Code listing 4.9. Firstly, four mocked probes are built and added to the list. All of them are located in Europe, each in a different country. The first is located in Great Britain and does not support any IP version, since it does not have `'ready'` status. The probe located in the Czech Republic supports both IP versions and the other two probes support either IPv4-only or IPv6-only. The function `findMatchingProbes` is called on line 13 with measurement options demanding a maximum of four probes located in Europe supporting IPv6. It returns 2 probes in total, the one located in CZ that supports both and the one that supports IPv6 only

```
it('should find only probes supporting IPv6', async () => {
    const probes: Array<DeepPartial<Probe>> = [
        // buildProbe(..., status, isIPv4Supported, isIPv6Supported)
        await buildProbe('s-1', { continent: 'EU', country: 'GB' }, 'ping-failed', false, false),
        await buildProbe('s-2', { continent: 'EU', country: 'CZ' }, 'ready', true, true),
        await buildProbe('s-3', { continent: 'EU', country: 'PL' }, 'ready', true, false),
        await buildProbe('s-4', { continent: 'EU', country: 'AU' }, 'ready', false, true),
    ];

    fetchProbesMock.resolves(probes as never);

    const { onlineProbesMap, allProbes } =
    await router.findMatchingProbes({ locations: [{ continent: 'EU', limit: 4 },],
                                    measurementOptions: { ipVersion: 6 } });
...
    expect(allProbes.length).to.equal(2);
    expect(allProbes.filter(p => p.location.country === 'CZ').length).to.equal(1);
    expect(allProbes.filter(p => p.location.country === 'AU').length).to.equal(1);
});
```

Code listing 4.9 Probe filtration test by ipVersion

4.5 StatusManager

Status manager's new fields `isIPv4Supported` and `isIPv6Supported` together with the events used for reporting their updated values were tested using integration tests with mocked responses. The Code listing 4.10 shows that the mocked ping response on the first two calls will return a failure on lines 6 and 7. Since there are 6 ping calls in total – first 3 for checking if IPv4 is available and other 3 for IPv6. If more than half of these pings fail (meaning 1 or 0 out of 3 succeeds), the corresponding isIPvXSupported filed is updated to false – in this case, the `isIPv4Supported` field on line 14.

```
1  it('should update the status during regular checks, different values for ipv4 and ipv6', async () => {
2      const statusManager = initStatusManager(socket, pingCmd);
3
4      await statusManager.start();
5  ...
6      pingCmd.onCall(1).rejects({ stdout: 'host not found' });
7      pingCmd.onCall(2).rejects({ stdout: 'host not found' });
8      await sandbox.clock.tickAsync(11 * 60 * 1000);
9      expect(statusManager.getStatus()).to.equal('ready');
10     expect(statusManager.getIsIPv4Supported()).to.equal(false);
11     expect(statusManager.getIsIPv6Supported()).to.equal(true);
12     expect(socket.emit.callCount).to.equal(6);
13     expect(socket.emit.args[3]).to.deep.equal([ 'probe:status:update', 'ready' ]);
14     expect(socket.emit.args[4]).to.deep.equal([ 'probe:isIPv4Supported:update', false ]);
15     expect(socket.emit.args[5]).to.deep.equal([ 'probe:isIPv6Supported:update', true ]);
16 }
```

Code listing 4.10 Test of events being emitted properly updating the new fields isIPv4Supported and isIPv6Supported

Other required integration tests included proper handling of these update events from probes. In Code listing 4.11 mocked probe emits updates for each field on lines 2 and 3, and then the API should internally update these probe's fields as well. On line 6 the request for listing all probes is retrieved and then the the mocked probe's supported IP versions are compared.

```
it('should handle isIPv4Supported and isIPv6Supported events from probe', async () => {
    probe.emit('probe:isIPv4Supported:update', true);
    probe.emit('probe:isIPv6Supported:update', false);
    await waitForProbesUpdate();

    await requestAgent.get('/v1/probes?adminkey=admin').send()
        .expect(200).expect((response) => {
            expect(response.body[0]).to.deep.include({
                isIPv4Supported: true,
                isIPv6Supported: false,
                ...
            });

            expect(response).to.matchApiSchema();
        });
});
```

Code listing 4.11 Test of handling updates of fields isIPv4Supported and isIPv6Supported

4.6 Command output parsing

The most challenging part was testing the command output parsing to satisfy the requirement R1.1.2. Firstly, the raw output for all commands needed to be obtained. The commands had to use option `-6` meaning that the device as well as the router had to support IPv6 connections. Since the author did not have access to IPv6 connection, the challenge was resolved by using remote access to the server provided by the supervisor to retrieve these outputs manually. The partial output of the command traceroute is shown in the Code listing 4.12.

```
traceroute to google.com (2a00:1450:4026:808::200e), 20 hops max, 80 byte packets
 1  fe80::%eth0 (fe80::%eth0)  8.385 ms  8.406 ms
...
11  2001:4860:0:1::3e57 (2001:4860:0:1::3e57)  2.066 ms  2.049 ms
12  hem08s10-in-x0e.1e100.net (2a00:1450:4026:808::200e)  1.268 ms  1.285 ms
```

Code listing 4.12 Traceroute command parsing test – raw output

This raw traceroute output should be then parsed into the JSON format shown in Code listing 4.13. Similar tests were created for each command with various sets of measurement options.

```
{
    "testId": "test",
    "measurementId": "measurement",
    "result": {
        "status": "finished",
        "resolvedAddress": "2a00:1450:4026:808::200e",
        "resolvedHostname": "hem08s10-in-x0e.1e100.net",
        "hops": [
        ...
        ],
        "rawOutput": "traceroute to google.com (2a00:1450:4026:808::200e), 20 ..."
    }
}
```

Code listing 4.13 Traceroute command parsing test

The parsing of the addresses needed to be altered. The original regular expression showed in Code listing 4.14 on line 1 did not correctly match the IPv6 addresses, therefore it was replaced by a regular expression on line 2. The new one matches IPv4 as well as IPv6 addresses. Added `(?:::)?` part allows for zero or one occurrence of `"::"` sequence within the IPv6 address.

```
1  -  const reHost = /(\S+)\s+\((?:((?:\d+\.){3}\d+)|([\da-fA-F:]))\)/;
2  +  const reHost = /(\S+)\s+\((?:((?:\d+\.){3}\d+)|([\da-fA-F:]+(?:::)?[\da-fA-F:]+))\)/;
```

Code listing 4.14 Regular expression for parsing resolver matches IPv6 adresses

4.7 Documentation

The Globalpig platform uses OpenAPI for documentation. The Code listing 4.15 shows documentation of field `ipVersion`, that was added to the measurement options for each measurement type. The DNS's `ipVersion` option refers to the `resolver` instead of the `target`.

```
ipVersion:
  type: integer
  description: The IP protocol to use. Only allowed, if the target is a host name.
  default: 4
  enum:
    - 4
    - 6
```

Code listing 4.15 Documenting the ipVersion option

Code listing 4.16 and Code listing 4.17 show edits of the documentation for fields `target` and `resolver` respectively. Both now allow IPv6 addresses.

```
 MeasurementTarget:
   type: string
   description: |
     A publicly reachable measurement target.
-    Typically a hostname or an IPv4 address, depending on the measurement `type`.
+    Typically a hostname, an IPv4 address or an IPv6 address, depending on...
```

Code listing 4.16 Documenting the target options including IPv6

```
 MeasurementResolver:
   description: A DNS resolver to use for the query. Defaults to the probe system resolver.
   anyOf:
     - type: string
     format: ipv4
     description: The IPv4 address of the resolver.
+    - type: string
+    format: ipv6
+    description: The IPv6 address of the resolver.
     - type: string
     format: hostname
     description: The Fully Qualified Domain Name (FQDN) of the resolver.
```

Code listing 4.17 Documenting the resolver options including IPv6

Conclusion

In conclusion, extending the Globalping platform's functionality to accommodate IPv6 targets and allow probes running on IPv6-only networks into its network is an imperative step in addressing the challenges stemming from the gradual depletion of IPv4 address space. The adaptation enables the users to efficiently monitor, benchmark, debug, and optimize the performance and reachability of their global services while remaining relevant and ready to navigate the evolving landscape of IPv6 connectivity.

The thesis introduced readers to the domain in the Globalping chapter, providing a comprehensive overview of the endpoints provided by the API. These included creating a measurement – whether of type ping, dns, mtr, traceroute, or HTTP – listing all probes and getting a measurement by ID. Furthermore, the chapter explored the event-driven architecture of the platform, along with its containerization and load-balancing mechanism.

The focus of the Analysis and Design chapter centered on conducting an in-depth analysis of user requirements. It also dealt with comparisons of multiple options for setting a default `ipVersion`, determining the field's placement within the schema, determining a probe's available connection types, and finding a way to report it to the API. It culminated in the creation of a design for a solution that was accepted and implemented subsequently.

In the Implementation chapter, the design was realized by integrating the new measurement option `ipVersion` and incorporating Joi validation based on the `target` or, in the case of DNS, `resolver`. It showed a process of allowing IPv6 targets while implementing validations for private and blacklisted addresses. Additionally, it explained the implementation details of probe filtering based on the `ipVersion` that they support demanded by the measurement options.

The subsequent Testing and Documentation chapter provided an overview of the automated unit and integration tests developed to ensure the proper functioning of all newly implemented features. This encompassed tests for the accurate evaluation of private and blacklisted addresses, reliable filtering of the probes, and also proper parsing of raw command outputs – a task that presented notable challenges. Moreover, the chapter addressed the comprehensive updates made to Globalping's OpenAPI documentation, ensuring clarity on all endpoints, their responses, and parameters.

To sum it up, all the goals that were set in the beginning were successfully met. The thesis documented all work undertaken, providing valuable insights for anyone embarking on similar ventures to extend the functionality of their platforms with IPv6 support. The newly added functionality, accompanied by rigorous testing on both the API and probe sides, was included in the merge request that awaits approval from project maintainers. Notably, while the thesis focused on the backend processing of IPv6 targets and probes, successful deployment necessitates edits in configuration files for supporting communication via IPv6 on the network level, so that the probes update system and the load balancer function properly.

Bibliography

1. *Globalping – Internet and web infrastructure monitoring and benchmarking* [online]. 2012. [visited on 2024-02-29]. Available from: `https://www.jsdelivr.com/globalping`.
2. *jsDelivr – A free, fast, and reliable CDN for JS and open source* [online]. 2012. [visited on 2024-02-29]. Available from: `https://www.jsdelivr.com/docs/api.globalping.io`.
3. *ping | Microsoft Learn* [online]. 2023. [visited on 2024-03-19]. Available from: `https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/ping`.
4. LIU, Dale; BARBER, Brian; DIGRANDE, Luigi. CHAPTER 4 – Configuring Cisco Routers. In: LIU, Dale; BARBER, Brian; DIGRANDE, Luigi (eds.). *Cisco CCNA/CCENT Exam 640-802, 640-822, 640-816 Preparation Kit*. Boston: Syngress, 2009, pp. 145–168. ISBN 978-1-59749-306-2. Available from DOI: `https://doi.org/10.1016/B978-1-59749-306-2.00008-7`.
5. BOURNE, Kelly C. Chapter 17 – Performance Tuning. In: BOURNE, Kelly C. (ed.). *Application Administrators Handbook*. Boston: Morgan Kaufmann, 2014, pp. 296–320. ISBN 978-0-12-398545-3. Available from DOI: `https://doi.org/10.1016/B978-0-12-398545-3.00017-0`.
6. KUMAR, Rajesh. *Domain Name System (DNS) Tools – DevOpsSchool.com* [online]. 2023. [visited on 2024-03-19]. Available from: `https://www.devopsschool.com/blog/domain-name-system-dns-tools/`.
7. *The mtr command.* [online]. 2024. [visited on 2024-03-19]. Available from: `https://hostarmada.com/kb/ssh-and-linux/ssh-commands/the-mtr-command/#:~:text=The%20mtr(my%20traceroute)%20command,originated%20to%20the%20destination%20server.`.
8. *What is My Traceroute (MTR)? | Cloudflare* [online]. 2024. [visited on 2024-03-19]. Available from: `https://www.cloudflare.com/learning/network-layer/what-is-mtr/`.
9. *What is a Probe? | NETSCOUT* [online]. 2024. [visited on 2024-03-14]. Available from: `https://www.netscout.com/what-is/probe#:~:text=A%20system%20component%20that%20monitors,may%20also%20perform%20prevention%20actions.`.
10. *Everything You Need to Know About Network Probes – fortra.com* [online]. 2016. [visited on 2024-03-01]. Available from: `https://www.fortra.com/blog/everything-you-need-know-about-network-probes`.
11. STACK, M. *Event-Driven Architecture in Golang: Building complex systems with asynchronicity and eventual consistency*. Packt Publishing, 2022. ISBN 9781803232188. Available also from: `https://books.google.cz/books?id=ovKZEAAAQBAJ`.

12. *Introduction | Socket.IO* [online]. 2024. [visited on 2024-03-19]. Available from: `https://socket.io/docs/v3/#:~:text=Socket.IO%20is%20a%20library,be%20also%20run%20from%20Node.`.
13. *About – Redis* [online]. 2024. [visited on 2024-04-11]. Available from: `https://redis.io/about/`.
14. AKULOV, Dmitriy. *globalping/INFRASTRUCTURE.md at master · jsdelivr/globalping* [online]. 2023. [visited on 2024-04-11]. Available from: `https://github.com/jsdelivr/globalping/blob/master/INFRASTRUCTURE.md`.
15. *Redis Pub/Sub | Docs* [online]. 2024. [visited on 2024-04-26]. Available from: `https://redis.io/docs/latest/develop/interact/pubsub/`.
16. *Adapter | Socket.IO* [online]. 2024. [visited on 2024-04-12]. Available from: `https://socket.io/docs/v4/adapter/`.
17. *Redis adapter | Socket.IO* [online]. 2024. [visited on 2024-04-12]. Available from: `https://socket.io/docs/v4/redis-adapter/`.
18. *MariaDB Foundation – MariaDB.org* [online]. 2015. [visited on 2024-04-12]. Available from: `https://mariadb.org/`.
19. *What is a CI/CD pipeline?* [online]. 2022. [visited on 2024-04-15]. Available from: `https://www.redhat.com/en/topics/devops/what-cicd-pipeline`.
20. RAHUL AWATI, Kathleen Casey. *What is Docker Swarm? | Definition from TechTarget* [online]. 2024. [visited on 2024-04-15]. Available from: `https://www.techtarget.com/searchitoperations/definition/Docker-Swarm#:~:text=Docker%20Swarm%20is%20a%20container,virtual%20machines%20into%20a%20cluster.`.
21. *What is Load Balancing? – Load Balancing Algorithm Explained – AWS* [online]. 2024. [visited on 2024-04-15]. Available from: `https://aws.amazon.com/what-is/load-balancing/#:~:text=A%20load%20balancer%20is%20a,resource%20servers%20are%20used%20equally.`.
22. SHRISTI. *SSL/TLS termination. Ever heard this!!!! | by Shristi | Medium* [online]. 2023. [visited on 2024-04-15]. Available from: `https://medium.com/@reach2shristi.81/ssl-tls-termination-b7cc7de3eb54#:~:text=In%20this%20process%2C%20the%20SSL,processing%20burden%20on%20backend%20servers.`.
23. *The Socket instance (server-side) | Socket.IO* [online]. 2024. [visited on 2024-04-28]. Available from: `https://socket.io/docs/v4/server-socket-instance/`.
24. *joi – npm* [online]. 2024. [visited on 2024-04-25]. Available from: `https://www.npmjs.com/package/joi`.
25. *joi.dev – 17.12.3 API Reference* [online]. 2012. [visited on 2024-04-15]. Available from: `https://joi.dev/api/?v=17.12.3`.
26. KOLÁRIK, Ing. Martin. *IPv6 support · Issue #447 · jsdelivr/globalping* [online]. 2023. [visited on 2024-04-25]. Available from: `https://github.com/jsdelivr/globalping/issues/447`.
27. KOLÁRIK, Ing. Martin. *Selecting IP version* [personal communication]. 2024-03-23.
28. *OS | Node.js v22.0.0 Documentation* [online]. 2024. [visited on 2024-04-26]. Available from: `https://nodejs.org/api/os.html#osnetworkinterfaces`.
29. *Index of /lists* [online]. 2024. [visited on 2024-04-28]. Available from: `https://lists.blocklist.de/lists/all.txt`.
30. *Feeds for network protection – via firewalls & BGP routers* [online]. 2024. [visited on 2024-04-28]. Available from: `https://www.spamhaus.org/blocklists/network-protection/`.

31. *Team Cymru: Threat Intelligence and Risk Visibility Tools* [online]. 2024. [visited on 2024-04-28]. Available from: `https://www.team-cymru.org/Services/Bogons/fullbogons-ipv6.txt`.

32. *What is the Cymru bogons blocklist?* [online]. 2024. [visited on 2024-04-28]. Available from: `https://knowledge.validity.com/s/articles/What-is-the-Cymru-bogons-blocklist?language=en_US#:~:text=The%20Cymru%20bogons%20blocklist%20is,Regional%20Internet%20Registry%20(RIR)..`

33. *What is the BlockList.de blocklist?* [online]. 2024. [visited on 2024-04-28]. Available from: `https://knowledge.validity.com/s/articles/What-is-the-BlockListde-blocklist?language=en_US`.

34. *IANA IPv6 Special-Purpose Address Registry* [online]. 2024. [visited on 2024-04-29]. Available from: `https://www.iana.org/assignments/iana-ipv6-special-registry/iana-ipv6-special-registry.xhtml`.

35. HINDEN, R.; DEERING, S. *RFC 4291: IP Version 6 Addressing Architecture*. USA: RFC Editor, 2006.

36. BLANCHET, M. *RFC 5156: Special-Use IPv6 Addresses*. USA: RFC Editor, 2008.

37. WEGNER, J.D.; ROCKELL, Robert (eds.). Chapter 8 – Multicast Addressing. In: *IP Addressing & Subnetting INC IPV6*. Rockland: Syngress, 2000, pp. 339–352. ISBN 978-1-928994-01-5. Available from DOI: `https://doi.org/10.1016/B978-192899401-5/50011-5`.

Contents of the attachment

```
 README.md ......................................... contents description
 thesis ......................... the directory of LaTeX source codes of the thesis
   BP_Valéria_Frčková.pdf ..................... the thesis text in PDF format
   BP_Valéria_Frčková.tex .................. the main thesis text in tex format
   figures ........... the directory containing diagrams and assignment of the thesis
   text ................. the directory containing the thesis chapters in tex format
 globalping ........................... the directory of source codes for the API
 globalping-probe ........................ the directory of source codes for probe
```