



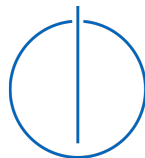
SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY —
INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

**Designing an Extensible Global Network
Measurement Platform**

Karthik Narumanchi





SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY —
INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

**Designing an Extensible Global Network
Measurement Platform**

**Entwerfen einer Erweiterbaren Globalen
Netzwerkmessplattform**

Author:	Karthik Narumanchi
Supervisor:	Prof. Dr.-Ing. Jörg Ott
Advisor:	Dr. Nitinder Mohan
Submission Date:	15.09.2023

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 15.09.2023

Karthik Narumanchi

Acknowledgments

I would like to thank my supervisor, Prof. Dr.-Ing. Jörg Ott, for giving me the opportunity to pursue this thesis with the chair of Connected Mobility.

I would like to express my sincere gratitude to my advisor, Dr. Nitinder Mohan, for his guidance, support, and for having given me the freedom to explore new ideas throughout this time.

I would also like to thank Hendrik Cech, Simon Zelenski, Ernst Bayer, and the RBG - Computer Operations team, for helping in setting up the computational infrastructure needed for this thesis work.

Finally, but most importantly, I would like to thank my parents, Nanditha and Aditya, for their endless love and encouragement.

Abstract

As Internet connectivity and reachability grow day by day, so do the challenges associated with them. Today, the infrastructure that powers the Internet is extremely prone to failures and disruptions. Identifying these issues is a complex effort due to the scale and heterogeneous nature of the Internet. The aim of this thesis is to develop a measurement platform allowing users to analyse the current state of the Internet and its services. To achieve this objective, a modular and distributed platform was built from the ground up. The platform was designed to allow users to create and execute measurements on probes that could be deployed globally. This was achieved by means of allowing users to self host a software container on heterogeneous hardware. This container was packaged with the required measurement services and utilities. Measurement support was added for both simple utilities like Pings and Traceroutes, as well as complex user defined code and scripts. To keep up with newly emerging satellite Internet constellations such as Starlink, the capability to extract and download satellite dish metadata was also added. Access for user, measurement, and probe management as well as the ability to download measurement results was made possible through a REST API. The platform and its services were deployed and Ping, Traceroute, and iRTT measurements were successfully run on probes in multiple continents. For the probes that were connected to a Starlink connection, satellite dish metadata was also extracted successfully.

Contents

Abstract	iv
1. Introduction	1
1.1. Problem	1
1.2. Objectives	2
1.3. Thesis Structure	2
2. Background	4
2.1. Internet Measurements	4
2.2. Measurement Platforms	5
2.3. Related Work	5
2.3.1. RIPE Atlas	5
2.3.2. EdgeNet	7
2.3.3. SpeedChecker	8
2.3.4. SamKnows	9
3. Requirements Analysis	11
3.1. Proposed System	11
3.2. Requirements Specification	12
3.2.1. Functional Requirements	12
3.2.2. Nonfunctional Requirements	13
3.3. System Modelling	14
3.3.1. Scenarios	14
3.3.2. Dynamic Modelling	15
4. System Design and Implementation	19
4.1. Overview	19
4.2. Subsystem Decomposition	21
4.2.1. API Service	21
4.2.2. Connector Service	24
4.2.3. Probe Service	25
4.3. Measurement Configuration	28
4.4. API Rate Limits for Measurements	29

4.5. Service Deployment - Orchestration	30
5. Evaluation	34
5.1. Setup and Infrastructure	34
5.1.1. Internal Services Deployment	34
5.1.2. Database Deployment	34
5.1.3. Probes Deployment	35
5.2. Evaluating Functionality	36
5.2.1. Predefined Measurement - Ping	37
5.2.2. Predefined Measurement - Traceroute	41
5.2.3. Arbitrary Measurement - iRTT	43
5.3. Evaluating System Performance	46
5.3.1. Measurement Execution Time	46
5.3.2. API Throughput and Response Time	46
6. Conclusion	49
6.1. Findings	49
6.2. Limitations	51
6.3. Future Work	53
6.3.1. User Interface	53
6.3.2. Continuous Integration and Deployment Pipeline	53
6.3.3. Enhancements	54
A. Reproducibility	56
A.1. Login	56
A.2. Fetching Probes and their Information	57
A.3. Measurement Execution	59
A.4. Result Downloads	60
Abbreviations	61
List of Figures	62
List of Tables	64
Bibliography	65

1. Introduction

1.1. Problem

Simultaneous support for both basic tools and custom measurements in existing network measurement platforms is very limited. The few that support them have other limitations such as a lack of measurement scheduling, limited ease of use, or a lack of public measurement data being made available to users. Additionally, some of them only allow users to run measurements on their own hardware. There is very rarely support for running custom measurements on crowd sourced and globally distributed hardware probes.

Furthermore, with the rise of satellite Internet constellations like Starlink [1], there is a need for measurement platforms to support these connectivity technologies and the features they offer. Starlink connections in particular offer unique dish metadata that enable users to see dish statistics in real time. They do this by exposing metadata through a gRPC API [2]. None of the existing network measurement platforms today have any support for extracting this data. Any metadata like this in combination with measurements and their results can be quite valuable to networks researchers.

Due to these limitations in current measurement platforms, there is a need for giving users the ability to perform all of these operations under one single system. This system should be supported by a global network of users and probes distributed across multiple continents. Users should be able to contribute to this network by easily hosting their own hardware for the purpose of running measurements. Support should be added for measurements ranging from simple Pings [3] to custom measurements that measure Quality of Service and Experience, and more. The system should additionally support the extraction of Starlink dish metadata in parallel while measurements are running on probes. More importantly, measurement data and result data should be made publicly available to all users for future networks research and analysis. Finally, the platform should be made highly extensible in order to add additional functionality and features as and when new trends and techniques emerge.

1.2. Objectives

The main objective of this thesis is the creation and development of an Internet measurement platform that must allow users to create measurements that are not just based on tools like Ping, Traceroute [4], DNS Lookup [5], etc., but also give users the ability to execute custom measurements. Custom measurements can be based on existing tools like iRTT [6], or iPerf [7], or they may be more complex and deal with packet capturing and video streaming data analysis. The platform must also be extensible and must allow for other measurement types to be added in the future. This will allow the platform to support new measurement tools that will be developed over time.

All measurements are meant to be run on probes that users can self-host. This ensures users can contribute to the platform and its network of resources, and allows them to take advantage of various probes hosted across the world. Through the means of hosting a client service within a Docker [8] container, users must be able to quickly spin up a software probe without the hassle of acquiring and maintaining specialized hardware for this purpose. To give users flexibility, these containers must be able to run on heterogeneous hardware such as physical machines, virtual machines, and even low powered hardware such as Raspberry Pi devices [9].

In combination with the support for custom measurements, if a user were to connect measurement probes to a satellite Internet connection such as Starlink for instance, the probes would need to gather dish metadata, if available. This would make the system better than the current state of the art as no measurement platform currently supports such satellite dish metadata extraction.

Finally, the platform must also allow its users to fetch and download results for all measurements, both new ones and those created previously. This will ensure that other users can reproduce existing results and measurements analysis.

1.3. Thesis Structure

This thesis introduces the foundational concepts of Internet measurements and measurement platforms in Chapter 2. The chapter also briefly describes and analyzes existing measurement platforms that were explored as part of the background work. Chapter 3 introduces the proposed software system and its components. The system requirements, both functional and nonfunctional are also listed here and system models are analysed based on these. In Chapter 4, the architecture and concepts involved

in the design and development of the new system are described. Subsystem components and their responsibilities are also discussed in detail along with other important ideas that have been implemented such as API rate limits and deployment strategies. Chapter 5 focuses on evaluating the newly developed system. Evaluation is based on both functional and nonfunctional requirements and is carried out by creating a test deployment environment and executing measurements on it. System performance is also analyzed in this chapter. Finally, in Chapter 6, the findings and limitations of the system are discussed and the chapter concludes this thesis by proposing new features and enhancements to the system in the future.

2. Background

This chapter introduces concepts and techniques related to measuring the Internet. First, Internet measurements are described in brief. After this, measurement platforms and what they entail are discussed. Finally, existing platforms and solutions are analyzed, including their features and drawbacks.

2.1. Internet Measurements

The Internet is a global system of networks that are connected to each other. These networks consist of inter-connected devices(nodes) made up of heterogeneous hardware and infrastructure that communicate with each other over a network and share information, resources, and services. This communication always takes place over an access medium, namely, hardwired, wireless, cellular or satellite and is made possible by large Internet service providers that establish connectivity between the various networks.

In any network, when a device sends any form of data to another device, the data often goes through several intermediary devices. How fast the data gets sent and through how many devices always depends on various factors such as the topology of the network, the number of devices in it, the underlying networking hardware or technologies, etc. To try and get an accurate representation of these variable factors, Internet measurements are used. In their simplest form, these measurements can simply be Pings and Traceroutes sent from one device to another. In the former case, packets of data, in bytes, are sent periodically to measure the time taken to both send and receive a response. In the latter case, packets of data are sent in the form of multiple hops from one device to another. Not only is the time taken for each hop measured but the overall route or path taken to the target destination is traced as well. Both these utilities are only a few examples used to measure both the connectivity and reachability in any network. There are several other utilities that help with measuring other factors too.

When utilities like these are applied globally, targeting devices that span countries and continents, the inner workings of the Internet can be measured quantitatively.

2.2. Measurement Platforms

Due to lack of access to devices in random regions across the earth, performing global Internet measurements can be seen as a fairly daunting task. This is where Internet measurement platforms attempt to solve the problem at hand. These platforms often leverage a global network of volunteer users and devices. Users sign up for hosting probes (devices or nodes) that are connected to the Internet and are used as sources to run measurements. These devices or probes located across the globe act as vantage points and therefore enable measurements on an extremely large scale.

The measurement utilities that these platforms and probes support often vary in nature. They might be basic utilities such as Pings, Traceroutes, DNS Lookups, and HTTP queries, or they may be arbitrary in nature, allowing users to define their own custom logic or scripts.

From a user perspective, when a user wants to run a measurement, they often access the platform through some sort of an interface, either a graphical interface, or a REST [10] API. Through the interface, the user can create measurements, retrieve results for them and manage their probes or devices. From the perspective of the platform and its business logic, workflows such as these would entail supporting not just measurement management, but user and probe management as well. In addition to this, some platforms make measurement request data and result data public, allowing for this data to be used in research and analysis or for result reproduction in the future.

2.3. Related Work

The following are a list of existing Internet measurement platforms that were reviewed as part of this thesis.

2.3.1. RIPE Atlas

RIPE Atlas is, at present, the largest Internet measurement platform [11]. It consists of a network of over 12000 probes spread across the earth [12]. These probes are a combination of both software and hardware probes. The software probes are Docker based and the hardware probes are small USB powered devices that are connected to an Ethernet connection. The hardware probes have been in existence for more than a decade and have seen multiple revisions improving performance and capabilities with each revision. Users (hosts) volunteer to host these probes and connect them to their own networks. While the probes are connected, measurements can be performed from

2. Background

them targeting any publicly accessible node on the Internet.

Measurement creation costs credits and costs vary depending on the measurement type and duration. Hosts are rewarded with credits based on the duration of hosting a probe. They can subsequently use these credits to execute measurements. They can also transfer them to other users. The measurements that are supported are Ping, Traceroute, DNS, SSL, HTTP, and NTP [13]. Each measurement type further supports customizable parameters such as Packets and Size for instance, for a Ping type. Measurements can be run on multiple probes and can also be scheduled to run at a specific start time.

Primary access to the platform is through a graphical user interface accessed via the website. Secondary access is through a REST API that the platform exposes. Both access interfaces allow most or all operations to be performed including but not limited to measurement management, keys management, and probes management.

All data that RIPE Atlas collects is freely and publicly available [14]. Data includes, but is not limited to, measurement request data, measurement result data, probe information and location data. The platform also exposes a Streaming API through the means of both an HTTP GET API and also a WebSocket [15] connection for real time data flow of all public results data.

Although the platform actively supports Starlink based satellite Internet connectivity for its probes, capabilities are on par with other hardwired or cellular probes. There is no metadata extraction from either the satellite dish or the router. There is also a lack of support for arbitrary execution of custom code or measurements on probes connected to the platform.

Figure 2.1 shows the internal architecture of RIPE Atlas as of 2015 [16]. All communication across the system components are handled by message queue servers that guarantee message delivery. These servers are connected to a Brain component that handles all business logic and measurement scheduling on probes. When a probe first starts up, the registration server receives a connection request from it. The server then routes the probe to its nearest Controller which will then forward all measurement request information to it.

All data is stored in two ways. First, an SQL database is used to store all the system data that pertains to probes, measurements, users, etc. Second, all the result data is stored in a big result store. This store is an HBase [17] cluster and uses MapReduce [18] operations for batch processing and aggregation.

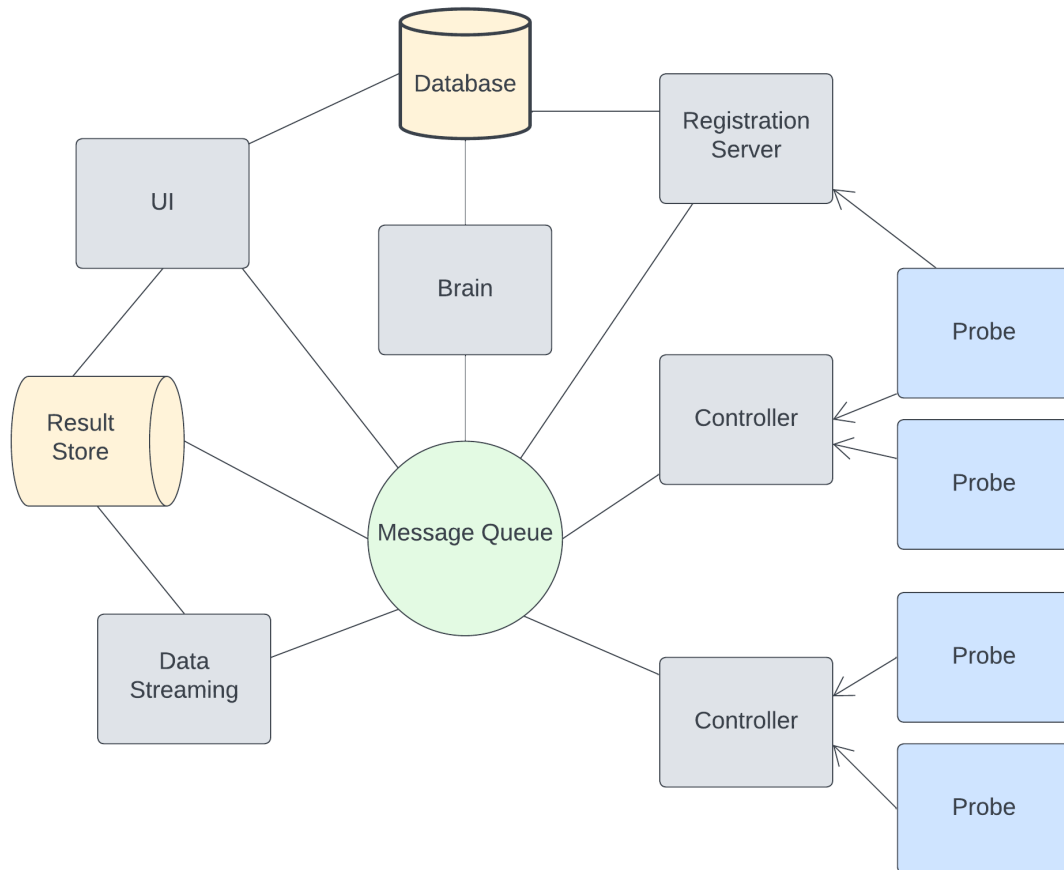


Figure 2.1.: RIPE Atlas architecture and system components

All the components of the system can be scaled up and down independently.

2.3.2. EdgeNet

EdgeNet is a globally distributed Internet research platform that allows users to run Internet experiments. Experiments can be run on nodes that are contributed by users. They are arbitrary in nature, ranging from a simple shell script to a full fledged web server. A node can either be a physical machine or a virtual machine.

2. Background

Users wanting to run experiments are vetted by means of a verification workflow. A user first registers with an institutional email address, followed by an EdgeNet administrator verifying the user's information, credentials and past research work. Once a user has been successfully verified, they can proceed to run experiments.

Nodes, either virtual or physical, use Kubernetes [19] and Docker for running experiments. The former is used in the form of *kubelet* [20] which is the agent that runs on the nodes. It handles container orchestration and management workloads. The latter is the container engine used to run user experiments. Nodes are required to have public IP addresses and can support both ARM64 and x86-64 CPU architectures. In order to host a node, a user downloads a shell script that proceeds to setup the required dependencies.

Experiments are created in the form of a Docker container. A user creates an experiment using tools or utilities and packages it all into a Docker image. Once the image has been built, it is then pushed to a container registry. To run the experiment on nodes, a user then creates a deployment using a Kubernetes configuration file. While creating a deployment, the user can specify deployment selector values to choose node locations, number of node targets, etc. Nodes can be requested with minimum resource limits as well by specifying required cpu and memory specifications. All deployment related tasks are carried out using Kubernetes' *kubectl* command-line tool [21]. After deployment has been completed, a user can monitor the experiment and retrieve logs and results using the tool.

EdgeNet provides a detailed acceptable use policy that lists guidelines and rules for general use of the platform and its services. Rules are listed for both node usage and network usage and violations of rules have consequences such as disabling accounts.

A user needs to be quite familiar with both Docker and Kubernetes to use the platform. There is, at present, no other user interface or API access to create experiments. As a result, experiment request data and experiment results are not public or stored for future retrieval. Users can only access results for the experiments they create.

2.3.3. SpeedChecker

SpeedChecker is an Internet measurement platform that sources, collects, and provides user Quality of Service and Quality of Experience measurement data [22].

Data is collected through tests that are user initiated or scheduled on mobile devices

and web browsers [23]. The former is enabled by both their own mobile applications and also by providing Software Development Kits that integrate with other mobile applications to run tests focused on Speed, Latency, Jitter, Video, Voice Over IP, and Packet Loss. The latter is provided in the form of an HTML API that can be integrated into websites. All measurements and tests target their CDN that has servers located in over 285 locations across the globe. Custom servers can also be targeted based on the SDK versions.

All crowd-sourced data can be accessed by means of two datasets, namely Speed Test Datasets and Cellular Coverage Datasets. The former [24] is formatted to provide information on Upload Speeds, Download Speeds, Ping, ISP, Latitude, Longitude, etc. The latter [25] focuses on cellular data such as Mobile Country Code, Mobile Network Code, Signal Strength, Frequency Channel, Signal Quality, etc.

Data is provided to Regulators, Mobile Network Operators, Fixed Network Providers(ISPs), as well as researchers.

2.3.4. SamKnows

SamKnows is a network measurement company offering solutions catered to measuring broadband and cellular performance for both home users as well as institutions such as large Internet Service Providers and Internet Regulators [26]. Started in 2008, they first offered measurements run on hardware devices(Agents) called as Whiteboxes [27]. These devices are connected directly to home routers and run tests to measure Internet performance. Over time, the company has collaborated directly with ISPs and regulators to establish performance standards and benchmarks.

SamKnows' test suite is very large and supports various tests based on user Quality of Service and Quality of Experience [28]. These include, but are not limited to, Speed Tests, Latency and Packet Loss Tests, DNS Lookups, Voice Over IP Quality Tests, Traceroutes, Video Streaming Quality Tests, etc.

Out of all these tests, the latency, traceroute and DNS tests are used very commonly. The latency test SamKnows uses is based on UDP and measures round trip time of UDP packets sent periodically from an agent to a target test server. The test is configured to run continuously over time and is set to send packets once every 1.5 seconds. Metrics recorded are based on minimum, average and maximum round trip times. The traceroute test they use is based on an open source client *mtr* [29, 30] and sends three probes to each hop to measure round trip times and packet loss from the hop responses.

2. Background

Finally, the DNS test that SamKnows uses can be configured with optional settings for additional endpoints, timeouts, and query values. It typically queries common hostnames such as *google.com* and *facebook.com*.

Today, SamKnows offers many more Agents, either hardware or software for running measurements [31]. New hardware agents supported are referred to as Router Agents. These are offered as a firmware upgrade to existing Internet routers and are intended for ISPs to embed into these routers to measure network performance. New software agents offered are based on an iFrame web app, a Mobile SDK that integrates their cellular and wireless measurement services, a Docker based measurement agent, and their own smartphone application to measure the phone's connection.

The Docker agent is available as an image and is distributed via DockerHub. It makes use of an *alpine linux* base image and is approximately about 17MB in size [31]. It supports hardware belonging to both the ARMv7 and x86 architectures which include devices such as Raspberry Pis and general purpose computers. Due to the nature of the Docker container running on top of the host machine, the container requires direct access to host machine networking. When a user wants to host the agent, they are provided with a one-line command to spin up the container. The agent supports all tests and functionality of their hardware router agent including the QoS and QoE tests.

All tests that the agents perform target high capacity test servers that the company maintains across the earth [32]. For tests that ISP Router Agents perform, ISPs can deploy their own additional servers within their networks.

3. Requirements Analysis

This chapter proposes the new system, its architecture, and the requirements it must fulfil. Requirements are identified using the FURPS+ model [33].

3.1. Proposed System

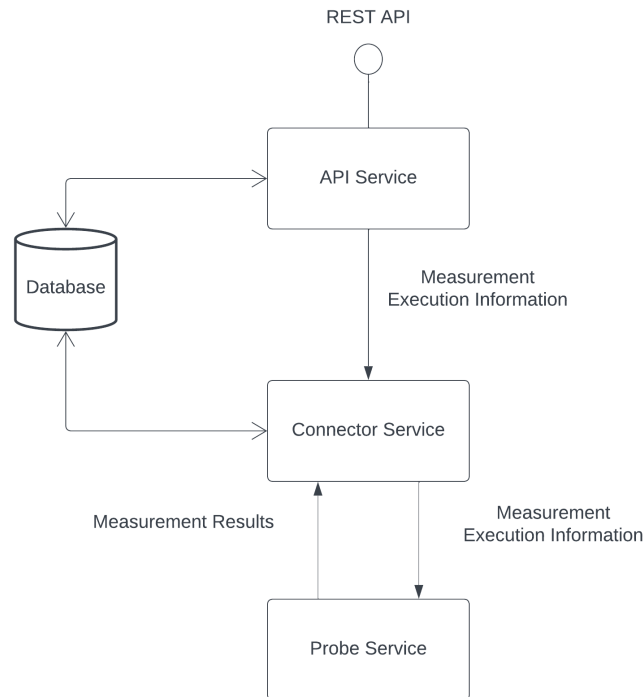


Figure 3.1.: Proposed system with major components and their interactions

Figure 3.1 shows an overview of the proposed system. The entire system is to be decomposed into three major service components.

The first service component, the API service, is the only user facing service. It exposes a REST API interface for end users and takes incoming requests for all operations including user management, measurement management, probe management, and downloading measurement results. It forwards measurement execution information to the intermediary Connector service component. Additionally, the API service is directly connected to a data persistence component to store and retrieve information pertaining to users, measurements, and probes, among other objects.

The second service component, the Connector service, is an internal intermediary service. It receives measurement execution information from the API service and forwards a subset of relevant data to the probes connected to it, to facilitate measurement execution on them. This service monitors all Probe service instances connected to it for a healthy connection. It also receives measurement results data from these probes as and when they complete measurement execution. Like the API service, it is also connected to the database for data persistence and retrieval.

The third and final component is the Probe service. This service is a client service intended to run within a container using Docker and executes measurements within this environment. It must be capable of running on heterogeneous hardware including Raspberry Pi devices. This service maintains a persistent connection to a Connector service instance and receives measurement execution information from it. After the measurements have been executed, it uploads the results back.

The Probe service has the added responsibility of extracting dish metadata from certain satellite connections such as Starlink. Starlink routers and dishes expose dish metadata which includes information on dish obstruction, dish angle, etc. This information is provided by means of a gRPC API. The Probe service connects internally to this API and fetches this information when a measurement is executing. It then sends this data to the Connector service instance for persistence.

3.2. Requirements Specification

3.2.1. Functional Requirements

This subsection lists the functional requirements for the proposed system.

FR1 Support for standardized Predefined measurements: It should be possible for users to create Predefined measurements based on Ping, Traceroute, DNS Lookup, and

HTTP GET/POST.

FR2 Support for user created Arbitrary measurements: It should be possible for users to create custom measurements based on arbitrary code so that they can use tools like iRTT, iPerf, and packet capturing software.

FR3 Support for retrieving satellite dish metadata: It should be possible for the system to retrieve satellite dish metadata(if available) for measurements run while connected to satellite Internet connections like Starlink.

FR4 Support for storage and retrieval of measurement results data: It should be possible for the system to store and fetch measurement results.

3.2.2. Nonfunctional Requirements

This subsection lists the nonfunctional requirements for the proposed system.

Usability

NFR1 Ease of use for probe hosting: The user must be able to retrieve probe configuration data for self-hosting

Reliability

NFR2 Restart policies: The system must automatically attempt to restart on disconnects or crashes.

NFR3 Result re-uploads: The client Probe service should automatically re-upload results on disconnects or crashes.

NFR4 Security - Encryption: User passwords must be encrypted prior to storing them in the database.

NFR5 Security - Role-based Access Control: Access to operations, and the operations themselves must be split based on two roles - User role, and Admin role.

NFR6 Security - Unauthenticated and unauthorized access: Unauthenticated and unauthorized users should not have access to application data including measurements

and their results.

Performance

NFR7 API rate limiting: There should be artificial limits on measurement creation to prevent the system from getting overloaded.

Supportability

NFR8 Extensibility: The system must be extensible. It should be possible to add support for new measurement types and features in the future.

Implementation

NFR9 Docker: The client Probe services must be container based and must use Docker.

NFR10 Raspberry Pi hardware: The client Probe services must run on low powered Raspberry Pi devices.

NFR11 Deployment: The internal services should be deployed using a container orchestration solution.

Interface

NFR12 Results - Formatting: Satellite dish metadata and results for Predefined measurements should be JSON [34] based. All result downloads must be ZIP archive based.

3.3. System Modelling

This section discusses aspects of the functional specification of the system.

3.3.1. Scenarios

This subsection lists scenarios focused on features of the system from the viewpoint of a user.

User management

A user with an Admin role can create users with either an Admin role or a User role. Each user is identified with a combination of a unique email address and a password.

Any user can login with their credentials to retrieve an authentication token. They can also update their credentials and delete their user resource.

Probe management

A user with an Admin role can create a probe resource. Every probe is associated with an existing user.

Any user can retrieve information on all probes. Any user can also retrieve hosting configuration data for their probes.

Measurement management

Any user can create and stop their measurements. Any user can view all measurements.

Result management

Any user can download measurement results for any measurement.

3.3.2. Dynamic Modelling

The models presented in this subsection focus on the behaviour of the system. Some of the most important interactions between a user and the system are depicted with the use of UML sequence diagrams.

Fetch probes

Figure 3.2 shows the sequence of messages exchanged between the user and the API service when the user wants to fetch all probes and their details. First the user authenticates and receives an access token. Using the access token, the User makes the request to fetch all probes. The API service verifies access and sends the probes response.

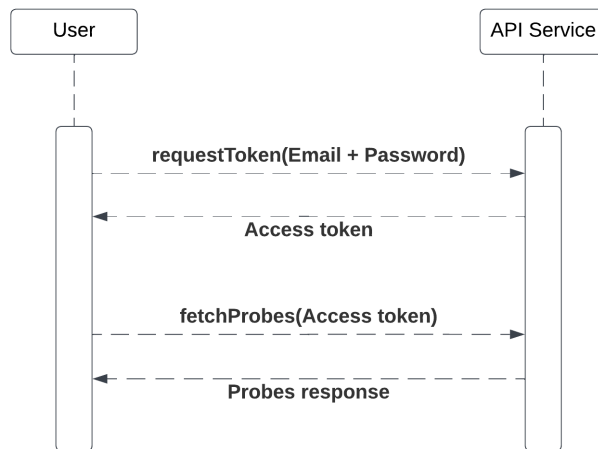


Figure 3.2.: UML sequence diagram depicting the sequence of messages exchanged when a user fetches all probes

Create measurements

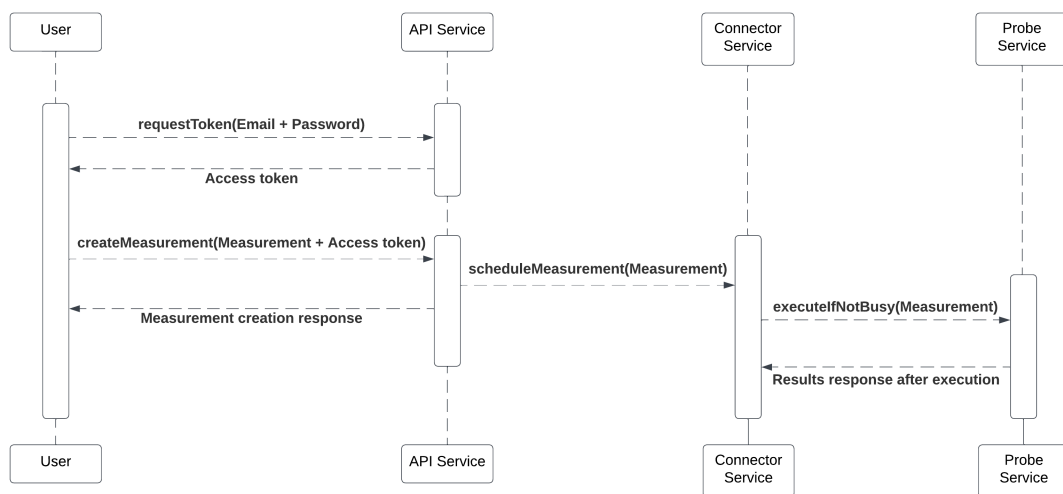


Figure 3.3.: UML sequence diagram depicting the sequence of messages exchanged when a user creates a measurement

Figure 3.3 depicts a sequence diagram depicting system behaviour when a user creates

a measurement.

First, the user authenticates and receives an access token. The user then uses this token to create a measurement request to the API service. The API service schedules the measurement by forwarding it to the Connector service. The API service then sends a measurement creation response back to the user, indicating that the measurement is created. The Connector service forwards measurement information to probes specified by the user. If the probe is not currently busy executing another measurement, it then executes the new measurement and sends the results back to the Connector service.

The measurement creation response sent from the API service to the user contains the measurement identifier.

View measurements and Fetch results

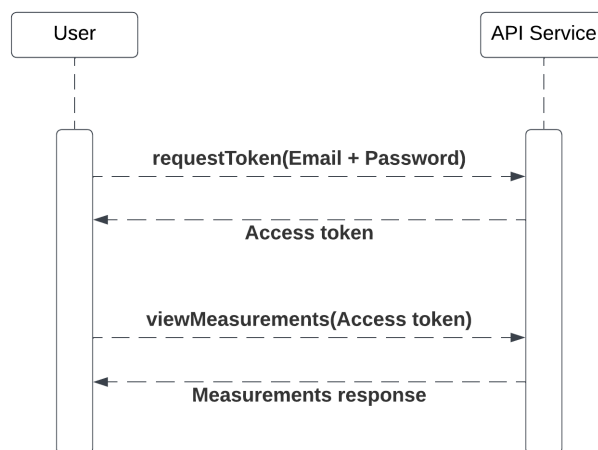


Figure 3.4.: UML sequence diagram depicting the sequence of messages exchanged when a user views all measurements

Figures 3.4, and 3.5 are additional sequence diagrams for *Viewing measurements* and *Fetching results* respectively. Like in the case of the previous interactions, user authentication is performed to retrieve an access token prior to performing the required operations.

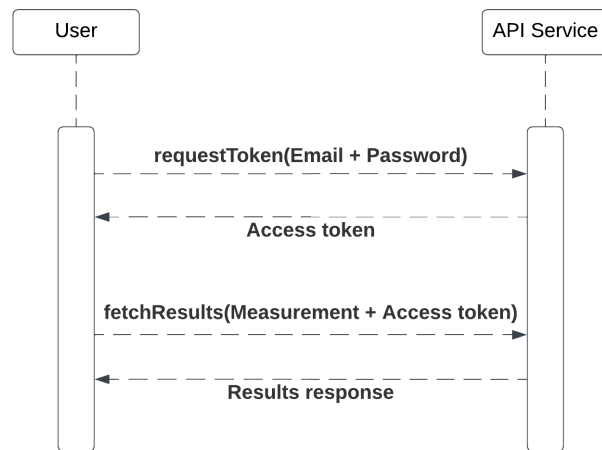


Figure 3.5.: UML sequence diagram depicting the sequence of messages exchanged when a user fetches results for a measurement

4. System Design and Implementation

4.1. Overview

Broadly, there are two types of measurements the new system will support - Arbitrary and Predefined measurements.

Arbitrary measurements: These are meant to give users the ability to execute custom code of their choice packaged within a Docker container. They will run for a user specified duration. Because these measurements are packaged within a container, the new system will use a Docker based environment to spin up the container internally using the *docker run* command.

Predefined measurements: These are meant to be measurements executed quickly in a minute or two against a single target. Five sub-types are chosen based on common measurements - *Ping*, *Traceroute*, *Paris Traceroute*, *DNS Lookup*, and *HTTP GET/POST*. To take advantage of the Docker based environment created for Arbitrary measurements, these predefined utilities will all be packaged into a small, lightweight container which will run within the same environment. This will unify both execution pipelines internally.

As a result of supporting both these measurement types, the system will have to deal with different types of result data. The Predefined measurements results will be formatted in JSON but the Arbitrary measurements results will be raw unstructured data. Taking both this factor and the measurement execution pipeline into consideration, a new system architecture was created. Figure 4.1 shows the top level component diagram for the newly developed system.

It is an extension of the proposed system design described in Chapter 3(Section 3.1) with the use of two databases for persisting all data - System DB for storage of users, measurements, probes data, and Results DB for storing results and satellite dish metadata exclusively. Using a separate database for storing results allows for easier horizontal scaling of that database alone. Over time, the Results DB will accumulate in size significantly more than the System DB due to the storage of unstructured data.

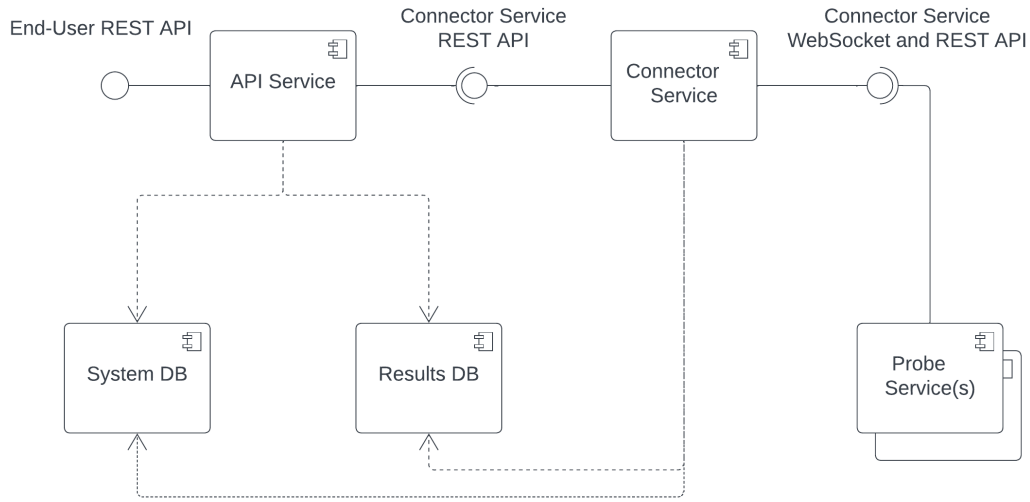


Figure 4.1.: Top level system component diagram

The three major service components remain the same with the diagram also indicating their exposed interfaces. The Connector service in particular exposes internal APIs and interacts directly with both the API service and the Probe service(s) due to its intermediary nature.

For all three services, Java based Spring Boot [35], an extension of Spring [36] was used as the base framework. Spring Boot is a production-grade enterprise web framework. It features comprehensive support for developing Java based web services and applications. It comes built-in with an embedded web server and has module support for web security, database management, logging and metrics, etc. Based on the need of each of the three services of the system, modules and dependencies were chosen appropriately.

For both the System Database and Results Database, MongoDB [37] was chosen due to these reasons:

1. The nonfunctional requirement NFR12 listed in Chapter 3(Section 3.2), and MongoDB's inherent support for JSON.
2. Use of JSON within the services framework ecosystem and Spring Boot's extensive support for MongoDB [38].

3. Supporting Arbitrary measurements meant that unstructured raw data storage would be needed. MongoDB's support [39] for BSON [40] and GridFS [41] was taken into consideration.
4. The extensible nature of the platform meant that new measurement types would keep getting added and existing measurement types would constantly keep receiving schema updates. A NoSQL document database such as MongoDB allows handling new data without disrupting the current structure [42].

4.2. Subsystem Decomposition

The three major subsystems are described in detail below.

4.2.1. API Service

The API service is the end-user facing service. It exposes a REST API for users and is their only entry-point to the entire platform. Figure 4.2 shows the complete internal architecture of the service.

The API service has multiple major responsibilities:

1. **User management:** The service exposes endpoints for the creation, updation and deletion of all users of the platform. In order to facilitate operations such as user creation, the service uses a Role-Based Access Control [43] mechanism wherein there are two broad roles of users - an Admin, and a User. The former has higher privileges and can create users. The latter can access all other user operations. For security, a PBKDF2 cryptographic key derivation function [44] is used for user passwords. In addition to this, all users have to request JSON Web Tokens [45] for accessing API operations. Both security measures are Internet Standards documented by their RFCs [46, 47]. The service persists all user data to the System Database.
2. **Probe management:** The service exposes endpoints for the creation and retrieval of probes and their details including probe configuration for self-hosting purposes. Again, RBAC is used allowing only Admin users to create a probe. All the other probe operations can be performed by any user. Every probe is linked to a user that is already created and includes information such as Country, Region, Status, IPv4, etc. All probe data is stored in the System Database.

When a user wants to host a probe, they make an API request to download probe configuration data for that probe. When the API service receives this request, it

4. System Design and Implementation

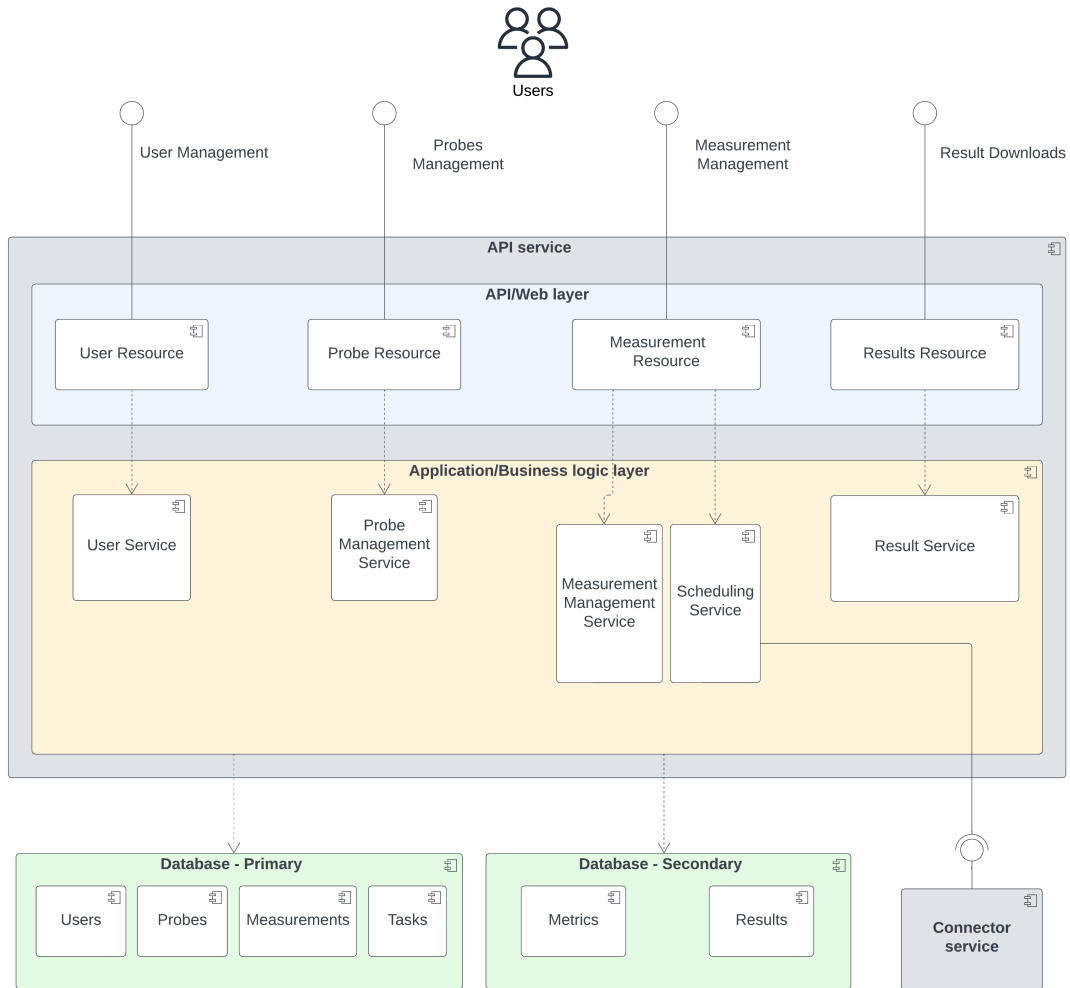


Figure 4.2.: Architecture of the API service

first ensures that the probe is linked to the requesting user. It then dynamically embeds data within the configuration like security data, connection strings, etc. The API then serves the user with a downloadable Docker Compose [48] based configuration file which the user can use to deploy the Probe service.

3. **Measurement management:** Endpoints are also exposed for creating, stopping, and viewing measurements. Any user can perform these operations. Users can choose to create measurements of both types, Predefined and Arbitrary. Every measurement is linked to the user that creates it and data includes various information such as Type, Specification, Repeat Specification, Probe Specification, Status etc. The Type and Specification in a measurement object contain data specific to the measurement type and its parameters. The Repeat Specification object deals with scheduling a specific measurement to repeat multiple times based on an interval duration. The Probe Specification object deals with the probes on which the measurement is intended to run. Again, like the previous two cases, all measurement data is stored in the System Database.

When a user creates a measurement to be executed on some probes, the API service receives the measurement request data. After verification of the data, it then forwards it to the relevant Connector service instances that are connected to their respective probes and returns the successful measurement creation response back to the user.

Measurement scheduling takes place as they come in. At the time of measurement creation, if probes specified are available and not busy, measurements are scheduled immediately and all data is forwarded to the Connector service. However, if at the time of creation, any probes are not available or are busy, measurements are not scheduled and are treated as conflicting. These scheduling conflicts are also checked for repeating measurements. Their scheduling status is indicated to users when they query for them.

4. **Result management:** Endpoint access also exists for downloading results for a measurement. Any user can download results for any measurement. Results are ZIP archive based and serve users with a downloadable ZIP file. The file contains differing files inside based on the type of the measurement. If the measurement was executed on any probe connected to a Starlink connection, the results can possibly also contain satellite dish metrics associated with that probe collected during the measurement execution time.

All results and dish metadata are fetched from the Results Database.

4.2.2. Connector Service

The Connector service is an intermediary internal service. It exposes a REST API to consume measurement information coming in from the API service. It also exposes a WebSocket connection and other REST API endpoints to receive data coming in from the probes. All of these interfaces are authenticated by means of API Keys.

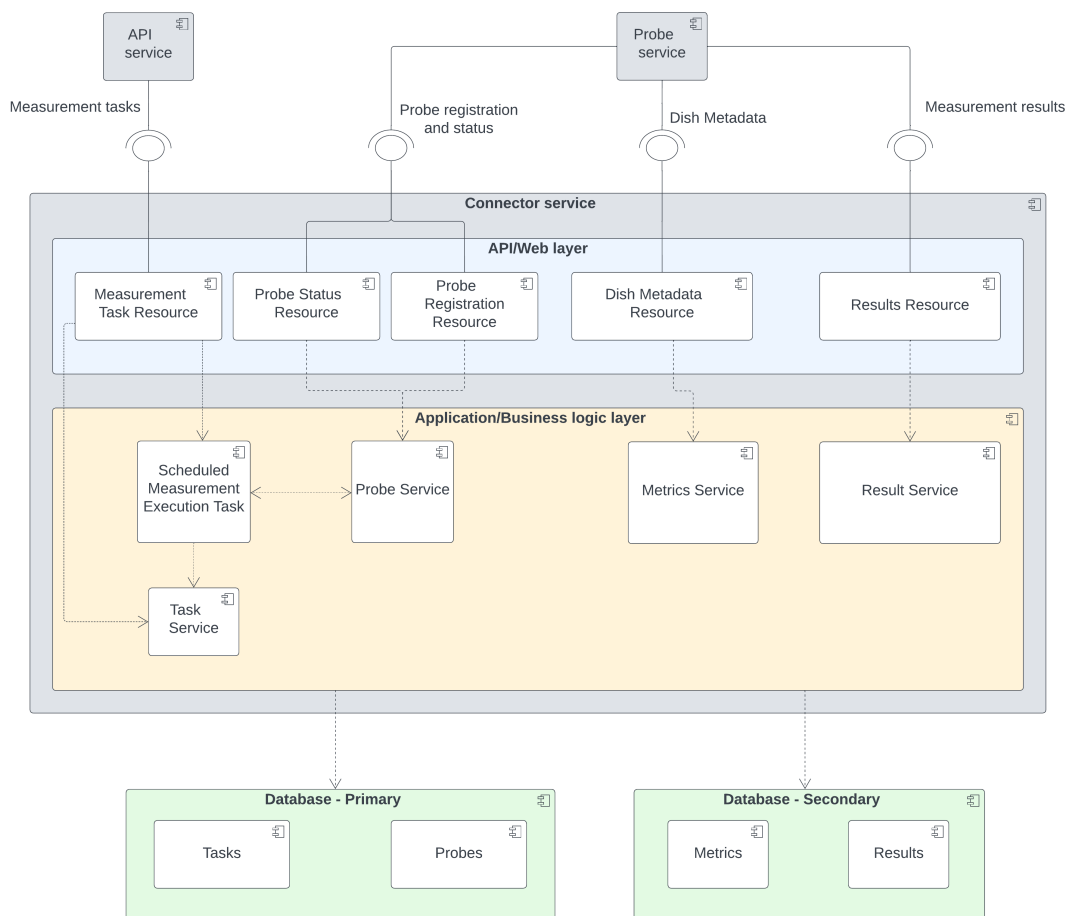


Figure 4.3.: Architecture of the Connector service

Figure 4.3 shows the complete internal architecture of the service.

From a system design perspective, the Connector service instances are meant to be deployed across different regions and are responsible for managing a set of probes within

those regions. In order to achieve this, each service exposes a WebSocket connection for these probes. The probes automatically initiate a connection to this socket when they start up. After successfully authenticating the connection request, the Connector service receives probe registration information that identifies each probe uniquely. The Connector service then persists relevant information to the System Database including probe status information thereby effectively *linking* itself to that probe. This *linking* remains active as long as the socket connection remains alive.

Now, when a user creates a measurement involving one of those probes, the API service, by fetching probe information from the System Database knows which Connector service instance to communicate with for that probe. It then proceeds to send measurement information to that instance using its API endpoint. Upon receiving this data, the Connector service instance proceeds to schedule tasks internally for that measurement. For scheduling, the Spring TaskScheduler interface is used [49]. When these scheduled tasks run, measurement information is forwarded to the relevant probe for execution using the active socket connection.

After the probe runs the measurement, it sends measurement results back to the Connector service instance along with any optional satellite dish metadata it collected. The results and metadata once received are then formatted into either JSON or BSON and stored in the Results Database.

The Connector service through the socket connection to a probe is always actively listening to WebSocket disconnect events. When a probe connection breaks, it immediately updates its status to indicate that the probe has been disconnected.

4.2.3. Probe Service

The Probe service is a client service intended to run measurements on hardware belonging to users. To satisfy the nonfunctional requirements NFR9 and NFR10 listed in Chapter 3(Section 3.2), the service is packaged to run within a Docker container and made lightweight enough to run on low powered Raspberry Pi devices.

The service itself does not expose any API endpoints. All communication is only one way towards the Connector service other than the measurement information it receives through a self-initiated WebSocket connection. This socket connection initiation takes place on startup every time the probe container gets spun up. Once this takes places, the service is ready to accept measurement requests.

4. System Design and Implementation

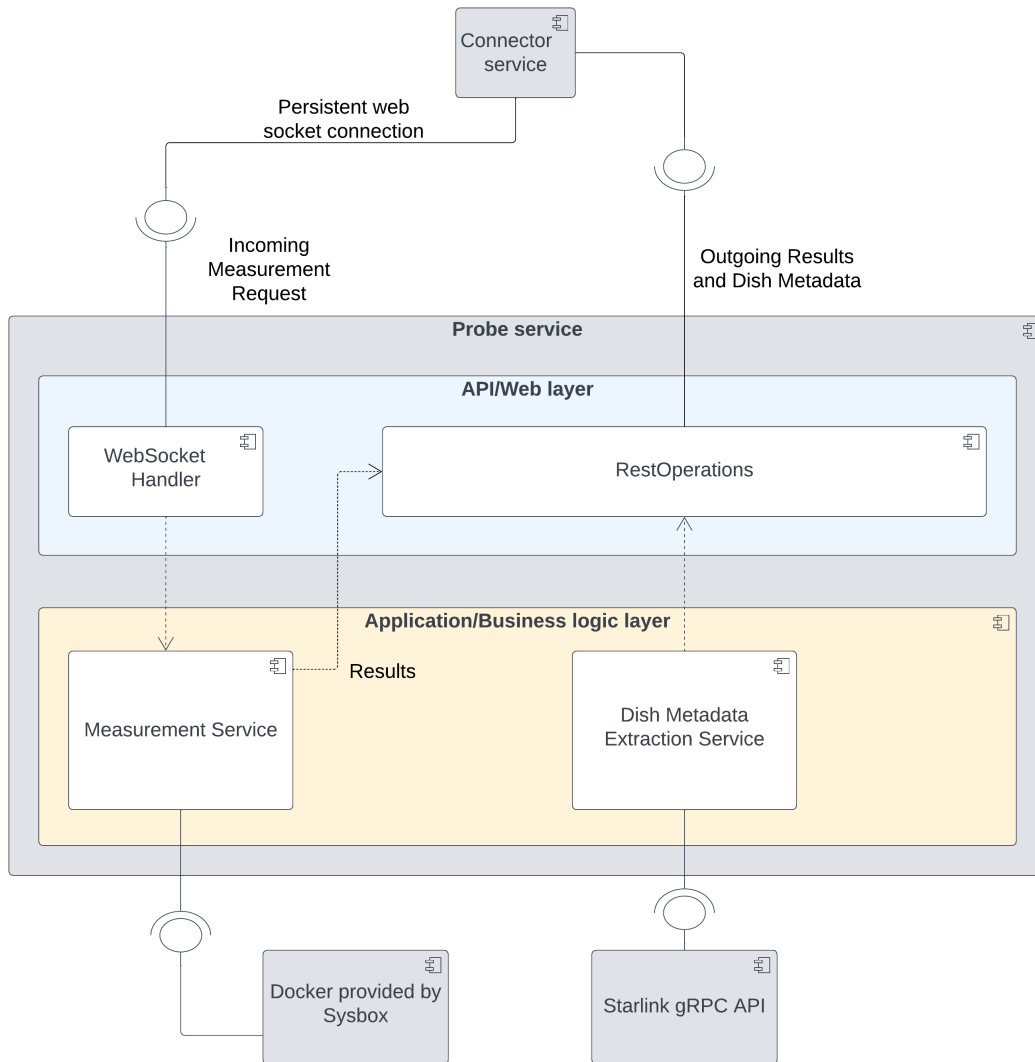


Figure 4.4.: Architecture of the Probe service

Figure 4.4 shows the architecture of the Probe service.

For the execution of the measurements, Docker containers are used. Based on the two major measurement types the system supports, Predefined and Arbitrary, different container images are pulled and are spun up to act as execution environments. For Predefined measurements, the container image is an extremely lightweight(5 MB) *alpine linux* image [50] with added utility support for *ping*, *traceroute*, *paris-traceroute*, *curl*, and *nslookup*, which are the current supported measurement subtypes. In the case of Arbitrary measurements, users package their own code into containers. These containers are then spun up similarly for measurement execution. After measurements have been executed, their results are then sent back to the Connector service for persistence.

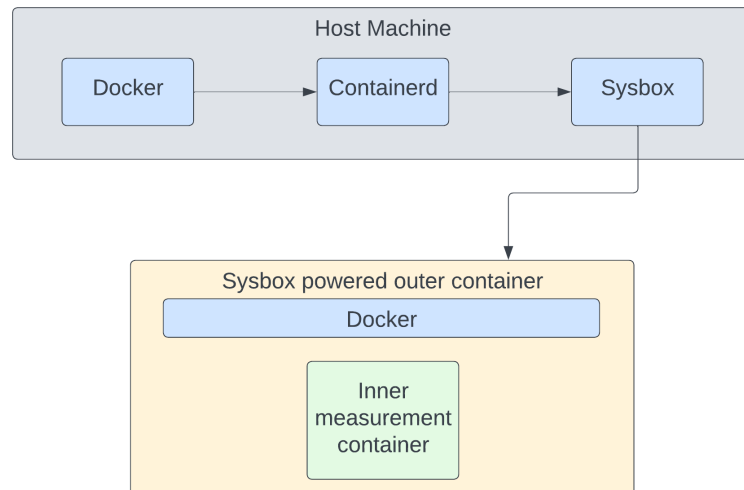


Figure 4.5.: Sysbox Docker in Docker approach

To enable such a Docker in Docker approach, the container runtime Sysbox [51] is used. Sysbox is a runtime that enables containers to run software such as Docker and Kubernetes. The software runs within containers without any modification required. Sysbox also greatly improves container isolation and security [52]. It allows a user to execute any code or software within Sysbox system containers safely preventing them from accessing the host machine. Therefore, by using Sysbox, the Probe service is first created as an outer system container. Then, once the Probe service container is up and running, measurement containers inside are spun up as inner containers.

Effectively, this allows both the outer system container and the inner containers to execute privileged measurement code without access to the host machine or device.

Finally, the Probe service also has support for collecting satellite dish metadata when measurements are running. Currently, there is support for extracting such data only from Starlink based satellite connections. Starlink routers and satellite dishes expose metadata using a gRPC API. By consuming this API from within the Probe service, dish metadata is collected for the duration of measurement execution. Once measurement execution is completed, this data is sent back to the Connector service for persistence.

4.3. Measurement Configuration

For both the Arbitrary and Predefined measurement types, the system supports configuration via parameters. These parameters are unique to each type of measurement and take in values from users.

Since Arbitrary measurements are custom and container based, they support more parameters. These are:

1. *Container Duration*

2. *Container Image*

This property refers to the identifier for the container image hosted on the DockerHub container registry [53].

3. *Container Entrypoint*

This property is used to define the container's default process when the container is run [54].

4. *Command Inputs*

This property is used to define the container's default process arguments when the container is run [54].

5. *Environment Variables*

This property array is used to pass user provided environment variables to the container when it is run [55].

6. *Output Path*

This property refers to the directory path within the container where results are stored. The Probe service internally maps a temporary directory to this path within the user container in order to retrieve measurement results [56].

7. *Linux CAP_NET_ADMIN capability* [57]

This property allows the user to indicate if the privileged linux CAP_NET_ADMIN capability should be granted to the user container. This capability is useful in specific cases such as network packet capturing or network interface configuration for instance.

For Predefined measurements and their five sub-types, these are their customizable parameters:

- **Ping**

Target, Number of packets, Packet byte size

- **Traceroute**

Target, Maximum hops, Method(UDP, ICMP, or TCP)

- **Paris Traceroute** [58]

Target, Maximum hops, Method(UDP, ICMP, or TCP)

- **DNS Lookup**

Target

- **HTTP GET/POST**

Target, Method(GET, or POST), Query String - in case of POST

4.4. API Rate Limits for Measurements

The nonfunctional requirement NFR7 listed in Chapter 3(Section 3.2) requires limits to be applied on measurement creation to prevent excess consumption of resources and overloading the system.

To address this, artificial limits have been applied to the measurement creation API as follows and apply at a user level:

- For Predefined measurements, there is currently a maximum limit of 25 simultaneous scheduled measurements.

- For Arbitrary measurements, there is currently a maximum limit of 3 simultaneous scheduled measurements, in combination with the following:
 1. Maximum runtime of one hour per measurement run
 2. Maximum 10 repetitions of the measurement
- To limit access to an excess number of probes, only a maximum of 100 probes can be specified per measurement of any type.

When a user attempts to create a measurement that exceeds any of these limits, the measurement is not scheduled to run and is deemed failed. These limits are an attempt at giving every user fair access to compute resources.

4.5. Service Deployment - Orchestration

The API service and the Connector service were designed from the beginning to be containerized and deployed using an orchestration solution. For this purpose, Oakestra [59] was chosen. It is a lightweight orchestration framework that was specifically designed for heterogeneous hardware and edge infrastructures.

By using Oakestra, these are some of the major benefits for the system's service infrastructure:

1. Service deployment performance:

Both the system services are packaged within containers that may be deployed on hardware ranging from low performance personal computers to high performance cloud machines. Consistent deployment times for these containers is a major concern due to the extensible nature of the platform. Oakestra solves this issue by making deploy times extremely minimal irrespective of the hardware underneath. Containerized service deployments in particular have shown to be faster compared to other orchestration solutions such as MicroK8s [60] and Kubernetes [61]. As a result, Connector service deployments in new regions can take advantage of this even if the regions have degraded network performance.

2. Service networking performance:

Due to the geographically distributed nature of the services, connectivity and issues such as latency and packet loss are a concern. Subpar connectivity between an API service instance and a remote Connector service instance will fail to guarantee time bound measurement execution in the real world. One of the benefits of Oakestra that fits into this scenario is its support for inter-cluster

networking. Oakestra's networking component, the Network Manager enables communication between services deployed across clusters. It achieves this with two features - subnetworking, and semantic addressing policies [62].

The former ensures that every worker node has its own subnetwork and allocates private namespace addresses to all services in a node. This allows every single service instance to have its own IP address.

The latter builds upon this and enforces traffic balancing policies based on either a round robin algorithm or a closest device algorithm. When multiple API service instances are scaled horizontally and are deployed across different regions, the round robin algorithm would help in uniform traffic balancing by routing user requests to every service instance one by one. When multiple Connector service instances are deployed similarly in different continents, the closest device algorithm would help probes connect to their closest Connector instance to reduce overall connection latency. Both strategies are extremely beneficial.

Taking advantage of these benefits, the system and its internal services can be deployed using multiple deployment strategies:

1. **One-machine minimal deployment:**

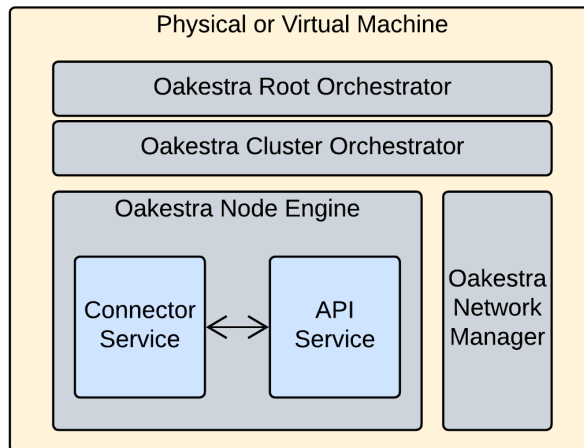


Figure 4.6.: One-machine service deployment using Oakestra

In this scenario, all of the core components of Oakestra are first setup on a single machine. The Root orchestrator, the Cluster orchestrator, the Node Engine and

Network Manager components are all deployed. Next, using the deployment descriptor and Oakestra’s REST API, both the internal services, the API service and Connector service are deployed through containerization on the Node Engine. One single instance of each service is started. Internally, Oakestra manages orchestration of these instances. Networking for the service communication is then setup to reflect the subnetwork addressing created by the Network Manager. By using the instance IPs, both services can communicate with each other directly. This scenario is suitable for a minimal deployment or a test environment where the Connector service instances need not be distributed across different regions. A working example of this setup was created for system evaluation, discussed in Chapter 5(Section 5.1).

2. Distributed hybrid deployment:

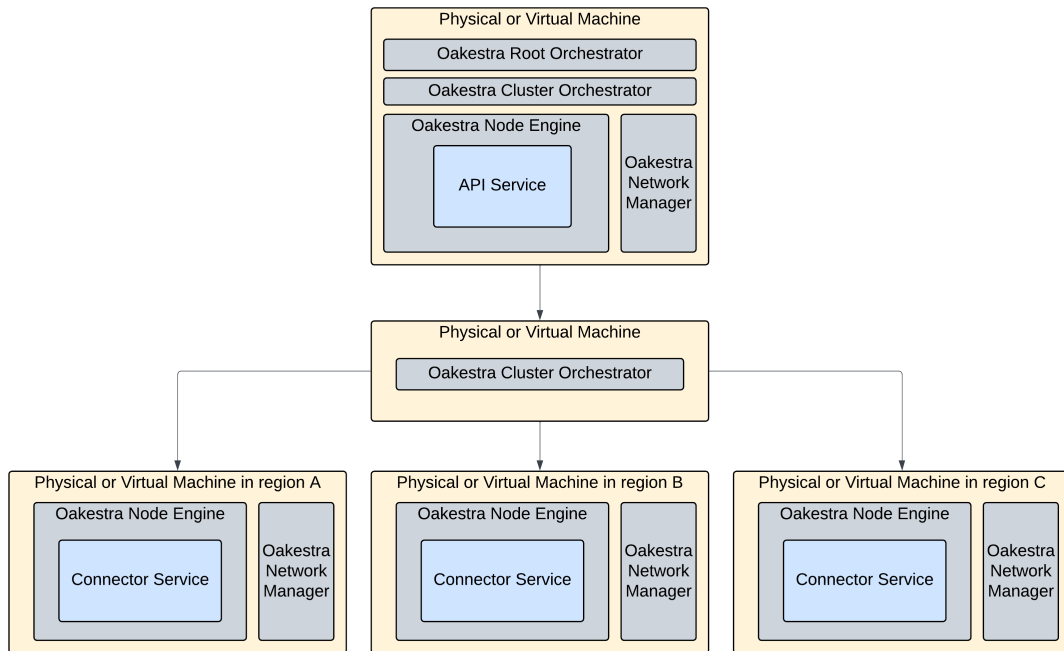


Figure 4.7.: Distributed service deployment using Oakestra

In this scenario, Connector service instances are scaled and distributed across different regions. This allows them to control a high number of probes across those regions. It is therefore more representative of a production environment. For this setup, multiple Cluster orchestrators or Node Engines or any combination

4. System Design and Implementation

of both can be used to scale the services as needed. These can be located in different locations and can be based on different types of computing hardware ranging from cloud based virtual machines to general purpose computers. This will enable every Connector service instance to control its own regional set of probes, thereby reducing overall load and increasing scalability. While all of these services are distributed, inter-service communication is again handled by the Network Manager component and its subnetwork addressing.

5. Evaluation

This chapter aims to evaluate the functionality and performance of the newly developed system. For this purpose, test infrastructure was setup and the system services were deployed on it. Measurements and tests were then executed.

5.1. Setup and Infrastructure

The infrastructure and system configurations used are described below.

5.1.1. Internal Services Deployment

Both the internal API and the Connector service instances were deployed using Oakestra’s one device one cluster strategy [63]. A single Oakestra cluster with one Root Orchestrator, one Cluster Orchestrator, one Node Engine component, and one Network Manager component were setup on a single virtual machine. Table 5.1 lists the machine specifications.

Table 5.1.: Machine specifications for deploying internal services

Name	Hardware	OS (Ubuntu Server)	(v) CPUs	RAM (GB)	Location
VM 1	Virtual Machine	20.04	4	4	Germany

The Oakestra deployment was made via container virtualization. One container instance for each service was deployed by using a deployment descriptor. Inter-service communication is handled by Oakestra’s Network Manager and its support for subnetwork IP addressing.

5.1.2. Database Deployment

Both the above services were connected to a MongoDB 6 Community Edition database server running on a virtual machine. Table 5.2 lists the machine specifications.

The database connection was secured using Mongo’s SCRAM authentication mechanism [64]. Additionally, incoming network access to this machine was IP restricted to

Table 5.2.: Machine specifications for deploying the database

Name	Hardware	OS (Ubuntu Server)	(v) CPUs	RAM (GB)	Disk Space (GB)	Location
VM 2	Virtual Machine	20.04	2	2	50	Germany

allow access only from machine VM1.

Figure 5.1 shows the deployment diagram for both machines VM1, and VM2.

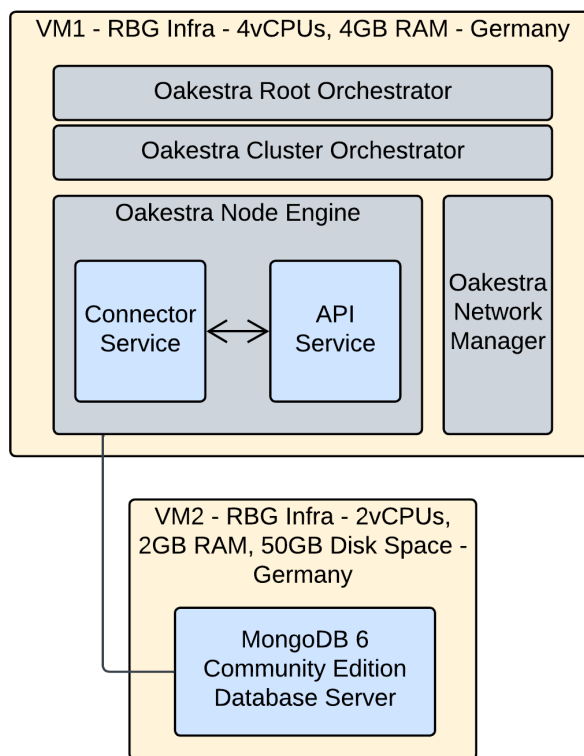


Figure 5.1.: Deployment diagram

5.1.3. Probes Deployment

Several Probe service instances were used and deployed on different hardware and network conditions across three different continents to simulate a real world environment

for evaluation.

Probe deployment process was also made to resemble a real world scenario simulating how an end-user would carry out the process. Once the probes were created by an Admin user, probe deployment configuration files(Docker based) were downloaded for each probe. Configuration properties were modified following instructions that would be presented to end-users. Finally, *docker compose* was used to start up the Probe service containers.

All probe containers successfully authenticated and connected automatically to the Connector service instance on startup. No intervention was required for any of the probes. Self registration was performed and this was verified by means of fetching probes data using the API service. Data showed all probes being connected successfully to the platform, ready to accept measurement requests.

Table 5.3.: Probes used and their deployment details

Name	Hardware	OS (Ubuntu Server)	(v) CPUs	RAM (GB)	Interface	Location
Probe A	RPi 4 Model B	22.04 LTS	4	4	Wireless	Germany
Probe B	RPi 3 Model B	22.04 LTS	4	1	Wireless	Germany
Probe C	Virtual Machine	20.04	2	2	Starlink	Germany
Probe D	AWS EC2	22.04 LTS	1	1	Wired	USA
Probe E	AWS EC2	22.04 LTS	1	1	Wired	India

Table 5.3 shows the probes used and their details.

5.2. Evaluating Functionality

Evaluation was performed by executing measurements on the deployed probes. First, two sets of Predefined measurements were run using Ping and Traceroute. Next, an Arbitrary measurement using iRTT was run to demonstrate functionality of both the major measurement types.

For both the Predefined measurements, the target selected was a Microsoft Azure App Center server located in central Europe.

5.2.1. Predefined Measurement - Ping

The Ping measurement parameters were selected as follows and the measurement was repeated 10 times totalling to 11 measurement executions spread across a total time of about 35 minutes across all 5 probes:

- Target IP: 4.232.99.0
- Number of Packets: 3
- Packet Byte Size: *Default(56 Bytes)*

Figure 5.2 shows the entire JSON payload for the measurement request.

```
{
  "type": "PING",
  "measurementSpecification": {
    "target": "4.232.99.0",
    "numberOfPackets": 3
  },
  "description": "Ping mmt-thesis",
  "repeatSpecification": {
    "numberOfRepeats": 10,
    "interval": 180
  },
  "probeSpecification": {
    "probeIds": [
      "64c9f3571d52ec12b0f2feb4",
      "64dd7d0320b80e736f5194ce",
      "64dd7d2320b80e736f5194cf",
      "64dd7c3720b80e736f5194cb",
      "64dd7c5520b80e736f5194cd"
    ]
  }
}
```

Figure 5.2.: JSON payload used for the Ping measurement request

Table 5.4 shows the overall results. The factors *Measurements Coverage*, *Results Coverage*, and *Dish Metadata Coverage* were chosen to evaluate the functioning of the system.

5. Evaluation

For *Dish Metadata Coverage* specifically, Probe C was connected to an active Starlink connection and metadata gathering was enabled for this probe. Ideally, for every second of measurement execution, one metadata record is fetched from the dish. For instance, if a measurement runs for 3 seconds, ideally this should result in 3 metadata records gathered.

Table 5.4.: Ping measurement evaluation results

	Probes				
Ping Results	Probe A	Probe B	Probe C	Probe D	Probe E
Measurements Scheduled	11	11	11	11	11
Measurements Executed	11	11	11	11	11
Measurement Coverage	100%	100%	100%	100%	100%
Measurement Results Returned	11	11	11	11	11
Measurement Results Coverage	100%	100%	100%	100%	100%
Total Execution Time	44s	104s	33s	33s	33s
Dish Metadata Records	-	-	31	-	-
Dish Metadata Coverage	-	-	93.93%	-	-

Results show complete measurement execution coverage. All scheduled measurements were executed on all of the probes successfully. Complete coverage for results downloads was also achieved. Probe C, which was the only probe connected to a Starlink connection returned satellite dish metadata covering about 94% of the total measurement execution time. Only two metadata records were missing.

Severe execution time discrepancies can be noticed across some probes. This is discussed in detail below in Subsection 5.3.1.

Round trip time results for the Ping measurements on all probes are visualized below in Figures 5.3 - 5.7. Results are normal for ping values seen in devices located in the probes' locations and the server being located in central Europe. Probes A, B, and C are located in Germany and have round trip time values between 25-50 milliseconds. Probe D is located in the USA and has RTT values between 95-97 milliseconds. Finally, Probe E is located in India and has RTT values between 129-130 milliseconds.

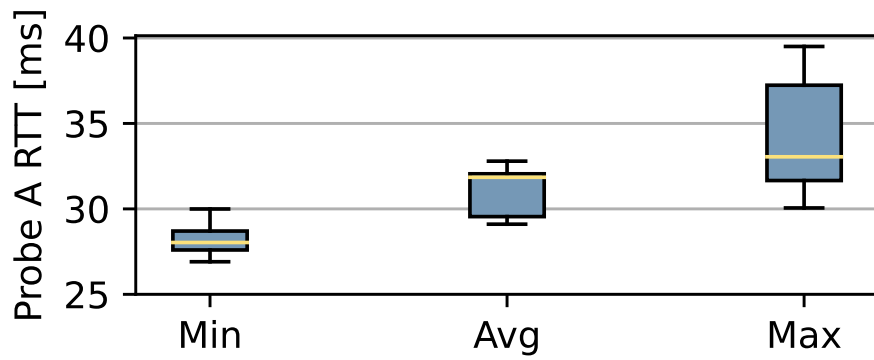


Figure 5.3.: Ping RTT values for Probe A

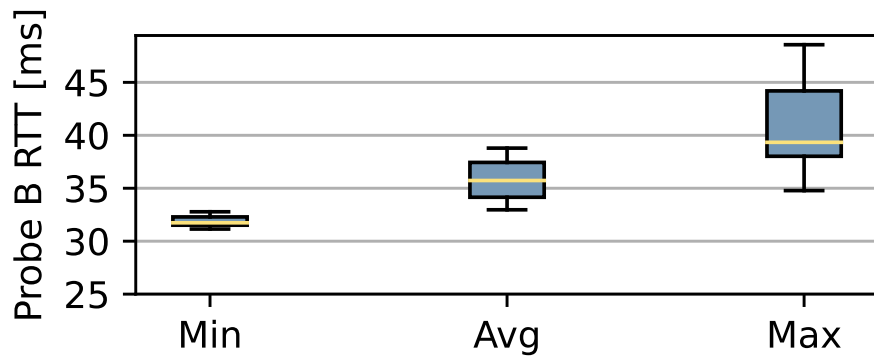


Figure 5.4.: Ping RTT values for Probe B

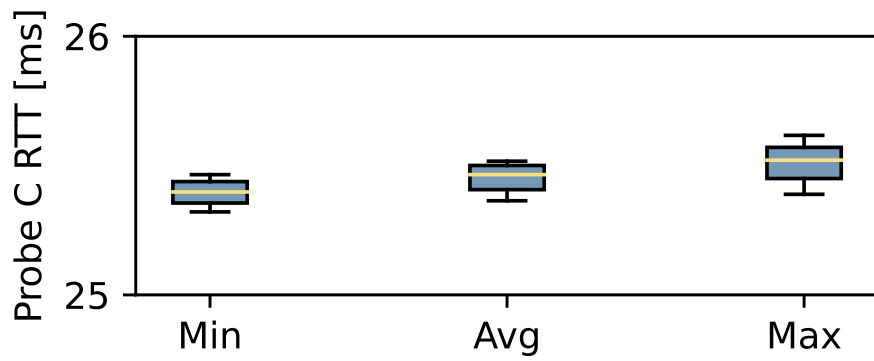


Figure 5.5.: Ping RTT values for Probe C

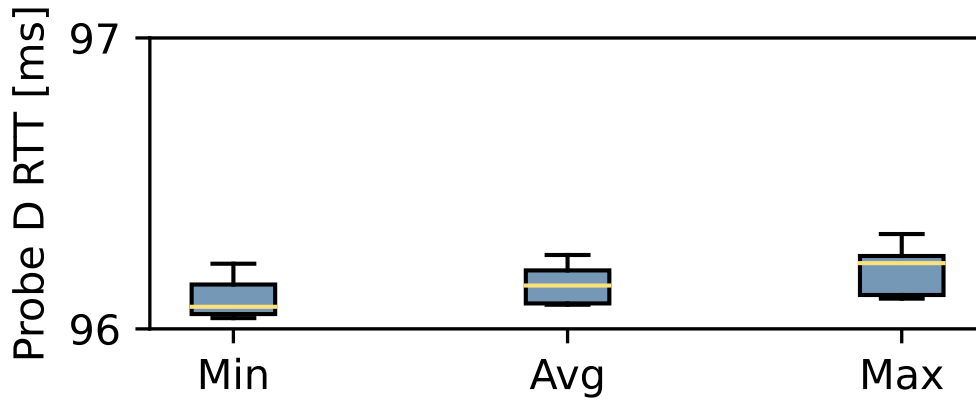


Figure 5.6.: Ping RTT values for Probe D

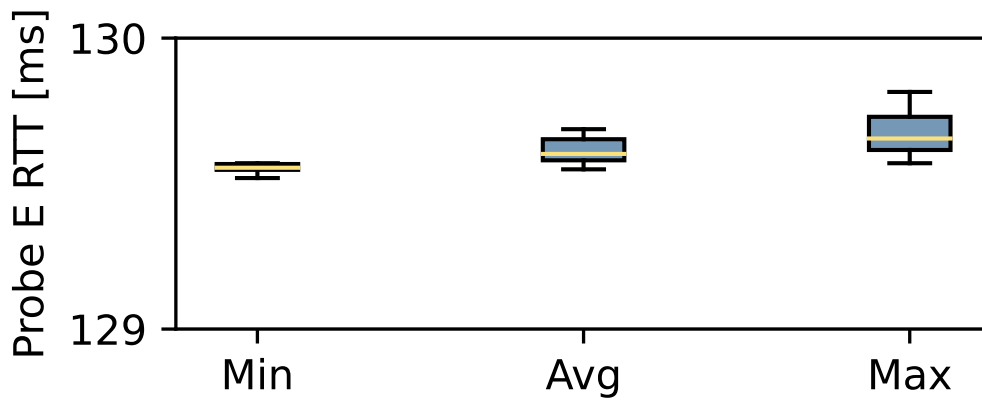


Figure 5.7.: Ping RTT values for Probe E

5.2.2. Predefined Measurement - Traceroute

For the Traceroute measurement, again, all five probes were chosen. The target selected was the same server located in central Europe.

The Traceroute measurement parameters were selected as follows and the measurement was repeated 10 times totalling to 11 measurement executions spread across a total time of about 35 minutes across all 5 probes:

- Target IP: 4.232.99.0
- Maximum Hops: 30
- Packets Per Hop: 3
- Method: ICMP

```
{
  "type": "TRACEROUTE",
  "measurementSpecification": {
    "target": "4.232.99.0",
    "maxHops": 30,
    "packetsPerHop": 3,
    "method": "ICMP"
  },
  "description": "Traceroute mmt-thesis",
  "repeatSpecification": {
    "numberOfRepeats": 10,
    "interval": 180
  },
  "probeSpecification": {
    "probeIds": [
      "64c9f3571d52ec12b0f2feb4",
      "64dd7d0320b80e736f5194ce",
      "64dd7d2320b80e736f5194cf",
      "64dd7c3720b80e736f5194cb",
      "64dd7c5520b80e736f5194cd"
    ]
  }
}
```

Figure 5.8.: JSON payload used for the Traceroute measurement request

5. Evaluation

Figure 5.8 shows the entire JSON payload for the measurement request.

Table 5.5 shows the overall results. The same factors as before - *Measurements Coverage*, *Results Coverage*, and *Dish Metadata Coverage*, were chosen to evaluate the functioning of the system.

Results again show complete measurement execution and result download coverage. All measurements were executed and their results were downloaded successfully. Probe C, connected to a Starlink connection, returned 12 metadata records for an execution time of 11 seconds. Data indicates that this is mostly due to an internal rounding error with timestamps. More importantly, no record was outside the boundary values of execution timestamps.

Table 5.5.: Traceroute measurement evaluation results

	Probes				
Traceroute Results	Probe A	Probe B	Probe C	Probe D	Probe E
Measurements Scheduled	11	11	11	11	11
Measurements Executed	11	11	11	11	11
Measurement Coverage	100%	100%	100%	100%	100%
Measurement Results Returned	11	11	11	11	11
Measurement Results Coverage	100%	100%	100%	100%	100%
Total Execution Time	22s	92s	11s	22s	22s
Dish Metadata Records	-	-	12	-	-
Dish Metadata Coverage	-	-	100%	-	-

Hop results for the Traceroute measurements are visualized in Figure 5.9.

Results show that the probes located in Germany - Probes A, B, C had path lengths of 19-20 to the target server located in central Europe. Probe D, located in the USA had a higher path length in the range of 25-26. The highest path length of 27 was recorded for Probe E, located in India.

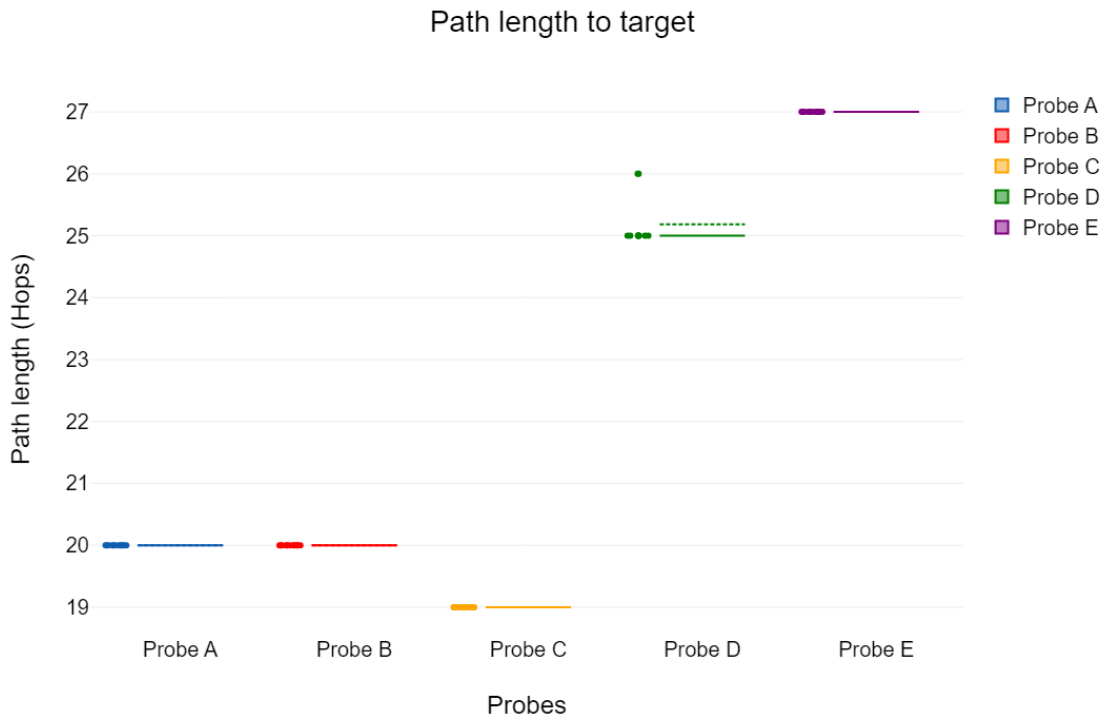


Figure 5.9.: Traceroute hops to target for all probes

5.2.3. Arbitrary Measurement - iRTT

For the Arbitrary measurement, iRTT was chosen to demonstrate the ability to create custom measurements. iRTT is a measurement tool used to test round-trip time, delay, packet loss, etc. It performs all testing between a server and a client, both of which come bundled with the tool.

For the evaluation, an iRTT server was setup within internal infrastructure that can only be accessed by Probe C. Hence, only Probe C was selected to run the measurement. For the client, a Docker container was built using an *alpine linux* image as the base image and iRTT was installed on top of it. By using this container, the one-off measurement was created with the payload shown in Figure 5.10.

```
{
  "type": "ARBITRARY",
  "measurementSpecification": {
    "durationInMinutes": 5,
    "containerImagePath": "cmnetworkplatform/irtt",
    "containerEntrypointString": "sh",
    "cmdInputStrings": [
      "-c",
      "irtt client -d 10s 131.159.25.81:64381 >> /home/output.txt"
    ],
    "outputPath": "/home",
    "addLinuxNetworkAdminCapability": "false"
  },
  "description": "iRTT mmt-thesis",
  "repeatSpecification": {
    "numberOfRepeats": 0,
    "interval": 180
  },
  "probeSpecification": {
    "probeIds": [
      "64c9f3571d52ec12b0f2feb4"
    ]
  }
}
```

Figure 5.10.: JSON payload used for the iRTT measurement request

After the measurement execution, results were saved to the file *output.txt* as specified within the measurement request above and were uploaded successfully from the Probe service to the Connector service. For the duration of the measurement, satellite dish metadata was also extracted.

The measurement results are shown in Figure 5.11. They show that the iRTT client within Probe C successfully connected to the server and all tests were run without any issues. Round trip times were consistent with both the devices sharing the same local network.

5. Evaluation

```
[Connecting] connecting to 131.159.25.81:64381
[131.159.25.81:64381] [Connected] connection established
seq=0 rtt=525µs rd=370µs sd=154µs ipdv=n/a
seq=1 rtt=435µs rd=223µs sd=212µs ipdv=89.6µs
seq=2 rtt=444µs rd=238µs sd=206µs ipdv=8.42µs
seq=3 rtt=468µs rd=243µs sd=225µs ipdv=24.8µs
seq=4 rtt=573µs rd=350µs sd=223µs ipdv=105µs
seq=5 rtt=454µs rd=244µs sd=210µs ipdv=119µs
seq=6 rtt=398µs rd=202µs sd=196µs ipdv=56.2µs
seq=7 rtt=320µs rd=182µs sd=138µs ipdv=78µs
seq=8 rtt=446µs rd=238µs sd=208µs ipdv=127µs
[131.159.25.81:64381] [WaitForPackets] waiting 1.72ms for final packets
seq=9 rtt=439µs rd=220µs sd=218µs ipdv=7.83µs
```

	Min	Mean	Median	Max	Stddev
	---	----	-----	---	-----
RTT	320µs	450µs	445µs	573µs	67.7µs
send delay	138µs	199µs	209µs	225µs	29.4µs
receive delay	182µs	251µs	238µs	370µs	61µs
IPDV (jitter)	7.83µs	68.4µs	78µs	127µs	46.4µs
send IPDV	1.92µs	27.9µs	14µs	70.5µs	26.3µs
receive IPDV	5.15µs	57.4µs	42.2µs	147µs	50.8µs
send call time	57.2µs	146µs		432µs	104µs
timer error	70.4µs	553µs		1.08ms	339µs
server proc. time	8.1µs	9.19µs		13.4µs	1.56µs

```
duration: 9s (wait 1.72ms)
packets sent/received: 10/10 (0.00% loss)
server packets received: 10/10 (0.00%/0.00% loss up/down)
bytes sent/received: 600/600
send/receive rate: 533 bps / 533 bps
packet length: 60 bytes
timer stats: 0/10 (0.00%) missed, 0.06% error
```

Figure 5.11.: iRTT results

5.3. Evaluating System Performance

5.3.1. Measurement Execution Time

Every time a probe runs a measurement, it records local timestamps for start time and end time of the execution. The time delta results in total measurement execution time.

Based on both the Ping and Traceroute measurements performed in Section 5.2 above, results showed severe discrepancies in total execution time.

Table 5.6.: Execution time differences for measurements

	Probes				
Execution Time Results	Probe A	Probe B	Probe C	Probe D	Probe E
Ping Measurements Executed	11	11	11	11	11
Total Ping Execution Time	44s	104s	33s	33s	33s
Traceroute Measurements Executed	11	11	11	11	11
Total Traceroute Execution Time	22s	92s	11s	22s	22s

Taking a look at Ping times first, Probes C, D, and E perform the best with a total time of 33 seconds each. This is ideal and in line with the standard Ping utility sending one packet every second. However, the Raspberry Pi 3 based probe, Probe B shows extremely poor performance. Second, Traceroute values yet again show degraded performance for Probe B, performing significantly worse than the other probes.

Overall, looking at both sets of results, there is a severe lack of performance in the Raspberry Pi 3 based probe. The lack of performance was found to be due to very high CPU usage when the inner Docker container was spun up to execute the measurements. This occurred both during the image pull process and the actual measurement execution process. In combination with the CPU bottleneck, memory usage was also extremely high. Possible solutions to improve performance are discussed in Chapter 6(Section 6.2).

5.3.2. API Throughput and Response Time

To measure the end-user REST API throughput, Apache JMeter [65] was used. JMeter is a load and performance testing tool used extensively to test web applications.

5. Evaluation

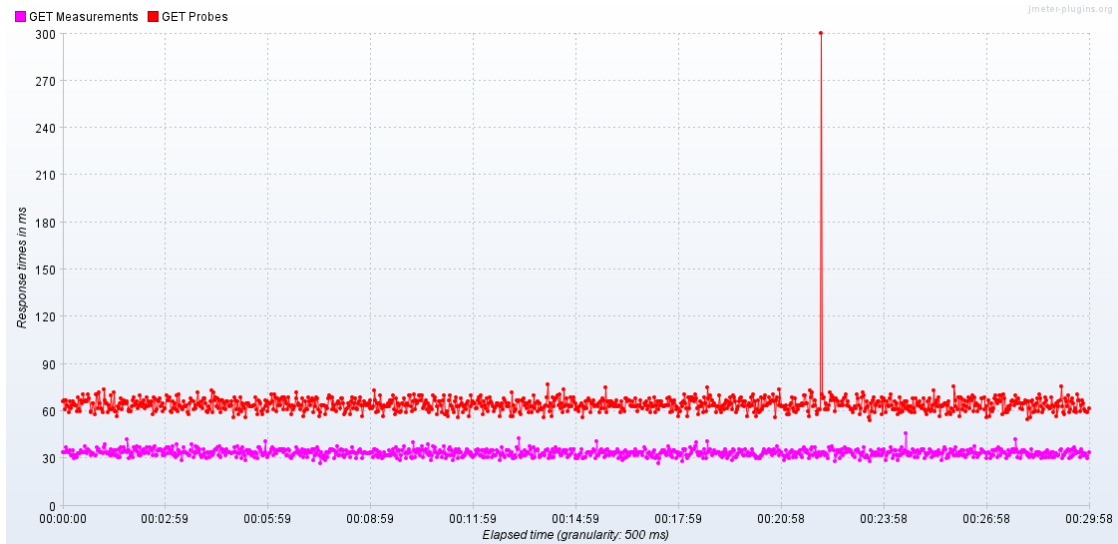


Figure 5.12.: Response times over time for 30m load

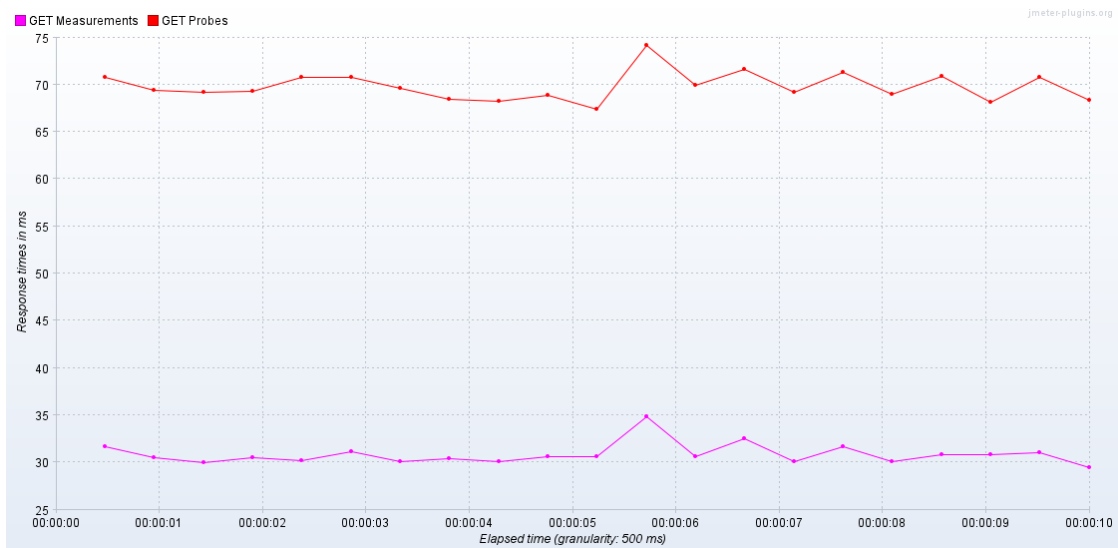


Figure 5.13.: Response times over time for 10s burst load

5. Evaluation

For testing consistent load over time, 1000 threads were run, ramping up uniformly over a period of 30 minutes. The endpoints targeted were *GET Probes* and *GET Measurements*. Pagination was implemented for both endpoints on the API service. Throughput was found to be *33.4 requests/min* with *0% error*. The response time graphs are shown in Figure 5.12. Average response latency for the *GET Probes* endpoint was *64ms* and *33ms* for the *GET Measurements* endpoint with only one outlier for the former being in the 300ms range.

For testing burst loads, 1000 threads were run, ramping up quickly over 10 seconds. Same endpoints were targeted again. Throughput was found to be *99.9 requests/min* with *0% error*. The response time graphs are shown in Figure 5.13. Average response latency for the *GET Probes* endpoint was *69ms* and *30ms* for the *GET Measurements* endpoint.

6. Conclusion

6.1. Findings

This section describes the current status and overall findings of the work done as part of this thesis.

First, we revisit the functional and nonfunctional requirements of the system as listed in Chapter 3(Section 3.2), and state their completeness. Completeness is assessed based on whether a requirement has been fully implemented or not.

Table 6.1.: Status of requirements completeness

Requirement	Status
FR1 - Support for standardized Predefined measurements	Implemented
FR2 - Support for user created Arbitrary measurements	Implemented
FR3 - Support for retrieving satellite dish metadata	Implemented
FR4 - Support for storage and retrieval of measurement results data	Implemented
NFR1 - Ease of use for probe hosting	Partially Implemented
NFR2 - Restart Policies	Implemented
NFR3 - Result Re-uploads	Not Implemented
NFR4 - Security - Encryption	Implemented
NFR5 - Security - Role Based Access Control	Implemented
NFR6 - Security - Unauthenticated and Unauthorized Access	Implemented
NFR7 - API Rate Limiting	Implemented
NFR8 - Extensibility	Implemented
NFR9 - Docker	Implemented
NFR10 - Raspberry Pi Hardware	Implemented
NFR11 - Deployment	Implemented
NFR12 - Results - Formatting	Implemented

6. Conclusion

Table 6.1 shows that all requirements except for NFR1 and NFR3 were fully implemented and tested. Full coverage of functionality was achieved.

Second, we focus on the system evaluation performed in Chapter 5. Evaluation has shown that the overall functionality of the system was in line with expectations and all of the objectives discussed in Chapter 1(Section 1.2) were realized. First, deployment of all services was carried out successfully and did not result in any unforeseen problems. Second, all measurements of both supported types, Predefined, and Arbitrary were run successfully on the test deployment infrastructure without any errors during execution. Additionally, all the result data for the measurements created was retrieved and was formatted as expected without anything missing. Finally, even Starlink dish metadata was extracted successfully from probes that were connected to an active Starlink connection during measurement execution.

However, evaluation also shows that overall system performance was not as expected on low compute performance probes like the Raspberry Pi 3.

Finally, we compare features of existing Internet measurement platforms with the system developed as part of this thesis. Table 6.2 shows the comparison.

Table 6.2.: Measurement platforms feature comparison

Feature	Measurement Platforms				
	This System	RIPE Atlas	EdgeNet	SpeedChecker	SamKnows
Interface	REST API	Web UI + REST API	Kubectl	Web UI + Mobile Client(s)	Web UI + Mobile Client(s)
Standardized Measurements	No	Yes	No	Yes	Yes
User Defined Preset Measurements	Yes	Yes	No	Yes	No
User Defined Custom Measurements	Yes	No	Yes	No	No
Run Measurements on Own Probes	Yes	Yes	Yes	Yes	Yes
Run Measurements on Others Probes	Yes	Yes	Yes	Yes	No
Public Measurement Data Available	Yes	Yes	No	Yes	No
Satellite Connectivity Metadata	Starlink Only	No	No	No	No
Cellular Connectivity Metadata	No	No	No	Yes	No

To elaborate further on some features selected:

1. *Standardized Measurements* here indicate that these measurements cannot be modified by users. Their types and parameters are chosen by the platform itself.
2. *User Defined Preset Measurements* are measurements created by users that have preset types such as Ping, Traceroute, etc. whose parameters can be modified.

3. *User Defined Custom Measurements* are measurements created by users with their custom code or logic packaged within a script or a container.
4. *Satellite Connectivity Metadata* refers to metadata that includes data such as dish angle, dish obstruction data, dish elevation data, etc.
5. *Cellular Connectivity Metadata* refers to metadata that includes data such as cellular signal strength, cellular signal channel, etc.

6.2. Limitations

There were several limitations found that are important to consider for future work. This section lays out some of the most important ones and describes possible solutions to overcome them.

- **Performance and optimization:** Evaluation showed significant performance issues for the Probe service running on the Raspberry Pi 3 device. During both measurement runs, performance was worse compared to the other probes. Both CPU usage and memory usage were found to be extremely high when the Probe service spun up the inner Docker container for measurement execution. The first ever measurement run took a little longer than the rest of the runs due to the initial image pull.

There are some steps that can be taken to improve overall performance and reduce measurement execution time. First, even though logging is minimal on the probes, reducing it further to only record errors may improve service performance. Second, reducing overall Class footprint may result in lesser JVM heap and non-heap memory usage. Setting limits on JVM memory usage may help in improving performance here. Finally, pre-pulling the Predefined measurement Docker container on setup will reduce time taken for the pull on first measurement execution.

- **Lack of requirements completeness:**

Section 6.1 of this chapter showed that two specific nonfunctional requirements, NFR1, and NFR3 were not implemented completely.

NFR1 - Ease of use for probe hosting was only partially implemented. The ability for an end-user to host a probe was made possible through access to an API endpoint which allowed the user to download a probe deployment configuration file. This file was based on *docker compose* and required the user to make some property modifications within. Instructions for performing these

actions were embedded within the file. While this was a working method, it was extremely unintuitive from an end-user perspective. This was exacerbated by the lack of a full fledged user interface or website which meant that the user had to first download the file, open it, go through every single instruction, make modifications, and then finally spin up the container. A better solution would have been to offer the user a downloadable script which upon running would take user input interactively, download the configuration, modify it based on user input and automatically spin up the container. As an alternative, if a user interface were to be developed in the future, this process could be offloaded to that instead.

NFR3 - Result re-uploads was not implemented. This was found to be a fairly important limitation for the overall reliability of the system. If any Probe service instance or container were to crash during measurement execution for any reason at all, even though the container attempts to restart and reconnect automatically, it does not attempt to re-upload results for the previously executing measurement. This was found to be linked to the way results are currently processed on the probe. For Predefined measurements, all result data is only stored in memory. It is not persisted to disk. For Arbitrary measurements, all result data is stored on disk. This prevented a common solution from being developed for attempting a result re-upload. This issue could, in the future, be resolved by persisting all results to disk on the probe and validating on restart if previous result data was sent successfully or not. If not, an attempt could be made to re-upload the result thereby improving the reliability of the system.

- **Lack of measurement failure information:**

At present, Arbitrary measurements are designed to work based on code packaged within Docker containers. The system does not check for the correctness of this code or any data that is present within the container. It merely runs the container for the user specified duration and uploads results based on the user specified output path. For any reason, if the container were to not run correctly, or failed to start running, there is no debugging information supplied back to the user. The only way for the user to check for a successful run is by downloading the results for the measurement. Providing failure information back to users would allow them to fix any unresolved issues occurring as a result of faulty code within their containers and would greatly improve user experience. As an added benefit of doing this, it would also enable them to stop repeat-measurements preemptively and would save overall system resources too.

6.3. Future Work

This section describes future work and development that remains.

6.3.1. User Interface

Currently the system is only offered with a Swagger based web interface [66] that uses an OpenAPI specification [67]. This interface can be used to interact with the system. It can also be used to view API documentation.

While the system is perfectly usable in this state, it is not user friendly and combines operations that both a User role and an Admin role can perform, in one single web page. Due to the nature of the Swagger interface, users also have to manually look up API details and payload schemas when they first interact with the interface.

Another issue is the lack of information or guidance on performing the several operations that the system supports. For a User role, these would be authentication and authorization, fetching probes and their information, or creating measurements and fetching their results. For an Admin, two such operations would be user and probe creation.

Due to these limitations, a likely next step would be to develop a more robust user interface or a dashboard that not only allows for separate workflows for both a User and an Admin, but also one that is more user friendly and allows for displaying all the information on how one would use the system, in one single place.

6.3.2. Continuous Integration and Deployment Pipeline

Another future goal would be to implement a CI/CD pipeline to handle the entire DevOps [68] workflow from code changes to testing and release.

During the development of this system, all tasks including planning, code changes, testing and validation, packaging, and service deployments were handled manually. This was challenging due to the nature of the services made to run on heterogeneous hardware and infrastructure, coupled with measurement execution being Docker container based. For instance, a lack of a CI/CD pipeline meant that container image builds had to be executed manually every single time there was a change to the relevant code base. Additionally, testing on different hardware and various network conditions

was a rigorous process too.

With a CI/CD pipeline, most of these challenges can be overcome and would reduce overall development and maintenance time significantly. Unit tests and API tests that exist currently can be automated to check for correctness on code changes. Testing can be performed on cloud based infrastructure, configured to simulate heterogeneous hardware and network conditions. Packaging and deployment can be automated once tests run successfully. Configuration mapping and secrets for all the services could be handled more efficiently and securely.

6.3.3. Enhancements

Even though the system in its current state satisfies all the functional requirements, it is still lacking in features and more functionality. Over time, adding more of them and enhancing existing ones would benefit the system and its users.

1. **Packet capturing:** Currently, when a user wants to execute an Arbitrary measurement on a probe, custom code can be added to perform packet capturing in the background. Packet capturing or sniffing is often used to analyze networks and identify potential network performance issues including packet loss or network congestion. Even though a user can add this functionality manually to measurements, supporting this directly as an optional feature when a measurement is created would benefit users and help them save time. An additional benefit would be the existing nature of the archival of measurement results. Packet capture data from measurements could be stored along with measurement results for reproduction and analysis in the future.
2. **Tagged probes:** In its current state, when users fetch probes and their details, they have to manually look up if a probe is connected to a satellite Internet connection or what network interfaces it supports. These can either be a wired network interface, a wireless interface, or a cellular interface. Repeating this task for multiple probes is a daunting process. Adding support for tagged probes would allow users to filter and search for probes that are tagged with a certain *key* quickly. For example, a user wanting to execute measurements only on probes connected to a cellular interface could quickly filter for these probes by looking up the *cellular* tag. This would save time and would make the system more user friendly.
3. **Rate limiting with credits:** In Chapter 4(Section 4.4), measurement rate limits were discussed. They limit consumption of resources in the overall system

by preventing users from creating measurements that span an extremely long duration or from creating measurements on an extremely large number of probes. This helps in managing overall system resources more efficiently and is an attempt to give users equal access to them. In combination with some of these limits, support for user credits could be introduced. First, creating measurements would require a certain number of credits. Second, users choosing to host probes would accumulate credits based on their probe host time. Third, these users would then use these accumulated credits to perform measurements or send them to other users for measurements created by the latter. This would incentivize users to host probes and contribute to the overall system growth. In combination with existing rate limits, this would also help in maintaining overall health and access to system resources.

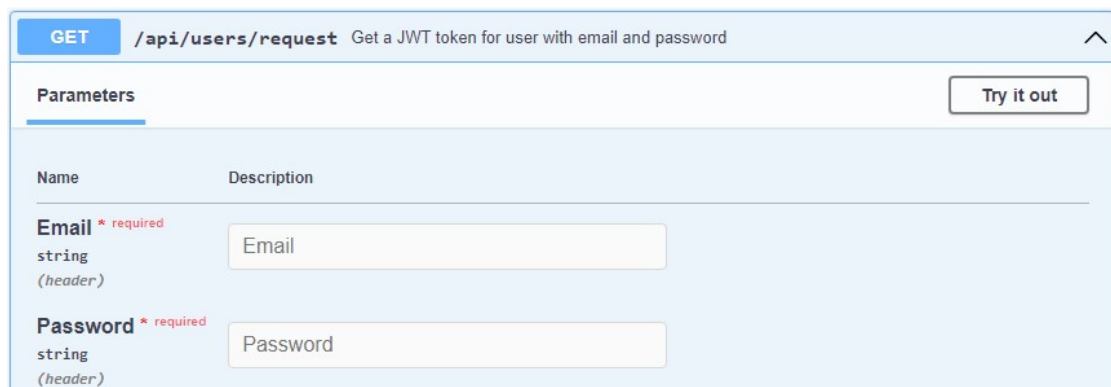
4. **Securing services with SSL:** All inter-service communication at present is secured via the means of an API key. Encrypting this communication would be a key security milestone. Since all the services use the Spring framework, adding SSL support with the help of SSL Bundles [69] would add an extra layer of security. The evaluation deployment as part of this thesis was based on isolated infrastructure that was highly controlled with access limitations. However, for a production deployment, SSL security would be a necessity.
5. **Add IPv6 support:** Currently, all Predefined measurement types support only targets based on the IPv4 address family. With IPv6 access growing year by year, it would greatly benefit new research focused on both adoption and coverage. In addition to this, there is no automatic network discovery on the Probe services to check if IPv6 is supported on the client side. Adding this functionality would help users filter for probes that support IPv6 before they execute their measurements.

A. Reproducibility

This appendix lists the steps required for reproducing user operations on the newly developed system, such as user authentication and authorization, fetching probes and their information, measurement creation, and measurement result downloads.

A.1. Login

First, in order to perform operations on the API web interface, a user needs to login and retrieve a JWT for authentication and authorization. Figure A.1 shows the login API endpoint.



The screenshot shows an API endpoint configuration interface. At the top, it displays the method **GET** and the endpoint `/api/users/request` with a description: "Get a JWT token for user with email and password". A "Try it out" button is located in the top right corner. Below this, there is a "Parameters" section with a table listing the required parameters:

Name	Description
Email * required string (header)	<input type="text" value="Email"/>
Password * required string (header)	<input type="text" value="Password"/>

Figure A.1.: The login API endpoint for fetching a JWT

If the request is successful, a server response containing the Bearer JWT is returned as part of the response headers, as shown in Figure A.2. The Bearer JWT must be used for every subsequent API call to the system.

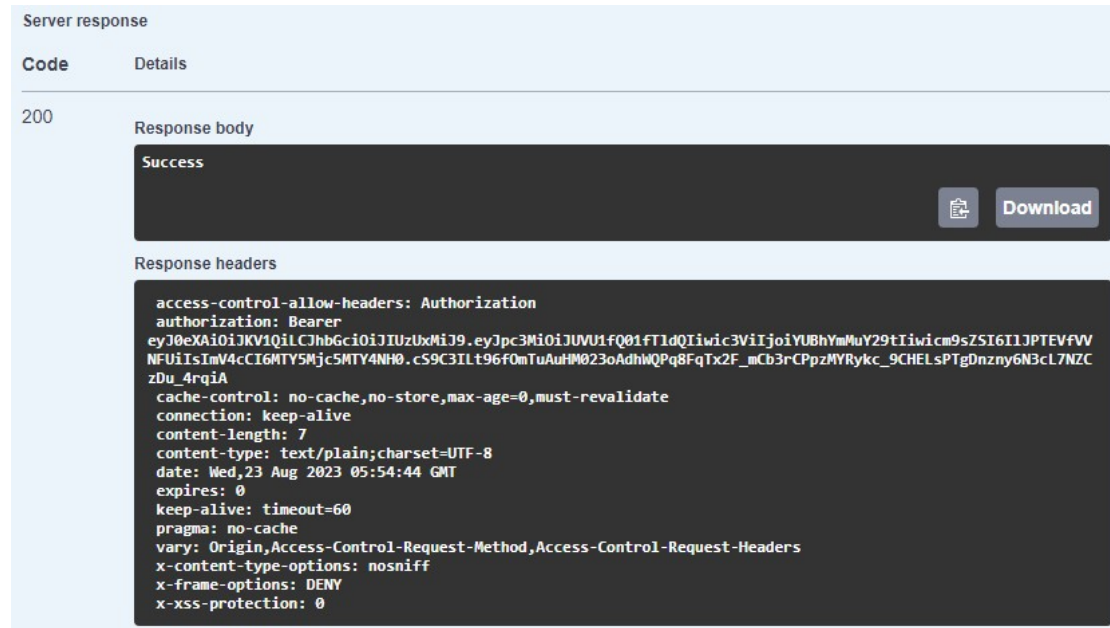


Figure A.2.: A successful login response containing a Bearer JWT

A.2. Fetching Probes and their Information

In order to choose the probes for measurement execution, their details have to be fetched first. Figure A.3 shows the API endpoint for this.

Probes can be filtered based on their country and status.

The response is paginated and will contain a list of all probes based on filter criteria. Probe information returned will include their IDs which will be used during measurement creation.

Probe ^

GET /api/probes Get all probes with a filter on country and probe status ^

Parameters Try it out

Name	Description
country string (query)	<input type="text" value="country"/>
status string (query)	Available values : CONNECTED, RUNNING, DISCONNECTED <input type="text" value="--"/>
page integer(\$int32) (query)	Default value : 0 <input type="text" value="0"/>
size integer(\$int32) (query)	Default value : 10 <input type="text" value="10"/>

Figure A.3.: The API endpoint to get all probes and their details

A.3. Measurement Execution

Once probes and their IDs have been chosen, the next step is to create a measurement.

The request body of this API call takes in either a Predefined measurement schema or an Arbitrary measurement schema. Other parameters can be modified too, and all available parameters and their details can be found in the Schema section of the API web interface.

Figure A.4 shows the API endpoint for creating a measurement with the request body containing example values.

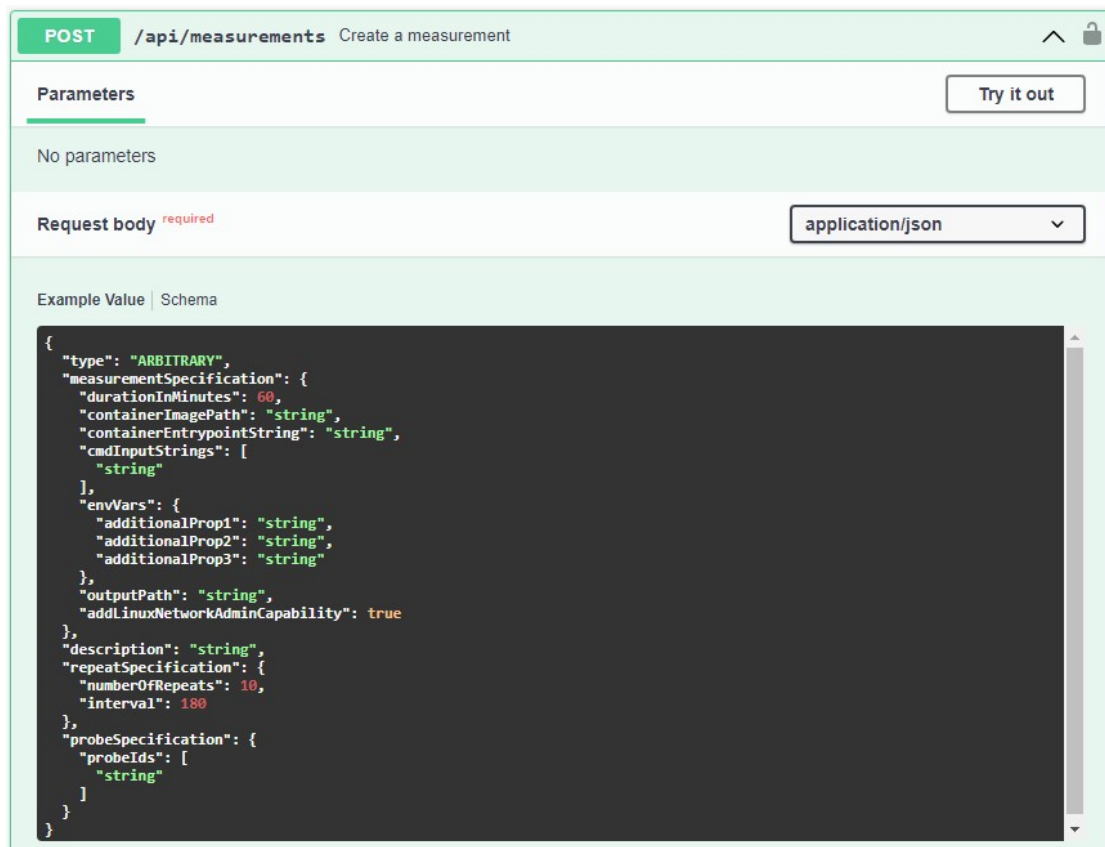


Figure A.4.: The API endpoint to create a measurement

Once a measurement has been created successfully, a response containing the measurement ID is returned. Figure A.5 shows one such response.

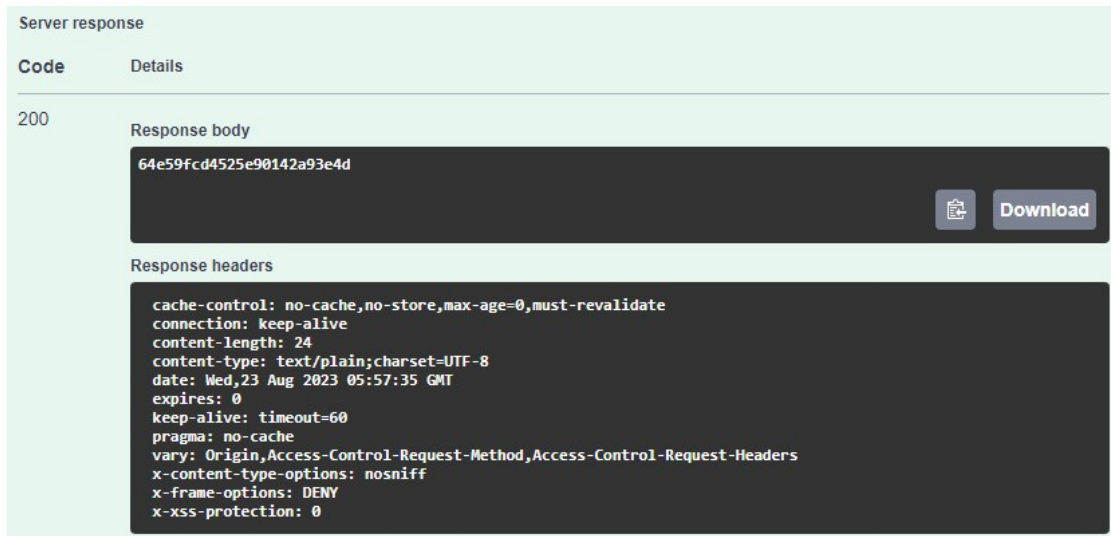


Figure A.5.: Successful measurement creation response

A.4. Result Downloads

After the measurement has been created and executed, the next step is to download the measurement results. Figure A.6 shows the API endpoint for this.

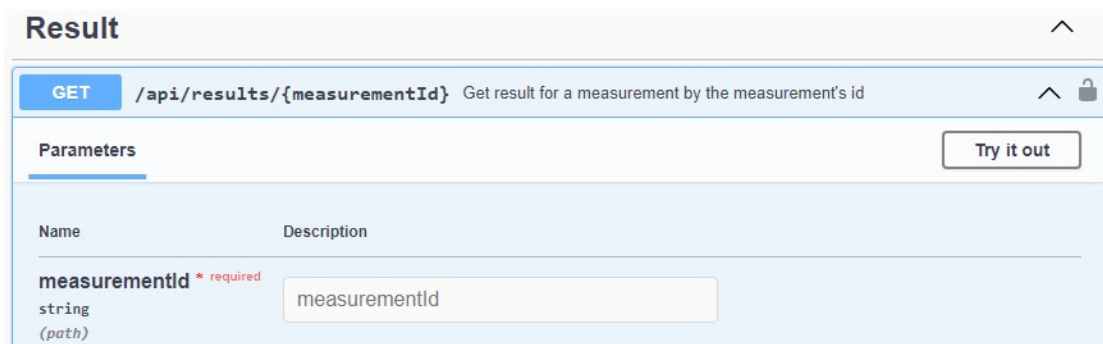


Figure A.6.: The API endpoint to get results for a measurement

One single ZIP archive file is downloaded as part of a single measurement result. In the case of a Predefined measurement, the results are bundled in a single JSON file within. In the case of an Arbitrary measurement, the results are directory based. The ZIP archive file can optionally contain another JSON file containing the satellite dish metadata gathered on the probes during the measurement execution.

Abbreviations

API	Application Programming Interface
CD	Continuous Deployment
CDN	Content Delivery Network
CI	Continuous Integration
DNS	Domain Name System
FR	Functional Requirement
FURPS	Functionality, Usability, Reliability, Performance, Supportability
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
IPv4	Internet Protocol Version 4
IPv6	Internet Protocol Version 6
iRTT	Isochronous Round-Trip Tester
ISP	Internet Service Provider
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
JWT	JSON Web Token
NFR	Nonfunctional Requirement
NTP	Network Time Protocol
QoE	Quality of Experience
QoS	Quality of Service
RBAC	Role-Based Access Control
REST	Representational State Transfer
RFC	Request for Comments
RTT	Round Trip Time
SCRAM	Salted Challenge Response Authentication Mechanism
SDK	Software Development Kit
SQL	Structured Query Language
SSL	Secure Sockets Layer
UML	Unified Modeling Language

List of Figures

2.1. RIPE Atlas Architecture	7
3.1. Proposed System	11
3.2. Fetch Probes Sequence	16
3.3. Create Measurement Sequence	16
3.4. View Measurements Sequence	17
3.5. Fetch Measurement Results Sequence	18
4.1. Top Level System Design	20
4.2. API Service Architecture	22
4.3. Connector Service Architecture	24
4.4. Probe Service Architecture	26
4.5. Sysbox Docker in Docker Approach	27
4.6. One-machine Service Deployment	31
4.7. Distributed Service Deployment	32
5.1. Deployment Diagram	35
5.2. Ping Measurement JSON Payload	37
5.3. Ping RTT Values for Probe A	39
5.4. Ping RTT Values for Probe B	39
5.5. Ping RTT Values for Probe C	39
5.6. Ping RTT Values for Probe D	40
5.7. Ping RTT Values for Probe E	40
5.8. Traceroute Measurement JSON Payload	41
5.9. Traceroute Hops To Target	43
5.10. iRTT Measurement JSON Payload	44
5.11. iRTT Results	45
5.12. API Response Times Over Time	47
5.13. API Response Times Over Time - Burst Load	47
A.1. API Login	56
A.2. API Login Response	57
A.3. Get Probes and their Details	58

List of Figures

A.4. Create Measurement	59
A.5. Create Measurement Response	60
A.6. Get Measurement Result	60

List of Tables

5.1. Services Deployment Machine	34
5.2. Database Deployment Machine	35
5.3. Probes Deployment	36
5.4. Ping Evaluation Results	38
5.5. Traceroute Evaluation Results	42
5.6. Execution Time Results	46
6.1. Requirements Completeness	49
6.2. Feature Comparison	50

Bibliography

- [1] SpaceX. *Starlink*. 2023. URL: <https://www.starlink.com/technology>.
- [2] gRPC. 2023. URL: <https://grpc.io/>.
- [3] ping. 2023. URL: <https://linux.die.net/man/8/ping>.
- [4] traceroute. 2023. URL: <https://linux.die.net/man/8/traceroute>.
- [5] nslookup. 2023. URL: <https://linux.die.net/man/1/nslookup>.
- [6] P. Heist. *Isochronous Round-Trip Tester*. 2023. URL: <https://github.com/heistp/irtt>.
- [7] iPerf. 2023. URL: <https://iperf.fr/>.
- [8] Docker. 2023. URL: <https://www.docker.com/>.
- [9] R. P. Foundation. *Raspberry Pi*. 2023. URL: <https://www.raspberrypi.com/>.
- [10] L. Gupta. *What is REST?* 2022. URL: <https://restfulapi.net/>.
- [11] RIPE Atlas. 2023. URL: <https://atlas.ripe.net/about/>.
- [12] RIPE Atlas Network Coverage and Statistics. 2023. URL: <https://atlas.ripe.net/results/maps/network-coverage/>.
- [13] RIPE Atlas Measurement Definitions. 2023. URL: <https://atlas.ripe.net/docs/getting-started/user-defined-measurements.html#definitions>.
- [14] RIPE Atlas Public Data. 2023. URL: <https://atlas.ripe.net/docs/getting-started/what-is-ripe-atlas.html#public-data>.
- [15] *The WebSocket API*. 2023. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API.
- [16] RIPE NCC Staff. "RIPE Atlas: A Global Internet Measurement Network." In: *The Internet Protocol Journal - Volume 18, Number 3*. 2015.
- [17] Apache HBase. 2023. URL: <https://hbase.apache.org/>.
- [18] MapReduce. 2023. URL: https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html.
- [19] Kubernetes. 2023. URL: <https://kubernetes.io/>.

- [20] *kubelet*. 2023. URL: <https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/>.
- [21] *kubectl*. 2023. URL: <https://kubernetes.io/docs/reference/kubectl/>.
- [22] *SpeedChecker*. 2023. URL: <https://www.speedchecker.com/>.
- [23] *SpeedChecker Applications and SDKs*. 2023. URL: <https://www.speedchecker.com/products/mobile-apps-and-sdks.html>.
- [24] *SpeedChecker Speed Test Data*. 2023. URL: <https://www.speedchecker.com/products/speed-test-datasets.html>.
- [25] *SpeedChecker Cellular Coverage Data*. 2023. URL: <https://www.speedchecker.com/products/cellular-coverage-datasets.html>.
- [26] *SamKnows*. 2023. URL: <https://www.samknows.com>.
- [27] *SamKnows History*. 2023. URL: <https://www.samknows.com/company/history>.
- [28] *SamKnows Tests*. 2023. URL: <https://www.samknows.com/tests>.
- [29] *SamKnows Traceroute*. 2023. URL: <https://www.samknows.com/tests/traceroute>.
- [30] *mtr*. 2023. URL: <https://github.com/traviscross/mtr>.
- [31] *SamKnows Agents*. 2023. URL: <https://www.samknows.com/technology/agents>.
- [32] *SamKnows Test Targets*. 2023. URL: <https://www.samknows.com/technology/test-targets>.
- [33] B. Bruegge and A. H. Dutoit. *Object Oriented Software Engineering Using UML, Patterns, and Java*. Prentice Hall, 2010.
- [34] *JSON*. 2023. URL: <https://www.json.org/json-en.html>.
- [35] *Spring Boot*. 2023. URL: <https://spring.io/projects/spring-boot>.
- [36] *Spring*. 2023. URL: <https://spring.io/>.
- [37] *MongoDB*. 2023. URL: <https://www.mongodb.com/>.
- [38] *Spring Data MongoDB*. 2023. URL: <https://spring.io/projects/spring-data-mongodb>.
- [39] *MongoDB - JSON and BSON*. 2023. URL: <https://www.mongodb.com/json-and-bson>.
- [40] *BSON*. 2023. URL: <https://bsonspec.org/>.
- [41] *GridFS*. 2023. URL: <https://www.mongodb.com/docs/manual/core/gridfs/>.
- [42] *MongoDB - Advantages of NoSQL Databases*. 2023. URL: <https://www.mongodb.com/nosql-explained/advantages>.

- [43] N. Hunter. *Role-Based Access Control for a complex enterprise*. 2023. URL: <https://delinea.com/blog/role-based-access-control-for-a-complex-enterprise>.
- [44] *PBKDF2 Algorithm*. 2023. URL: <https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/crypto/password/Pbkdf2PasswordEncoder.SecretKeyFactoryAlgorithm.html#PBKDF2WithHmacSHA256>.
- [45] *Linux Capabilities*. 2023. URL: <https://jwt.io/>.
- [46] B. Kaliski. *PKCS 5: Password-Based Cryptography Specification Version 2.0*. 2000. URL: <https://www.ietf.org/rfc/rfc2898.txt>.
- [47] M. B. Jones, J. Bradley, and N. Sakimura. *JSON Web Token (JWT)*. 2015. URL: <https://www.ietf.org/rfc/rfc7519.txt>.
- [48] *Docker Compose*. 2023. URL: <https://docs.docker.com/compose/>.
- [49] *Spring TaskScheduler*. 2023. URL: <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/scheduling/TaskScheduler.html>.
- [50] *alpine - Docker Hub*. 2023. URL: https://hub.docker.com/_/alpine.
- [51] *Sysbox*. 2023. URL: <https://github.com/nestybox/sysbox>.
- [52] *Sysbox Security*. 2023. URL: <https://github.com/nestybox/sysbox/blob/master/docs/user-guide/security.md>.
- [53] *Docker Hub*. 2023. URL: <https://hub.docker.com/>.
- [54] *Docker run reference - CMD*. 2023. URL: <https://docs.docker.com/engine/reference/run/#cmd-default-command-or-options>.
- [55] *Docker run reference - ENV*. 2023. URL: <https://docs.docker.com/engine/reference/run/#env-environment-variables>.
- [56] *Docker run reference - VOLUME*. 2023. URL: <https://docs.docker.com/engine/reference/run/#volume-shared-file-systems>.
- [57] *Linux Capabilities*. 2023. URL: <https://man7.org/linux/man-pages/man7/capabilities.7.html>.
- [58] *Paris Traceroute*. 2023. URL: <https://paris-traceroute.net/>.
- [59] G. Bartolomeo, S. Bäurle, N. Mohan, and J. Ott. "Oakestra: an orchestration framework for edge computing." In: *SIGCOMM '22: Proceedings of the SIGCOMM '22 Poster and Demo Sessions*. 2022, pp. 34–36. DOI: 10.1145/3546037.3546056.
- [60] *MicroK8s*. 2023. URL: <https://microk8s.io/>.

- [61] G. Bartolomeo, M. Yosofie, S. Baurle, O. Haluszczynski, N. Mohan, and J. Ott. "Oakestra white paper: An Orchestrator for Edge Computing." In: 2022. DOI: 10.48550/arXiv.2207.01577.
- [62] *Oakestra Networking Component*. 2023. URL: <https://github.com/oakestra/oakestra-net>.
- [63] *Oakestra Cluster Deployment*. 2023. URL: <https://www.oakestra.io/docs/getstarted/get-started-cluster/>.
- [64] *MongoDB SCRAM Authentication*. 2023. URL: <https://www.mongodb.com/docs/manual/core/security-scram/>.
- [65] *Apache JMeter*. 2023. URL: <https://jmeter.apache.org/>.
- [66] *Swagger UI*. 2023. URL: <https://swagger.io/tools/swagger-ui/>.
- [67] *OpenAPI Specification*. 2017. URL: <https://spec.openapis.org/oas/v3.0.1>.
- [68] *What is DevOps?* 2023. URL: <https://www.atlassian.com/devops>.
- [69] *Spring SSL Bundles*. 2023. URL: <https://spring.io/blog/2023/06/07/securing-spring-boot-applications-with-ssl>.