Networked Systems
ETH Zürich — seit 2015

Automated power measurement for network devices: Collecting data reliably made simple?

Bachelor Thesis

Author: Jonathan Chung

Tutor: Dr. Romain Jacob

Supervisor: Prof. Dr. Laurent Vanbever

February 2024 to July 2024

Abstract

Network equipment runs all the time and thus has all day power draw. To gain insights, data collection is needed and possible via protocols such as SNMP. However, this requires a considerable amount of setup and detailed communication with network equipment owners. Data quality is device-specific and may not be comparable. One way to make data collection easier and comparable is to measure devices externally with a power meter connected to the power supply unit of a device under test. We developed an easy-to-deploy system called "Autopower" which collects power data from power meters connected to Raspberry Pies that are controlled over the internet. To test our system, we measured three routers on campus.

Contents

Chapter 1

Introduction

1.1 Motivation

Sustainable networking, and specifically power draw becomes increasingly important in times of climate change. Since networking equipment, such as switches and routers, runs 24/7, knowing the real-world power draw is important for further research. Power data helps to validate hypotheses and aids in drawing conclusions which could finally lead to more efficient devices. Power measurements can be collected and published from multiple locations and then made publicly available, like a RIPE atlas [1] for power, the community thus getting open access to power data.

Although it is possible to obtain power data[2] via protocols like SNMP[3], the data may be too coarse-grained, untrustworthy, or not comparable between different vendors. Moreover, getting access to SNMP data might require additional, complicated setup on the operators' side, such as setting up accounts for externals, which could be considered a security risk. Even if access via SNMP is possible, the data is usually not very detailed (e.g., measurements only from 5-minute intervals) or not comparable to other devices due to different ways of measuring power.

SNMP may not be suitable for all kinds of measurements: The effect of short spikes on the power draw could be averaged out if the measurement interval is long. Increasing the interval increases the load of the device, which is not favorable as it could decrease throughput and increase power draw. For more fine-grained measurements, dependence on the network device and SNMP is not optimal.

Power measurements should be comparable between multiple devices and vendors to allow comparisons between different devices. However, if the power measurement is not always between the grid and PSU, the real devices' power draw remains unknown, making measurements incomparable between devices. One network device may for example measure power draw behind the PSUs making inefficiencies of the PSU not visible, while another one measures before the PSU, where inefficiencies of the PSU are visible. Thus, using an external power meter is advantageous.

Setting up power measurements should be made easy as this lowers the barrier for deployment, facilitates data collection and, we hope, results in giving us more quality data.

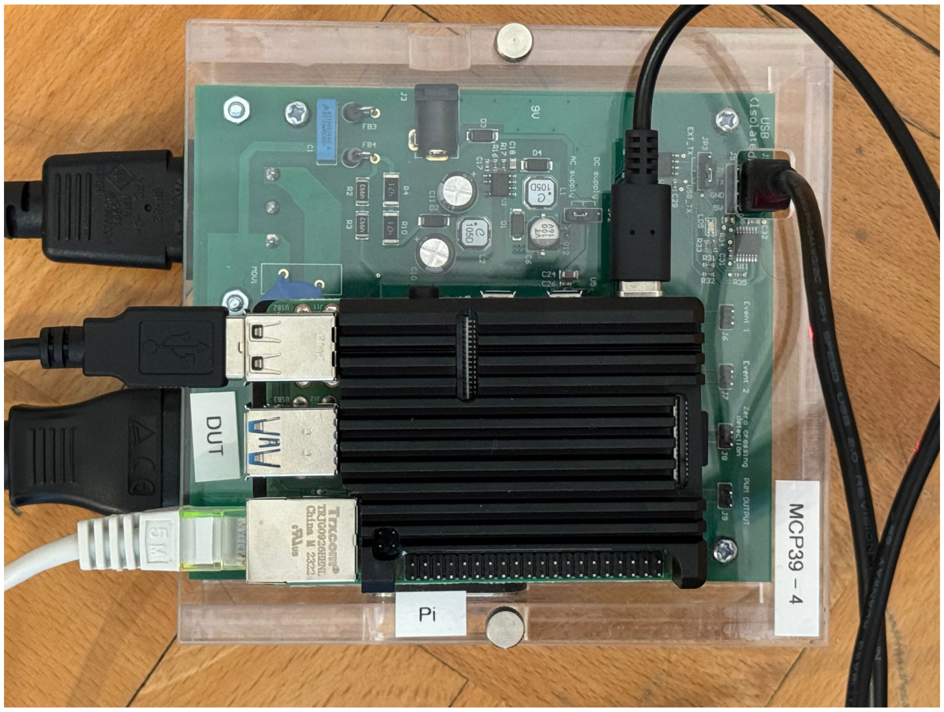


Figure 1.1: Autopower setup with Raspberry Pi and power meter

1.2 Requirements

To collect data successfully, we defined three goals a measurement system must satisfy:

- Reliability: Data points should not be lost. A short to medium network outrage should not lead to data loss. Additionally, we should be notified in event of failure which implies monitoring.
- Easy deployment: The solution should require only minimal setup. Most importantly, a data center provider should not need to configure anything on the devices under test. Deployment should be plug-and-play.
- Security: While the measurements may not be safety critical[1], data should be transmitted securely, and management should only be possible for authorized users.

To fulfill these requirements, we developed a custom power measurement system.

1.3 Autopower system

Similar to the probes used for the RIPE atlas[4], we developed a Raspberry Pi based device we call "Autopower" (Figure 1.1, Example deployment setup in Figure 4.1), based on custom software written in C++ and Python, which controls a power meter[2]. The measurement data (power in mW and timestamp) are stored in a database and periodically uploaded via gRPC [7] from the Autopower device to a server. Thanks to gRPC, each Autopower device can be controlled over the

[1]Evaluation of the data shows that work times can be seen in the power measurements (4.1.1). If this leaks information depends on the setting.

[2]MCP39F511N based power monitor demonstration board [5] controlled by the PinPoint software [6]

network, even if it is behind a NAT. This means that such a device can be easily deployed in a data center but also at home behind a NAT, to measure all kinds of (network) devices.

1.4 Overview

Chapter 2 describes the background with regard to related work, while Chapter 3 focuses on design choices and a detailed look at the system design. Chapter 4 evaluates the project, and Chapter 5 briefly describes further work and gives a short summary.

Chapter 2

Background

2.1 Related work

2.1.1 Core projects

We learned from Arlitt et al. [8] that considering data loss should play a fundamental role in the initial design. This is rooted in our project requirements (1.2). Even though we consider data loss as unavoidable, the design should take precautions to make it as unlikely as possible.

Our approach focuses on power measurement and is still noticeably smaller than that of Arlitt et al. who focus on traffic monitoring. Besides thinking about considering data loss, the core goals and design choices of our project are based on their lessons learned (as described in their Section 8[8]):

- "Think[ing] long term": We wrote detailed documentation for the deployment of Autopower devices and documented the project structure in README files.
- "Automat[ing] as many tasks as possible": A core goal of our project was automation. This is also why we called the project Autopower. Devices register and start measurements automatically; We set up continuous integration for building our C++ programs and Zabbix sends monitoring alerts via Slack in case of failures on the Autopower devices.

Our project would not have been possible or at least far harder if PinPoint [6] [9] had not existed. We use PinPoint as data source for communicating with the power meter. Even though the goal of PinPoint is energy profiling, the application can be applied easily to our use case.

2.1.2 Data publication and design ideas

The RIPE Atlas [1] provides traffic-related data, while our project operates similarly with respect to power. We deploy Autopower devices, which in case of the RIPE atlas are called probes [4]. Both projects deploy these devices at distributed locations to measure traffic or power data with the goal of publishing them to be freely available.

Boavizta [10] provides the Energizta [11] tool "to collect and report open data on server energy consumption". Adding our data to a project like this would approach the goal of providing a publicly available power dataset.

Chapter 3

Design

3.1 Prototype and design choices

To satisfy the requirements (1.2) and design objectives, we designed the Autopower system based on a client-server architecture where data is stored as backup on the clients and periodically uploaded to the server. To collect measurements, it would suffice to obtain data from the power meter, write it to a file and then periodically upload the files to some server. We based our initial prototype on a simple bash script doing just that. While this even allows secure data transmission via file transfer protocols such as SFTP, controlling the clients would in any case require custom software. A file-based approach would require a lot of custom logic on both the client and server side and would possibly not scale well or even become overwhelming.

A more sensible approach seemed to be one using a database together with a set of custom-designed software - especially with web-based tools used to control the Autopower devices. Even though setting up a database and writing custom code are initially more complex, this approach permits more expandability later on.

Our requirements led to a few design choices:

- For reliability, data should be locally stored on the client and on the server side. This allows recovery by re-uploading the data from the clients if the server has experienced data loss. We chose a relational database, namely PostgreSQL [12] since it is widely used and feature-rich. Local storage also allows data to be buffered during network outages.
- To monitor clients, we set up a Zabbix [13] server, which alerts us in case of failures.
- The encrypted connection for management and data uploads is enabled by the gRPC framework. Management commands are authenticated via a password by the server for security.
- An Autopower device must be secured, since it may be exposed directly to the internet: Thus, it is protected by a firewall and SSH only allows key-based authentication.
- We use PinPoint to collect measurements and use syscalls to manage PinPoint. Unfortunately, controlling PinPoint needs the client software to fork a new process and then manually obtain the measurement output from stdout via a pipe. There is currently no API for PinPoint. We consider this as a tolerable inconvenience.

- Once the device is deployed and has internet access, it automatically registers with the server. The Autopower device can now be controlled from the server side, even if the device is behind a NAT. Registering from the client side allows us to deploy an Autopower device that can start measuring immediately without needing to configure an IP on the server side.
- For ease of deployment, we wrote a deployment script to install all necessary programs and set up an Autopower device [14].

3.2 General design

This section gives an overview of the general design of the Autopower system.

Autopower consists of multiple applications and has a client and a server part. The client parts are installed on the Raspberry Pi, while the server side is hosted on a virtual machine in our data center.

Figure 3.1 shows the system as a graphical overview.

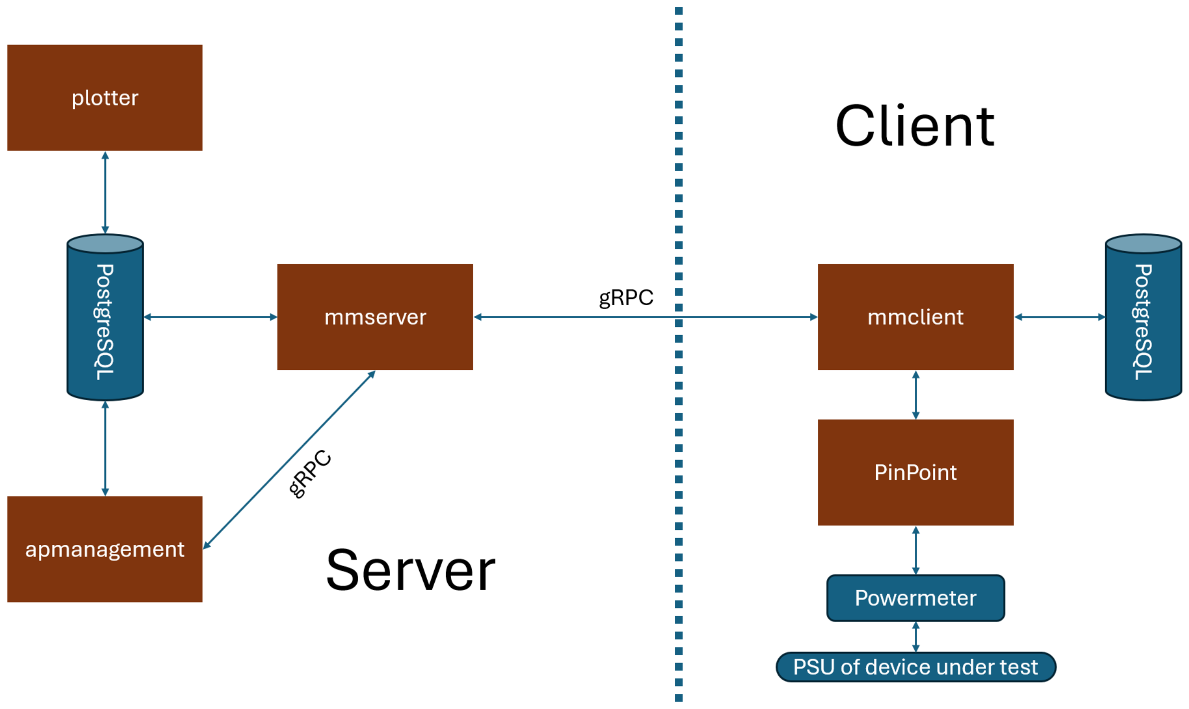


Figure 3.1: Autopower System overview

The server consists of three and the client of two logical parts:

- `mmclient`: A client-side application written in C++ which controls the PinPoint [9] software. PinPoint communicates with the power meter, and mmclient writes this output to the client-side database. mmclient also connects to the server via gRPC to receive commands and periodically uploads measurement samples from the database.
- `PinPoint` [9]: Client-side application used to fetch the measurement samples (Power in mW and timestamp) from the power meter.

- `mmserver`: Server-side application written in Python, which receives and saves data from the measurement clients. mmserver also issues and relays commands from management clients[1] to the respective Autopower device.
- `Plotter`: Server-side web application that plots data from each Autopower device grouped by device under test (DUT). Written in Python and based on Plotly [15].
- `apmanagement`: Server-side Flask application to manage the server-side database and issue commands to mmserver which relays them to the Autopower clients.

3.3 Detailed design

To store the measurement data points, we obtain the measurement value (power in mW) and a timestamp per measurement sample from PinPoint. For a reliable and extendable design, we decided to use a database instead of files containing samples.

3.3.1 A database instead of files

Storing measurements in a database instead of files allows querying via SQL. Thus, if users have basic knowledge of SQL, they can do data analysis directly on the data by connecting to the database without the need to consult external programs. If external programs such as Grafana[16] are needed in future, they can be integrated easily[17] assuming they support PostgreSQL.

Furthermore, to handle large amounts of data, we consider files less suited than a database. We would need to handle files manually and implement custom logic to ensure that we did not upload duplicates on the client side. File organization - especially on the server side where we receive data from multiple devices - would most likely become overwhelming.

Moreover, databases are designed for concurrency[18]. This means, for example, that concurrent inserts are possible. Multiple clients can upload to the server concurrently, and on the client side concurrent writes and uploads without needing to wait on each other are supported by default.

The client and server use a PostgreSQL database. Using PostgreSQL is mostly arbitrary, as any relational database (e.g., alternatively, MySQL[19]) would fit in the design. Since PostgreSQL is very feature-rich and widely used, the choice settled on this database management system (DBMS).

3.3.2 Database design

While the client side consists of two tables (`measurements` and `measurement_data`), the server side is more involved. The client saves measurement samples with timestamp and measurement value plus metadata grouping multiple samples to a measurement. In contrast, the server additionally stores the devices under test, runs, log messages, and client IDs of clients that were connected at least once.

ER Diagram and table description

The following ER diagram visualizes the database schema of the server. The client side only consists of the `measurements` and `measurement_data` relation and is omitted for brevity. Interested readers can examine the SQL code in the Autopower repository under [20].

[1]3.3.4 defines management clients

The tables include:

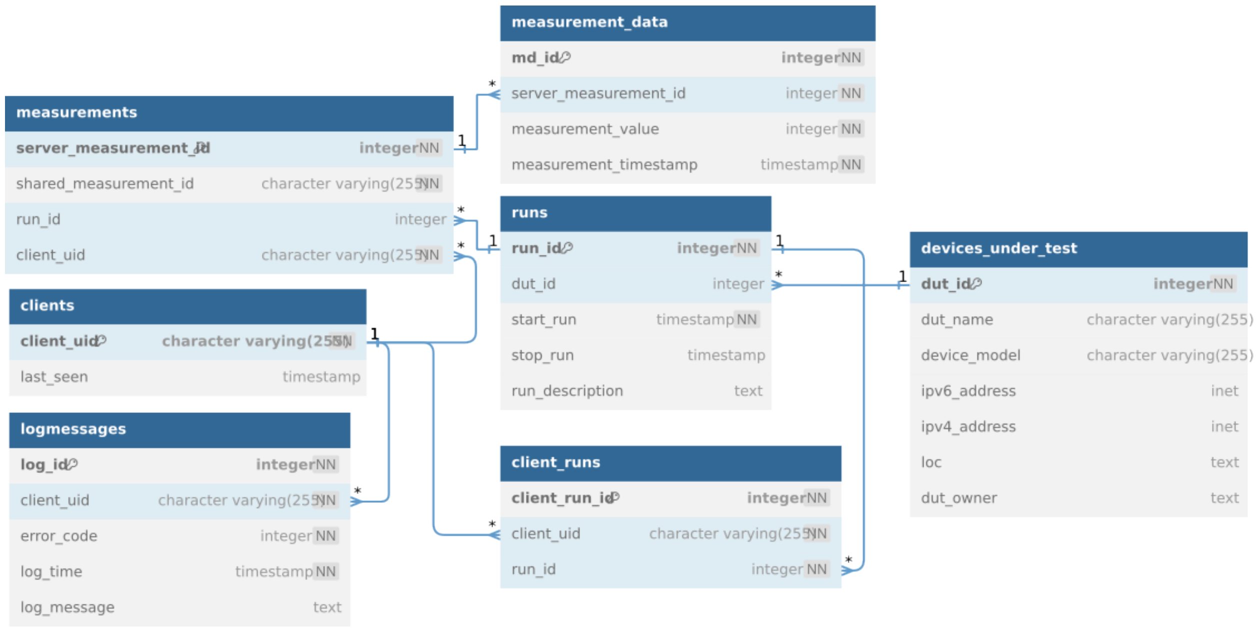


Figure 3.2: Entity relationship diagram of server database

- `measurement_data`: stores the samples with measurement value and timestamp from each device. On the client, this table additionally has a boolean field `was_uploaded` storing if a sample was uploaded successfully to the server.
- `measurements`: metadata of a measurement. Links shared identifier (`shared_measurement_id`) of the clients to a numeric, internal measurement id. A measurement may belong to some run that can be linked.
- `devices_under_test`: Represents devices under test (DUTs) with metadata such as IP addresses, locations, owners, etc.
- `run`: a collection of measurements of one DUT within a given time frame
- `client_run`: links runs and clients. Used to have this relationship even if no measurement yet exists (can happen if the client has not yet been deployed). Allows restricting which clients can participate in a run. Not further discussed.
- `clients`: Stores clients seen from the server with a timestamp where this client last interacted with the server. Not further discussed.
- `logmessages`: Stores error messages issued by Autopower clients. Not discussed in detail.

Measurement

A set[2] of continuous samples with the same settings at one Autopower device are grouped as measurement. Each measurement has a unique, shared measurement ID (along with an internal,

[2]For simplicity, we use the notion of a set here. To be more precise, this could also be a multi-set or bag, since there are no guarantees that a measurement sample is not duplicated. We assume that PinPoint does not produce such duplicates.

non-shared measurement ID on the server and the client side respectively). The shared measurement ID is generated by the client and, assuming each client name is unique, is also unique across the whole system.

Device under test

Network devices may have multiple power supply units; thus, we chose to deploy multiple power meters per device under test (DUT). As a result, multiple Autopower devices may need to be logically connected in the database. This is done via the `devices_under_test` relation.

Runs

Since measurement settings such as the sampling interval could change while measuring the same DUT or a power outrage lead to a new measurement once the Pi restarts, we introduced the notion of a "run" as another layer of abstraction. Runs correspond to a deployment at a device under test and link measurements together[3].

Shared measurement ID

To link server-side and client-side measurements, we use an ID which is unique over the whole system. For simplicity, calculating this ID should not rely on a consensus protocol or connection to the server. Also, the ID should always be known as soon as a new measurement starts to not introduce unknowns (NULLs) which introduce complexity in the database on the server and client side. If the server receives new samples, there may not be a measurement in the database for this device. Thus, the server must create a new measurement in its local database as soon as the first sample is received. To link with the client, this measurement uses the shared measurement ID.

Even though the client and server have numerical internal IDs for measurements, they are only unique per device but not unique over the whole system. For simplicity, a measurement is uniquely identified with the shared measurement ID as soon as data is shared between the client and server. For data transmission, only the shared measurement ID is used[4].

A shared measurement ID is comprised of the name of the Autopower device, the system time at the start of the measurement, and a random number in the following format:

`<clientUid>_<StartTimestamp>_r<RandomNumber>`

where the random number is drawn from C++

`std::uniform_int_distribution(0,200)`.

By adding a random number to the ID, the probability of a clash in case of misconfiguration and the same system time is reduced. Clashes are considered unlikely and only as a result of misconfiguring

[3]After the initial design, we noticed that the name "run" is not expressive enough. It would be more precise to call this table a "deployment". However, changing the database table name would imply deep-reaching code changes, which we do not consider worthwhile for now.

[4]Our approach is just one way to link client and server side measurements, but others may exist. We could base the whole system on shared measurement IDs only, making the `measurement_data` table store the shared instead of the internal measurement ID. The shared measurement ID is a string but the internal ID is an integer. Using an integer instead of a string in the `measurement_data` table saves space. Alternatively, we could also use only internal IDs and device names.

two physical clients with the same name. A shared measurement ID can be shared with the server at any time and allows unique identification of a measurement.

Index for performance

Sometimes it is necessary to quickly obtain minima/maxima or ranges of a measurement based on some timestamp (e.g., to display them in the UI). Getting a maximum timestamp from ≈ 8 million samples - which corresponds to one measurement of our initial deployment of over a month (4.1) - takes more than a second without index. Therefore, we created an index on the `measurement_data` table, which tells the DBMS to access the timestamps and server measurement IDs more efficiently when using range queries: Getting the last measurement timestamp now only needs less than 1ms thanks to the index[5].

3.3.3 gRPC

We use gRPC [7] to manage clients and upload data from the client to the server. gRPC is a framework for remote procedure calls (RPCs). This provides an API close to function calls in C++ or Python, a clean API with encryption support, automatic retry on failure, and streaming. One major reason to select gRPC was streaming.

On a high level, gRPC allows the client to register with the server, and the server streams requests containing commands, like starting or stopping measurements, to the client. Additionally, the client streams measurement samples to the server.

Since the connection is client-initiated, NATs can be traversed easily, while the server can directly issue commands without needing client-side polling. Choosing gRPC with a client-to-server connection avoids the need to set IPs of the client on the server, making deployment easier.[6] Having encryption built into the transmission protocol is fundamental to satisfying our security requirement.

3.3.4 gRPC API description

Client methods

The most important remote procedure calls called by an Autopower client include:

- `registerClient()`: Method which is called by the client to introduce itself to the server. The server stores the client's name in the database and RAM. Afterwards, the server streams requests (e.g. starting a measurement) for all registered clients via this method.
  Once a client receives requests, it calls the respective client functions. If the `registerClient()` call terminates, the client assumes that the connection failed and re-tries to register again every 5 seconds.
- `putClientResponse()`: Method used to acknowledge and upload a response to some request from the server. Every response includes an ID (requestId) to link to the request from the server. A response includes the result of a server-initiated request.

[5]We used "EXPLAIN ANALYZE" to measure the execution time of fetching the maximum timestamp. With index, the reported execution time was 0.180ms. Without index, the reported execution time was 1368.881 ms. In PostgreSQL creating an index creates and uses B-Trees and thus benefits range queries like fetching minima or maxima since there is no need to scan all data points[21].

[6]This client-to-server connection currently only concerns the core part of the project: managing measurements and data upload. Zabbix monitoring is currently server-initiated but can also be changed to a client-initiated protocol.

- `putMeasurement()`: Uploads measurement samples to the servers' database. Clients call `putMeasurement()` periodically to upload data that has not yet been uploaded.
- `getMsmtSttngsAndStart()`: Gets the measurement settings of a client. Then the client will start a measurement based on those settings. Instead of starting a measurement by pushing the parameters with a request through `registerClient()`, we initially thought of allowing the client to update its local configuration based on what is saved on the server from time to time. Strictly speaking, splitting retrieval of measurement settings in a separate method is not necessary, as the parameters are currently tied to remain the same within a measurement.

Management client methods

To control Autopower devices, the server provides additional management methods. We call clients which can call these methods "management clients". Every management function call needs a password to prevent unauthorized clients from issuing measurement commands. The most important gRPC methods for management clients include:

- `setMsmtSttngs()`: Sets the measurement settings for some device, but doesn't start a measurement at the device yet.
- `issueRequestToClient()`: Forwards some request to the client e.g. stopping or starting a measurement at some Autopower device.

Management function calls are used by a CLI script and the Management UI (3.3.8) to issue requests to Autopower devices. The protobuf definition (proto file) serves as technical documentation for all message types and remote procedure calls. The proto file is in the Autopower repository [22].

3.3.5 Signing and authentication

To authenticate clients, each client (including management clients which also have an additional password) has a certificate, signed by a Certificate Authority (CA) on the server. The server only accepts clients with a certificate signed by this CA, while the clients accept any certificate trusted by their (local) certificate store as long as the domain matches. Not trusting any servers except for those who have a certificate signed by the CA can also be configured on the client side by setting the allowed CA on the clients, too.

This does introduce a level of complexity but is a tradeoff between security and convenience: Before a device can be deployed, its certificate needs to be signed on the server side and manually moved from the device containing the private key of the CA to the Autopower client. Re-signing must happen from time to time (e.g. every year) depending on how long a certificate is valid. Even though this makes the setup more complex, it brings the benefit of state-of-the-art encryption and authentication.

3.3.6 mmserver and mmclient

mmserver

The server implements the gRPC methods mentioned above and serves as a proxy from management clients to Autopower devices running mmclient. The most important classes are:

- `ExternalClient()`: Represents an Autopower device. An object of this class stores measurement parameters, such as the sampling interval, which can be retrieved by mmclient.

Moreover, it holds a queue of requests to be issued to the Autopower device and a dict mapping request IDs to a response to some request issued to this Autopower client

- `ClientManager()`: Stores a shared (static) dict of ExternalClient objects which have registered at the client. An object of this class can be used to communicate with an Autopower device.
- `CMeasurementApiServicer()`: Class which implements all gRPC method calls.

Implementation of control protocol

Since gRPC does not allow the server to call methods on the client directly, we resort to a custom approach to control client functions. The server streams requests to a registered client and the client responds by calling `putClientResponse()` on the server.

Registering at the server and sending commands

Before a device can receive commands, it registers at the server by calling the `registerClient()` RPC via gRPC.

In case of failure, the client re-tries to call this method periodically until it succeeds. `registerClient()` adds a new client to the `ClientManager()` and database. Thereafter, it consumes requests from the `ExternalClient()`'s queue corresponding to the registered client and streams requests to the client. Therefore, the `registerClient()` call mainly serves as a connection from the server to the client to issue commands.

Receiving responses from the client

After a request has been received and executed on the client, the client sends a response to the server by calling `putClientResponse()`. This adds the response message to the dict in the `ExternalClient()` corresponding to the client from which the response came. Since each request is numbered with a request ID, a response can be mapped to a request directly[7].

Proxying a request

New requests can be issued by management clients such as the Management UI. A management client calls `issueRequestToClient()` with a message to some client. If the management clients' password can be verified, this request is repacked, scheduled and sent to the respective Autopower device. Afterwards, it waits for at most 30 seconds for a response from the device and returns the response or an error if a timeout occurred. This response is then handled on the management clients' side.

Storing data

The client periodically calls `putMeasurement()` to upload measurement samples to the server. This method adds a new measurement and client to the database if it does not yet exist. Afterwards, the method inserts the measurement samples into the `measurement_data` relation.

[7]The request ID idea is an easy implementation of sequence numbers found in protocols like TCP but does not serve to detect reordering or lost packages since gRPC is assumed to be reliable.

If the server throws an exception or the connection is lost, gRPC returns an error. The client then assumes that the server could not write a sample and aborts. If the server did not write a sample correctly, it will not acknowledge a successful write to the client. Thus, the failed sample, will not be marked as being successfully uploaded in the clients' database. The client will then try to upload the sample again in the future.

Linking runs and measurements

While the first sample of a measurement incoming to the server can automatically create a server-side measurement, linking new measurements to runs is more involved. Assuming a run exists and it has a pre-specified start time earlier than the first incoming sample of a yet unseen measurement on the server, a run gets linked to this newly created measurement automatically. However, since our Raspberry Pies do not use a real-time clock and the server's and client's clocks may not be in sync, automatic linking may not always be successful. If automatic linking fails, the run field of the measurement remains NULL. Hence, a run should be linked manually to measurements.

mmclient

The client creates three threads, one process, and communicates via condition variables. While gRPC has an asynchronous API [23], it seemed more difficult to use. Therefore, we implemented the client synchronously and used threads to separate out concurrently running functionality. All threads run indefinitely until mmclient is stopped. The three threads consist of:

- `managementThread`: Registers with mmserver and receives commands from the server
- `uploadThread`: Periodically uploads measurements to mmserver if a measurement is running.
- `measurementThread`: Starts or restarts PinPoint as a separate process if the measuring mode is active, and monitors PinPoint. Also retrieves the measurement samples and writes them to the database.

Starting measurements from the client side

Measurements start from the client side as soon as mmclient starts by calling the `startMeasurement()` function on the client. Measurements use default parameters set in the local configuration file. More specifically, the client creates a new entry in its database for measurement metadata such as the shared measurement ID. Then it sets the client in measuring mode and notifies the `measurementThread` to start PinPoint. As soon as the thread calling `startMeasurement()` gets notified that a measurement sample has been written to the local database within $10 + \frac{\text{samplingInterval[ms]}}{1000}$ seconds, `startMeasurement()` terminates successfully. Despite being arbitary, we believe that this formula allows an appropriate time for PinPoint to start and mmclient to write one sample to the client database. If the timeout occurs, `startMeasurement()` returns an error.

Starting the measurement as soon as the mmclient starts and not waiting on the server means that right after plugging in an Autopower device, it can start measuring even if the server is unavailable. Not depending on the server means that an Autopower device could be deployed stand-alone and completely offline, making deployment feasible in more settings. This, however, implies that the server may not always know if a measurement is in progress. Management clients may not know the state of all clients at all times and thus cannot always reliably display the state of Autopower devices.

Controlling PinPoint

PinPoint is controlled by the `measurementThread`. This thread waits until the server is set to measuring mode via `startMeasurement()`. If the thread is notified that measurement should start, it sets up a Unix pipe and starts a new process to run PinPoint. Also it parses the content of the pipe from PinPoint and writes it into the client's database. After writing to the database, it checks if the client is still in measuring mode and kills PinPoint if this is no longer the case. However, if PinPoint exited or crashed while in measurement mode, `measurementThread` tries to restart PinPoint.

In its default configuration, PinPoint did not reliably support running without executing a workload and sometimes exited after the first measurement. We added a separate command line flag, which allows executing without a workload and opened a pull request to upstream [24].

Since PinPoint does not provide a library, we would need to implement the communication protocol with the power meter or PinPoint ourselves from scratch. We resorted to running PinPoint from our program via `fork()`, `exec()` as a separate process.

Depending on those functions is not ideal from a security standpoint as this poses the risk of being able to inject arbitrary parameters to PinPoint or even the system if not validated correctly. We validate measurement inputs/counters against a whitelist and the sampling interval to be a parsable integer before calling `exec()`, since those strings may come from external sources.

3.3.7 Deployment

Deployment should be thought of from two perspectives. First, from that of the person deploying an already set up Autopower device in a data center, and second, from that of the person setting up and testing the functionality of an Autopower device. While the first must be as easy as possible, the second can involve more steps.

A person setting up and testing the Autopower device is more knowledgeable about the project and most likely has access to the server-side components. We expect that this person knows how to flash the operating system of a Raspberry Pi and can run commands on the command line. For initial deployment, this person follows documentation[25] and runs a deployment script[14] on the Pi. Afterwards, they sign the clients' certificate and test connectivity with the server. This only needs to happen once per new Autopower device.

The person running or deploying in the data center should not need to access the shell of the Autopower device but can be expected to be able to read documentation and set up network connectivity. Deployment in the data center consists of plugging in the DUT into the power meter and connecting the Pi with the network.

In the best case, no changes to the network in the data center would need to be made. However, in practice, it turns out that some setup on the operator side may not be avoidable. For security reasons, the operator may want to use a separate sub-net or need to approve a new devices based on its MAC address before internet access is available. Thus, some setup cost in the data center is unavoidable.

Alternative internet access

To get around the network limitation, alternatively, one could use an externally controlled network like a cellular network. Whether a cellular signal is available or not in a data center depends on the location of the data center and is not always practical. Still, cellular is worth considering if the obstacles of adding the device to some data center network are too high. In their default configuration, the Pies get an IP via DHCP, so if DHCP is available, network connectivity is plug-and-play. Moreover, since the Pies have WiFi capabilities - even though not preferred due to unreliability - WiFi could also be used if configured as primary or backup connection.

The core application also works without internet access or internet access via NAT. A deployment can be made even if incoming connections to the Pies are not possible. Currently, monitoring via Zabbix and SSH access will not work in case of a NAT, but configuring a reverse SSH tunnel and configuring Zabbix agents to active agents which register to the Zabbix server could be possible in this case. Since the device starts measuring without internet access, even if no internet is available, the devices could be deployed but not controlled.

Scripts, encryption and documentation

Despite the need for authentication, deployment should be as easy as possible. While being a trade-off between security and convenience, encryption was a hard requirement. To facilitate this, our scripts and step-by-step documentation[25] on how to deploy an Autopower device are crucial. We wrote a deployment script[14] that needs to be run on the Raspberry Pi to set up a new device which automates initial setup.

Furthermore, for Zabbix monitoring, the deployment script automatically creates a PSK which needs to be set up on the Zabbix server. After a device has been set up for the first time, the Pi can be used for multiple deployments and may just need occasional software updates and maintenance. If the IP address changes (e.g. due to deployment in another location) the IP address for Zabbix monitoring must be changed. Further work could automate this. However, mmclient will register automatically once an Autopower device is deployed and registered, and the Management UI will display the IP address of the Autopower device making adding the IP in Zabbix easier. Accessing an Autopower device works via SSH by the SSH key specified during initial setup.

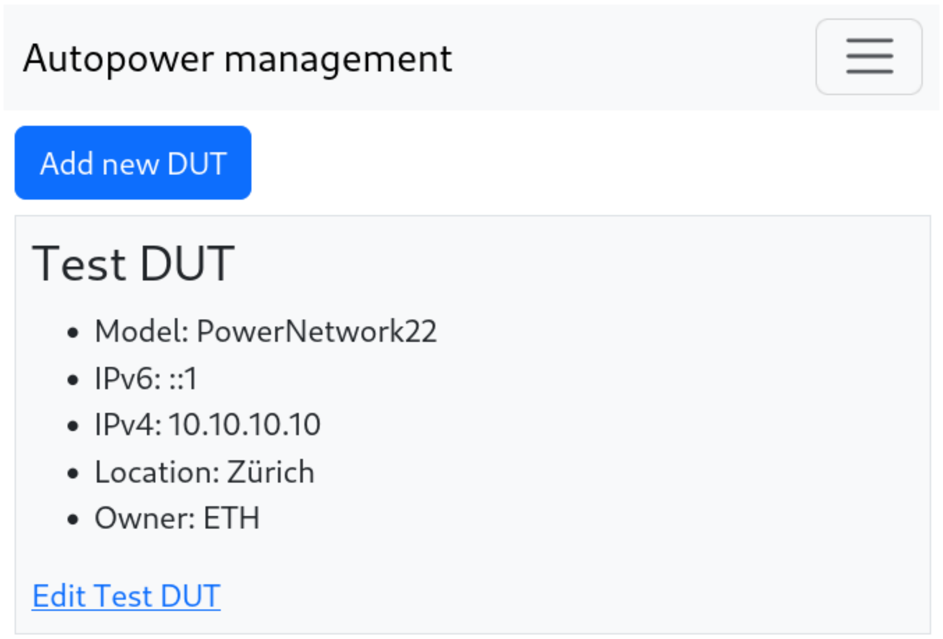
3.3.8 Server management: Management UI

Clients can be managed by a web app we call Management UI or apmanagement. This Flask-based UI controls the database and can issue commands to Autopower devices. The UI connects via gRPC to the server as a management client and can access the server-side database. While we expect that users issue more involved queries through the database directly, apmanagement also provides a GUI-based way to add and edit Runs (Figure 3.4) and DUTs (Figure 3.3) and link measurements to runs for quick access.

The main feature of apmanagement is starting, stopping and setting parameters for measurements at Autopower devices. Each Autopower device has a page (Figure 3.5) displaying past measurements and a form to start measurements with specific parameters, stop measurements and check connectivity to the clients. On this page, measurements can also be downloaded as CSV files.

gevent and hanging issues

Developing this web-based application with Flask turned out to be more challenging than initially expected. Since the app is I/O bound - it needs to query the database and the gRPC server - the worker type of the gunicorn server running the Python script should be gevent which allows

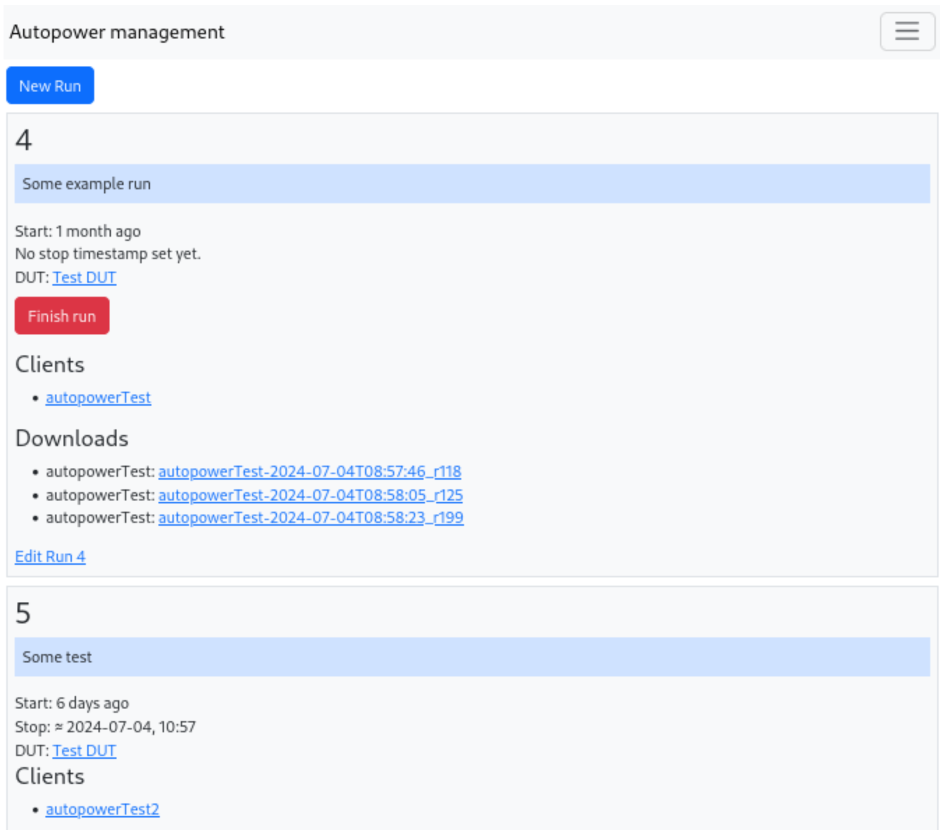


Test DUT

- Model: PowerNetwork22
- IPv6: ::1
- IPv4: 10.10.10.10
- Location: Zürich
- Owner: ETH

Edit Test DUT

Figure 3.3: Management UI: DUT management



4

Some example run

Start: 1 month ago
No stop timestamp set yet.
DUT: Test DUT

Clients

- autopowerTest

Downloads

- autopowerTest: autopowerTest-2024-07-04T08:57:46_r118
- autopowerTest: autopowerTest-2024-07-04T08:58:05_r125
- autopowerTest: autopowerTest-2024-07-04T08:58:23_r199

Edit Run 4

5

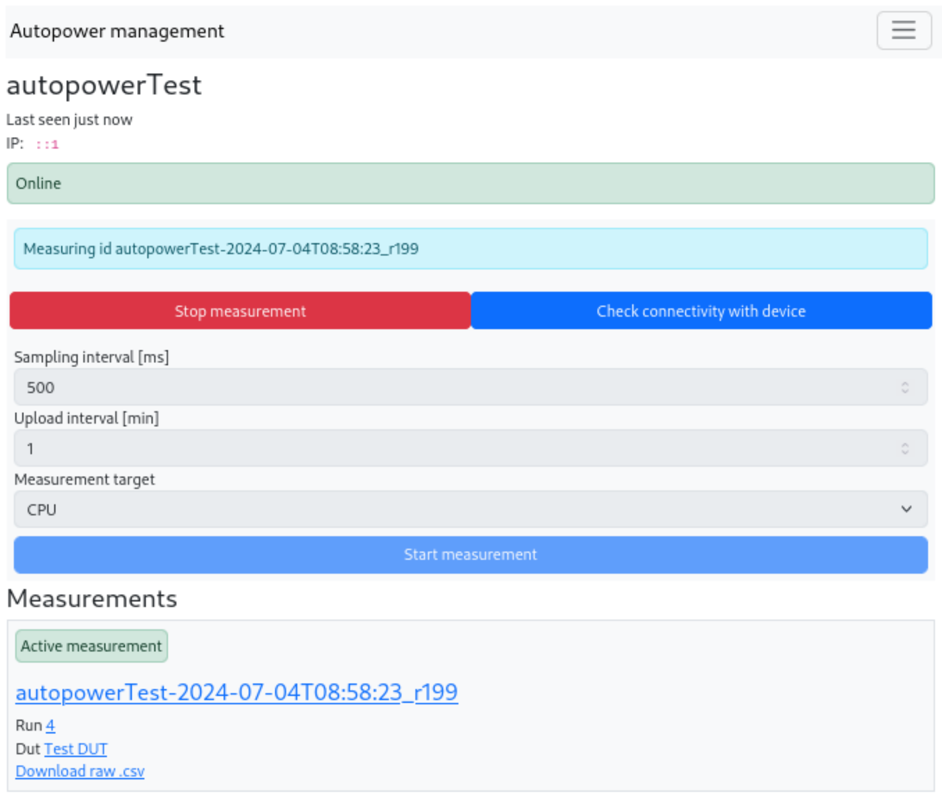
Some test

Start: 6 days ago
Stop: ≈ 2024-07-04, 10:57
DUT: Test DUT

Clients

- autopowerTest2

Figure 3.4: Management UI: Run management


Autopower management

autopowerTest

Last seen just now

IP: ::1

Online

Measuring id autopowerTest-2024-07-04T08:58:23_r199

Sampling interval [ms]: 500

Upload interval [min]: 1

Measurement target: CPU

Measurements

Active measurement

autopowerTest-2024-07-04T08:58:23_r199

Run 4

Dut Test DUT

Download raw .csv

Figure 3.5: Management UI: Device management

more concurrency. Using the default sync worker type is not recommended for an I/O bound application[26], as it could block the app - especially for longer streaming requests such as downloads which would only allow further work to be done once they were finished. Since gRPC and the database adapter psycopg2 are not compatible with gevent by default, the app started to hang after the first request which seemed odd since we had already moved away from the sync worker. It took some time and research to pin down the issue to this incompatibility and the solution that both libraries need "monkey patching" [27][28].

3.3.9 Plotter

To visualize the data, the Plotter displays the most recently incoming measurements per client (Figure 3.6) and allows analysis of past data by querying the database. The main page is refreshed periodically with the latest content available. Data visualization happens grouped by the device under test (Fig 3.7 shows the header of a device under test).

We plot measurements as time series and group weekly and hourly as separate plots (4.1.1).

It turns out that plotting the whole time frame takes multiple seconds in Plotly using Dash, thus, it is not recommended to plot long time frames. Initially, we considered plotting per Autopower device. However, we soon realized that it is more interesting to see all measurements of a device under test grouped together. Typically, seeing the total power a networking device draws gives more insights than seeing a single PSU, even though 4.1.1 shows that plotting per PSU gives the insight that PSUs have a different load distribution.

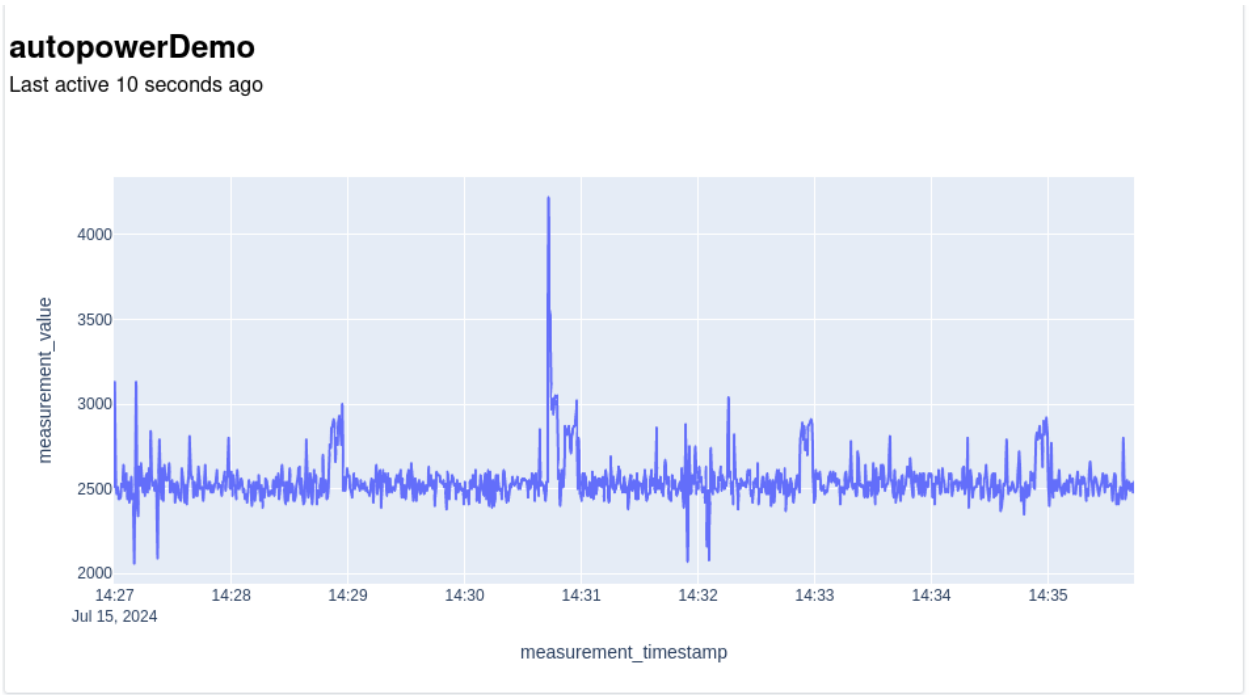
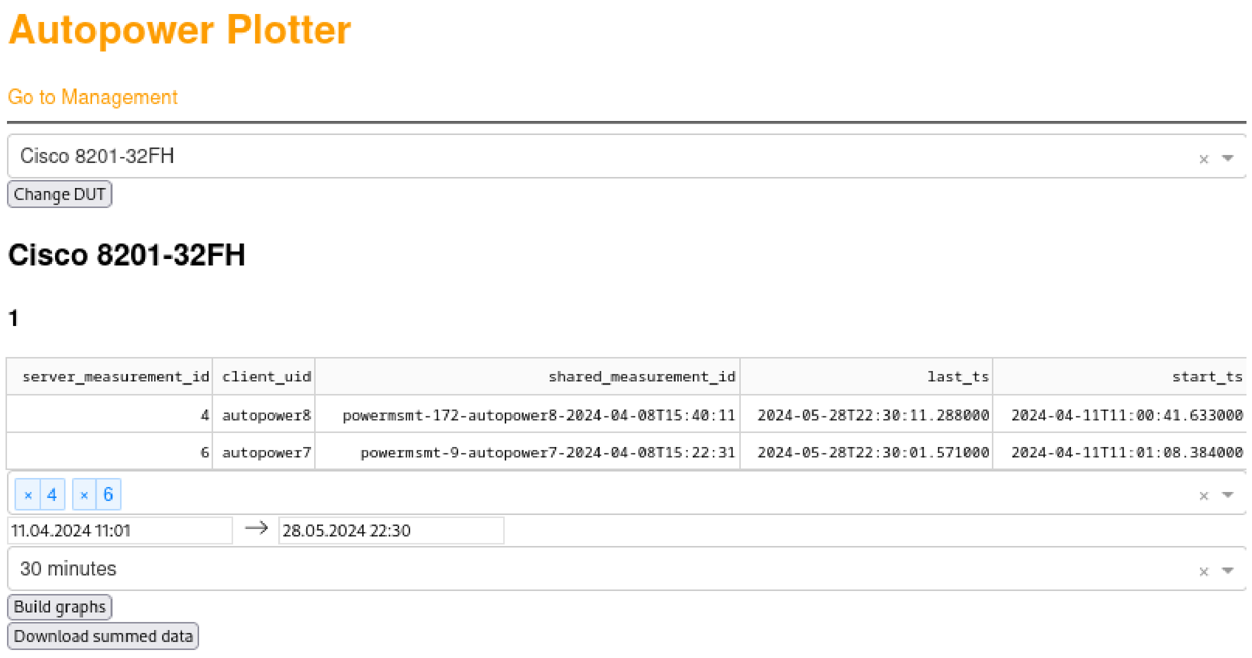


Figure 3.6: Plotter Homepage: A running measurement (demo)



Cisco 8201-32FH

Cisco 8201-32FH

1

server_measurement_id	client_uid	shared_measurement_id	last_ts	start_ts
4	autopower8	powermsmt-172-autopower8-2024-04-08T15:40:11	2024-05-28T22:30:11.288000	2024-04-11T11:00:41.633000
6	autopower7	powermsmt-9-autopower7-2024-04-08T15:22:31	2024-05-28T22:30:01.571000	2024-04-11T11:01:08.384000

× 4 × 6

11.04.2024 11:01 → 28.05.2024 22:30

30 minutes

Figure 3.7: Device under test header in Plotter

We removed the per-device plotting and replaced it with the possibility of selecting the measurements on the DUT page manually instead. This is expected to be more user-friendly and tailored to the actual use. Thanks to Plotly, the plots can be interactively zoomed in and analyzed.

Plotly, used for data analysis, does its job well, even though it is slow for large time intervals. Initially, some Management UI was integrated into the Plotter. Having more fine-grained control over the app led to the choice of completely factoring out management functionality into the Management UI. This more modular approach also meant that the Plotter could be deployed alone on a more powerful machine that only has access to the database without the need of tight protection with authentication as with the Management UI.

3.3.10 Monitoring

Reliability needs logging to pin down issues. Thus, the client notifies the server in case of errors. Those errors are logged in the `logmessages` table on the server but do not trigger a notification yet. To react quickly in case of failure, a broader monitoring solution is needed. We use Zabbix to monitor the OS of the clients. During initial installation, the deploy script installs zabbix-agent2, creates a PSK on this Pi, and sets a firewall rule such that only the configured IP of the Zabbix server can monitor the client. To act quickly, Zabbix passes errors to a Slack channel. Monitoring should be extended to monitor the database for log messages indicating errors and data loss.

Chapter 4

Evaluation

To evaluate the success of the project, we present the results of our first deployment together with a brief empirical overview of the measurement results to show the data we could collect with the system. Moreover, we discuss the lessons learned from this deployment. Afterwards, we discuss how far we fulfilled the requirements and what is still outstanding for a more complete system. Moreover, we describe how we profiled the power meter and calculate storage requirements on the client side.

4.1 First deployment and first measurement results

Collecting power measurements is the goal of our setup in a broader scope. While this report focuses on the implementation, we briefly describe the collected data on a high level to show our experiences with the system. Our first deployment was on campus in the SWITCH[29] network and can be considered as an early production test. Evaluation of the measurement data is part of another publication - nevertheless, the data shows that the devices present different, interesting characteristics.

4.1.1 Empirical results

We measured a Cisco 8201-32FH, N540X-8Z16G-SYS-A, and NCS 55A1-48Q6H with two Autopower devices per device under test. Figure 4.1 shows the measurement setup at a router at SWITCH with two Autopower devices - one per PSU. All devices show that the drawn power is not equally distributed per PSU (e.g., see the difference of $\approx$ 50W in power draw of the red vs. green measurement in Figure 4.2, which is a graph from the Plotter). The measurement IDs in the plots correspond to the servers' internal measurement ID, which in turn corresponds to the measurement performed by the Autopower device at a PSU. The summed measurement is the sum of both PSUs. Note that the power draw (in W) differs between devices. Notably, the N540X-8Z16G-SYS-A (Figure 4.5) draws an order of magnitude less power than both other devices (Figure 4.3, Figure 4.4).

Some devices show weekly patterns. The weekly plots bin samples by day of the week.

The N540X-8Z16G-SYS-A (Figure 4.5) and 8201-32FH (Figure 4.3) show repeated peaks of power draw at midday on weekdays, while the NCS 55A1-48Q6H (Figure 4.4) doesn't show these peaks. This could imply different applications of those devices. Probably, the N540X-8Z16G-SYS-A and

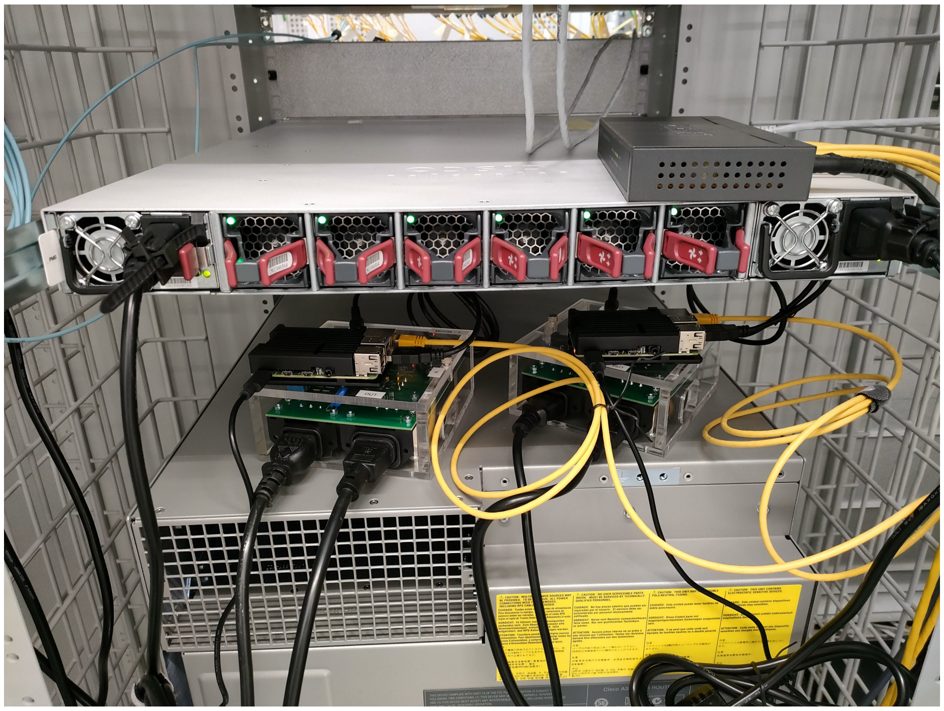


Figure 4.1: Autopower deployment at SWITCH

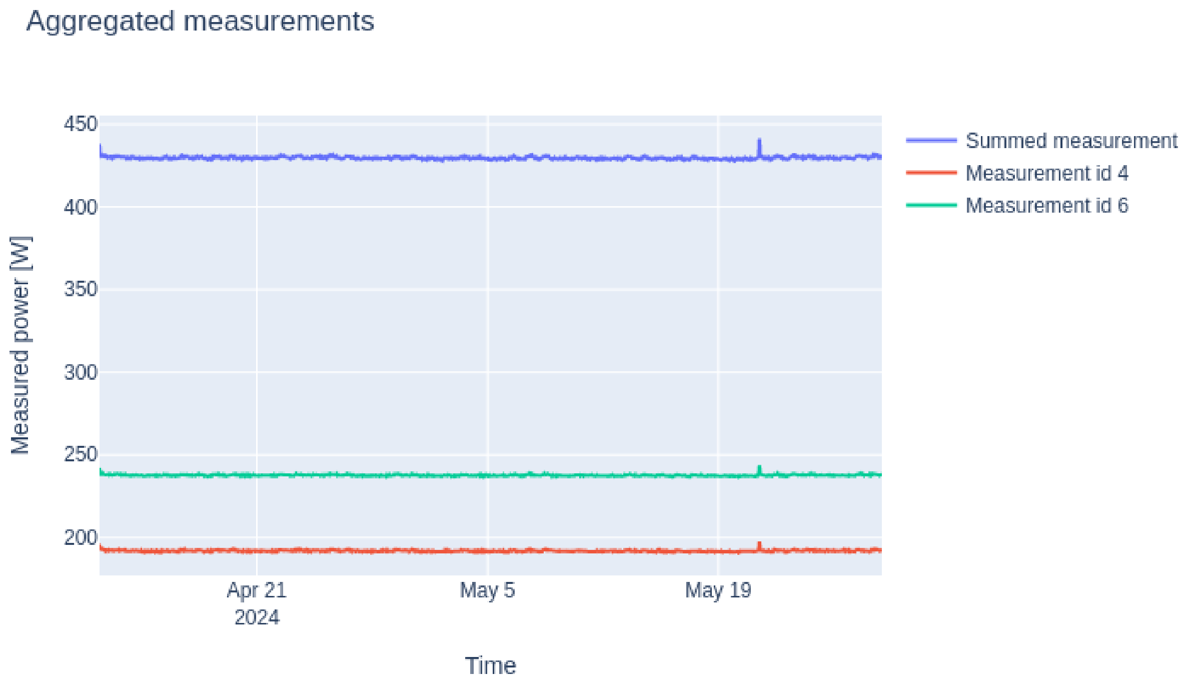


Figure 4.2: Cisco 8201-32FH Time series plot

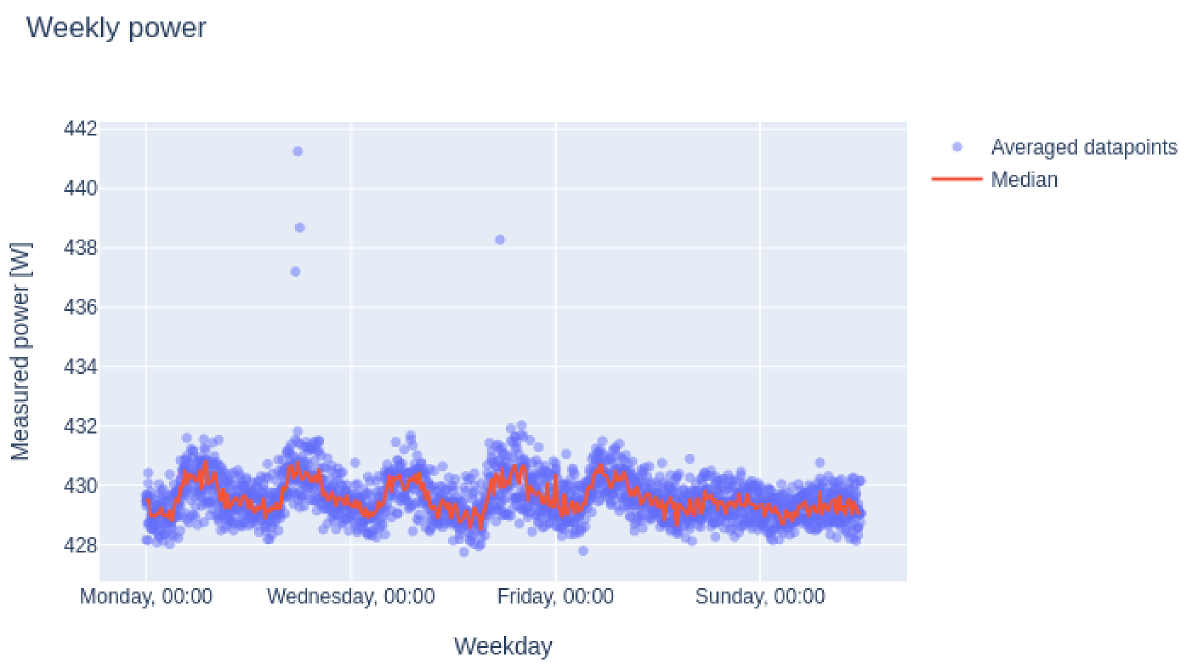


Figure 4.3: Cisco 8201-32FH Weekly plot

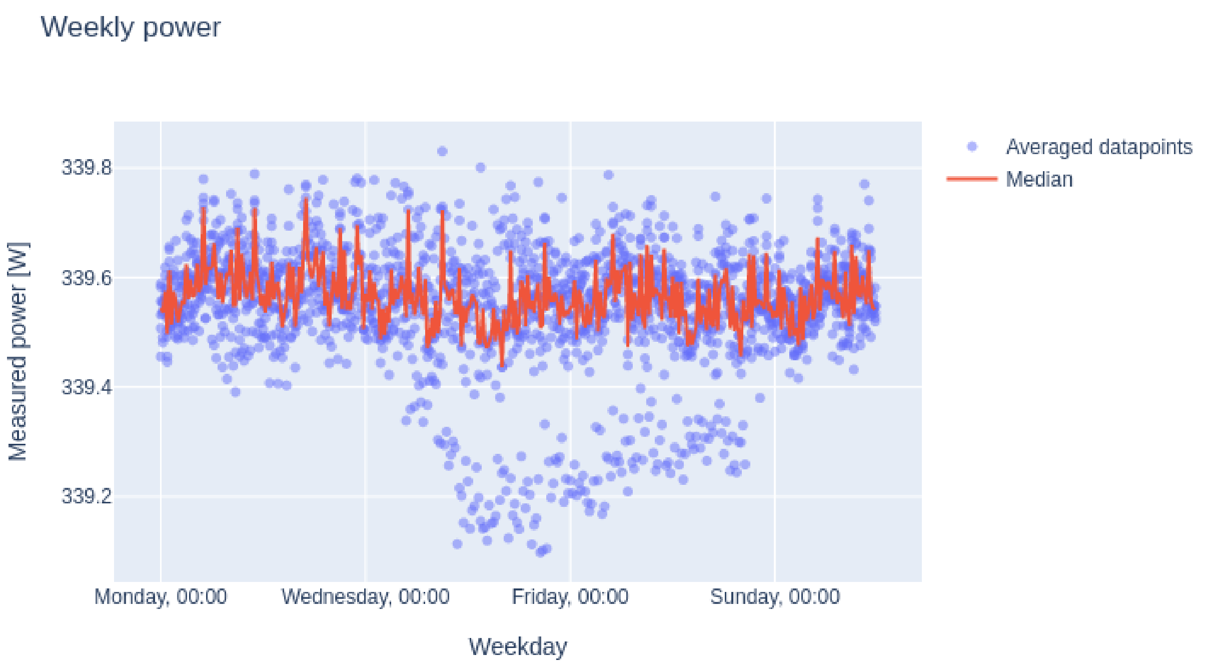


Figure 4.4: Cisco NCS 55A1-48Q6H Weekly plot

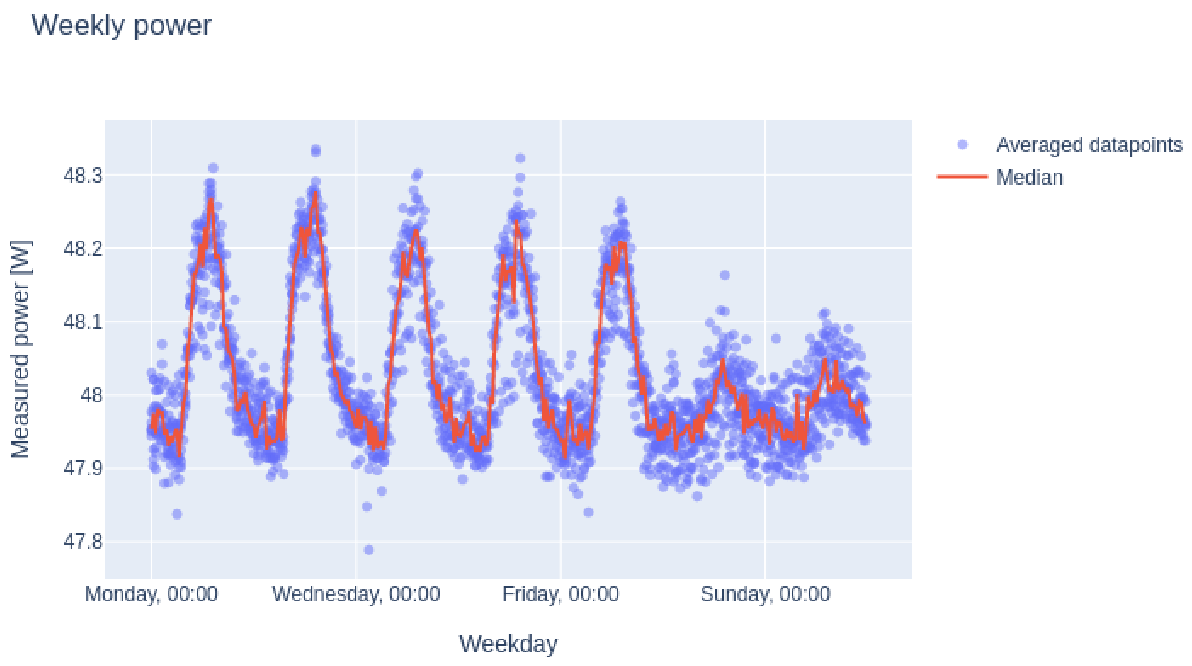


Figure 4.5: Cisco N540X-8Z16G-SYS-A Weekly plot

8201-32FH experience higher load during workdays. Detailed analysis such as comparison with traffic data remains for further work.

4.1.2 Takeaways from the first deployment

Deploying the Autopower devices was easy since we could plug in the power meters, and they could perform measurements right away. However, getting internet connectivity for data upload was a bigger hurdle: the ISP wanted to configure a subnet for our devices which took some time. Moreover, finding available RJ45 ports was challenging since not many were available. Therefore, it is beneficial to bring multiple switches for deployment and ask beforehand how the network is configured at the deployment location.

4.2 Evaluation of requirements

4.2.1 Reliability

Even though the software should still be considered as a prototype, it turned out to be mostly reliable. Most of the time, data came in as expected or could be recovered.

However, we experienced two recoverable and one unrecoverable data losses. Due to a bug with invalid SQL on the server and before implementation of acknowledgments, measurements were not saved to the server database, while the client assumed they were and thus marked them as being uploaded. Since the client stores the samples locally, we were able to reset the `was_uploaded` flag on the missing data points to trigger a re-upload. The data points came in later on as designed -

making this error recoverable.

Further, a network outrage during the first public deployment led to one-and-a-half day internet connectivity loss for some clients. After connectivity was re-established, the measurements were uploaded automatically. We had assumed that network outages of multiple days could occur during the design process. Thus - as expected - automatic recovery took place and shows that data will be uploaded eventually for shorter-term outrages. Zabbix helped us pin down the time at night when connectivity was lost and helped us share the outrage information with the ISP.

At the end of the first deployment, one device stopped uploading data even though the client remained responsive, but we did not find any non-uploaded data on the client. The log files on the client and server did not show any errors, and after a re-start of the measurement, the device started measuring again. We could not reliably pin down the issue in this case but assume that it could be related to a power outage of the power meter but not the Pi, which stopped PinPoint from outputting new data points. The Pi wasn't connected to the same power source as the DUT, and the ISP notified us about issues related to power. As an experiment, we unplugged and later on re-plugged the power meter and observed that PinPoint no longer continues measuring after a power outrage of the power meter. This makes the power outrage hypothesis plausible.

The Management UI now shows if an Autopower device is in measuring mode, but no data has been incoming for this device for more than one hour, and the client now tries to restart PinPoint automatically if no samples were received for $10 + \text{samplingInterval[s]}$ seconds. For further work, Zabbix could also monitor the server database and issue a warning if no data has been incoming for a specified time period.

In general, despite losing one week of data from one device, we could withstand a real-world network outrage, which we consider a success. Monitoring via Zabbix provided insights, and the client issues status messages in case of obvious failures to the server-side DB, both of which give important data for debugging.

In event of system failure, Zabbix notifications warn us via Slack which we were able to test at the end of our deployment by shutting down the devices. Currently, database backups of the server happen manually to a network-attached storage which is not ideal since it's not automated, but this could be implemented via a cronjob doing a database dump periodically. The Pies also serve as off-site backup, thus complete data loss would only happen if both the server and the client data become lost, something we consider as being unlikely. Despite the data loss of one device, we consider the reliability goal to be mostly fulfilled. However, improvements - especially on monitoring - are still necessary.

4.2.2 Security

Since gRPC provides encryption, transmitting measurements and commands to the clients are encrypted. Using certificates ensures that the server only accepts keys signed by a CA trusted from the server, thus only authorized clients can connect. We consider the use of certificates higher than baseline security, which would have only been symmetric encryption (e.g., with passwords).

Functions for management additionally use password-based authentication on top of the certificate-based encryption provided by gRPC. The server side stores the hashed password. We consider this secure enough, but further work could come up with more sophisticated security measures (e.g., signing each management message and then using the clients to verify that the message issuer indeed has permission to send a command).

Restricting monitoring access only to the IP of the Zabbix server and using a pre-shared key on the Autopower devices means an attacker would need to spoof the IP of the Zabbix server and guess the PSK. Considering an attacker would then only get monitoring access, the gain for the attacker is low.

More importantly, SSH access is restricted to key based login only. We consider these state-of-the-art best practices. For further protection, one could implement port knocking to only open the SSH port if a special sequence of ports gets data.

Due to the fact that the client-side application must execute PinPoint manually and we do not have an API, security concerns come up as soon as we need to pass user-defined parameters from the network to the application, even if we validate them. We perform validity checks on the parameters for PinPoint coming from the network and implement a whitelist on which counters/devices can be passed to PinPoint.

We assume that our validation is adequate, however for further work, this could be formally verified. Generally, we consider the application is secure enough for its purpose despite the possible improvements.

4.2.3 Ease of deployment

One of the main goals is easy deployment. After the device has been set up initially - which is a one-time process per Autopower device - the deployment location needs to provide internet access for full functionality of the application. Choosing gRPC brings the benefit that basic functionality does not require a permissive firewall, so if outgoing internet access is available and the device is already set up, deployment consists of plugging in the power meter, network, and Raspberry Pi. This could even be done by a user who does not know the internals of the device, and the network configuration (like firewalls) does not play a big role. This is a considerable step for easier data collection compared to needing to configure SNMP.

We could still improve the initial setup: Flashing multiple SD cards in parallel and automatic signing of the client certs would speed up the initial setup. Nevertheless, since this is a one-time process, we considered this low priority. Currently, if the IP address of an Autopower device changes - e.g., if the device is deployed in a different network, we need to re-configure the Zabbix server with the new IP of the Autopower device to be monitored, which is not ideal. Further research on how to facilitate this remains for further work.

To sum up, the initial setup of an Autopower device needs some time while the actual deployment is fairly easy. We consider this goal as fulfilled, but further work is needed on speeding up the initial setup and automating the Zabbix setup.

4.2.4 Scalability and performance

Performance and scalability were not core goals of the project. The Plotter - especially when loading a lot of data does not perform well.

Loading times of larger time frames are noticeable: Plotting about one month's worth of data points binned in 30-minute intervals needs around 23 seconds from clicking ”Build graphs” to the graphs showing up. The bottleneck is retrieving the data from the database. We see with htop that PostgreSQL on the server requires all four cores with almost 100% load while building the graphs. We build graphs in parallel using the Python ThreadPoolExecutor. Before that we were building

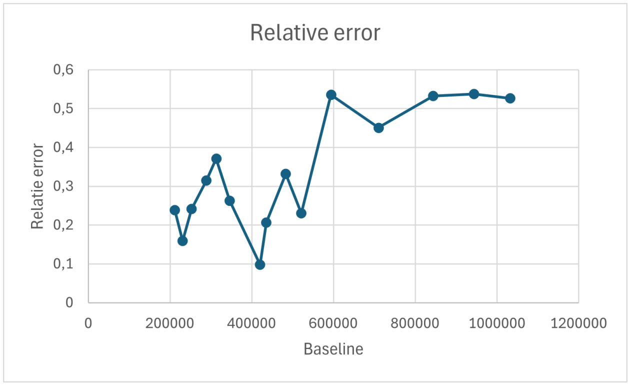


Figure 4.6: Relative error compared to baseline

the graphs sequentially, which needed around 26 seconds, meaning the benefit or concurrency is low on our machine or requires a major rewrite of the Plotter. Processing many data points is a non-trivial task, thus we do not believe that hosting the application for a larger audience would scale well - especially with our setup where the virtual machine only has four cores and 8 GB RAM. As PostgreSQL can be accessed through the network, putting the database and the Plotter on separate servers may lead to a more responsive application. However, Plotly also renders client side - thus the web browser could also be a bottleneck.

Having a Python-based server may not scale for hundreds of Autopower devices. Porting the server to C++ could yield a speedup. However, gRPC is advertised as being "high performance" [7]. It remains to see to which extent scaling up hurts performance.

4.3 Profiling the power meter

To test the accuracy of the MCP39F511N [5] we use as power meter, we compared its measurement outputs to a LMG670 by ZES Zimmer [30] which we consider our baseline. The LMG670 has an "accuracy of 0.015% of measured value + 0.01% of range" according to its product description [30]. To simulate the load, we used variable resistors with which we increased resistance and calculated the relative error by manually comparing the outputs of both power meters:

$$\text{avg. measurement of MCP39F511N device} - \text{baseline}/\text{baseline} \cdot 100 \tag{4.1}$$

The baseline was read from the display of the LMG670 and rounded. We only wanted to have a rough understanding of the accuracy of the MCP39F511N. The relative error was at worst close to 0.5%, which is within what the datasheet [31] suggests. We believe that an accuracy of 0.5% is accurate enough for our data collection: For a routers' PSU drawing 200W, this would yield about 1W error.

4.4 Storage requirements

Every Raspberry Pi has a 16 GB or 32 GB SD card, and we assume that data could be stored for over a year. We calculated storage requirements based on the server and simulated client data and used the 16 GB SD card as the baseline. The server saves data points from all clients, which gives us more data points and uses an index for performance, while the client-side schema has the `was_uploaded` boolean attribute. We see the storage requirements of the server database as upper bound since the index takes more space than an additional boolean attribute (the boolean takes one byte[32] per sample).

We ran `SELECT pg_total_relation_size('measurement_data');` to get the storage needs.

Storage requirements are $\frac{4.8\text{GB}}{48\text{Mio samples}} \approx 100\text{Byte/sample}$ meaning that with the default sampling interval of 500ms one day's worth of measurements takes $\approx$ 20MB. We derive this number as follows:

- With $\approx$ 48 Mio. samples in the `measurement_data` relation on the server, the total relation size of `measurement_data` is $\approx$ 4.8GB.
- The client side DB requires a bit less on simulated data via measuring the development machines CPU: $\approx$ 1.5 Mio. samples yield $\approx$ 100MB, meaning $\frac{100\text{MB}}{1.5\text{Mio samples}} \approx 70\text{Bytes/sample}$.
- For simplicity, we round the client-side storage needs up to $\approx$ 100Bytes/sample.
- By default, the device samples power every 500ms, which means that every second, the database grows by roughly 200 Bytes.
- Thus, we have $200\text{MB/s} * 3600\text{s/h} * 24\text{h/day} \approx 16.5\text{MB/day} \approx 20\text{MB/day}$.

Assuming that the OS takes 5 GB (which is an over-approximation since the uncompressed 2024-03-15-raspios-bookworm-arm64-lite[33] image takes 2.8 GB), we can calculate the time frame until the SD card is expected to be full:

- Assuming a 16 GB SD card, we have $16 - 5 = 11\text{GB}$ space for measurements
- This allows us to store $\frac{11\text{GB}}{20\text{MB/day}} \approx \frac{11260\text{MB}}{20\text{MB/day}} \approx 560\text{day} \approx 1.5\text{years}$ worth of data on the SD card.

Hence, even larger outrages or total offline measurements would not be bound by the size of the SD card. During the one-and-a-half day outrage, the monitoring showed high SWAP usage. We assume that fetching data from the database was not memory efficient. We no longer load all samples to RAM before uploading, but it remains to be tested if, after this change, the SWAP usage comes up again.

4.5 Overall project evaluation summary

In general, given the time constraints, the project went well. While the initial goal was measuring and transmitting data to an external source with some basic management, we were able to expand the project to have basic plots and an online UI instead of just a CLI-based application for management. Moreover, we were even able to collect one month's worth of measurements, with more measurements to come in the future, set up monitoring, and document deployment.

Chapter 5

Outlook and summary

Despite being able to collect and visualize data, controlling measurements remotely and monitoring, the project has many areas for further improvement.

5.1 Most important next steps

Most importantly, we should improve reliability by setting up more monitoring. Zabbix currently monitors the Autopower devices on an operating system level, but we haven't succeeded yet in monitoring the server database and getting notified in case of data loss or data not being incoming for a long time. Further, we could implement an automated server-side database backup to a separate, secure location.

Since we collected data for three devices successfully, we should next scale up. We already ordered more Raspberry Pies and power meters with the plan to deploy these devices in the future.

Scaling up raises the question of whether Python scales well enough or if we should rewrite the server in another language, such as C++.

The presence of our collected data is the foundation of further analysis. Combining this data with SNMP data could serve as validation data, aid experiments on out-of-the-lab devices, and allow us to publish data.

5.2 Quality of life improvements

The Plotter and Management UI would both need some thoughts on how to improve user experience: The Plotter does not yet group by the notion of runs and only uses measurements for now. Moreover, we should improve the performance of plot generation. For the client side, it is possible to switch to WebGL for GPU-accelerated scatter plots [34]. For the server side, the plotting script would benefit from more investigation of performance improvements, like rewriting SQL queries and checking for other ways to introduce concurrency. The Management UI should use more AJAX to provide a more responsive UI and use more interactive windows instead of new pages for database edits.

The initial setup of an Autopower device needs manual work. Currently, flashing the operating system, running the deployment script and signing the certificate needs some time per device. In future, we could speed up the process with multiple card readers and automate the signing process.

To manage many Autopower devices, we could move away from the manual setup via deployment script and towards a software like Ansible [35].

On the software development and design side, various smaller and larger tasks are still open. While we build the client via GitHub actions for ARM, this currently takes more than four hours since gRPC is built from source via nested virtualization. This is not efficient and has room for improvement. Once GitHub introduces native ARM runners in future[36], we should switch to building directly on the runner. Building the client for production does not happen often, and local development and debugging is possible and preferred.

Besides just building the client, CI could also run unit tests. Up to now We relied on manual testing, but unit tests for client and server should be part of further work.

On the software design side, we could write a library to control the power meter instead of using PinPoint and switch to a more event-based programming model instead of manually using threads. Both design changes mean a larger rewrite of the software. In the long term, more features like scheduling measurements, rebooting the Autopower clients via the application instead of SSH, and uploading the measured data automatically to some public database would be nice-to-haves.

All those points show that there is still work to do in future with monitoring being the most pressing - even though the system already has its basic functionality.

5.3 Summary

To sum up, we developed the Autopower system to provide reliable and easy-to-deploy devices for automated power measurement with data collection, visualization, device management, and monitoring capabilities. As a result, we collected over one month's worth of data from three network devices on campus. Main challenges are network access and initial setup. However, we believe that this system should be deployed on a larger scale, and its data serves as the foundation for further work in sustainable networking research.

Bibliography

[1] "RIPE Atlas - RIPE Network Coordination Centre." [Online]. Available: https://atlas.ripe.net/

[2] M. Chandramouli, B. Claise, B. Schoening, J. Quittek, and T. Dietz, "Monitoring and Control MIB for Power and Energy," Internet Engineering Task Force, Request for Comments RFC 7460, Mar. 2015, num Pages: 69. [Online]. Available: https://datatracker.ietf.org/doc/rfc7460

[3] R. Presuhn, "Management Information Base (MIB) for the Simple Network Management Protocol (SNMP)," Internet Engineering Task Force, Request for Comments RFC 3418, Dec. 2002, num Pages: 26. [Online]. Available: https://datatracker.ietf.org/doc/rfc3418

[4] "RIPE Atlas - RIPE Network Coordination Centre." [Online]. Available: https://atlas.ripe.net/campaigns/c04a682c-0809-4bb1-8852-49fc947e6e2e/

[5] "ADM00706." [Online]. Available: https://www.microchip.com/en-us/development-tool/adm00706

[6] S. Köhler, B. Herzog, T. Hönig, L. Wenzel, M. Plauth, J. Nolte, A. Polze, and W. Schröder-Preikschat, "Pinpoint the Joules: Unifying Runtime-Support for Energy Measurements on Heterogeneous Systems," in *2020 IEEE/ACM International Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*, Nov. 2020, pp. 31–40. [Online]. Available: https://ieeexplore.ieee.org/document/9307947

[7] "gRPC," publication Title: gRPC. [Online]. Available: https://grpc.io/

[8] M. Arlitt, M. Karamollahi, and C. Williamson, "A Retrospective on Campus Network Traffic Monitoring," *SIGCOMM Comput. Commun. Rev.*, vol. 53, no. 2, pp. 40–45, Jul. 2023. [Online]. Available: https://doi.org/10.1145/3610381.3610387

[9] "osmhpi/pinpoint," Jun. 2024. [Online]. Available: https://github.com/osmhpi/pinpoint

[10] Boavizta, "Tools — Boavizta." [Online]. Available: https://boavizta.org/en/tools

[11] "Boavizta/Energizta," Jul. 2024. [Online]. Available: https://github.com/Boavizta/Energizta

[12] P. G. D. Group, "PostgreSQL," Jul. 2024, publication Title: PostgreSQL. [Online]. Available: https://www.postgresql.org/

[13] "Zabbix :: The Enterprise-Class Open Source Network Monitoring Solution." [Online]. Available: https://www.zabbix.com/index

[14] "autopower/client/deploy.sh at main · nsg-ethz/autopower," publication Title: GitHub. [Online]. Available: https://github.com/nsg-ethz/autopower/blob/main/client/deploy.sh

[15] “Data Apps for Production — Plotly.” [Online]. Available: https://plotly.com/

[16] “Grafana: The open observability platform.” [Online]. Available: https://grafana.com/

[17] “PostgreSQL data source — Grafana documentation.” [Online]. Available: https://grafana.com/docs/grafana/latest/datasources/postgres/

[18] “3.4. Transactions,” May 2024, publication Title: PostgreSQL Documentation. [Online]. Available: https://www.postgresql.org/docs/16/tutorial-transactions.html

[19] “MySQL.” [Online]. Available: https://www.mysql.com/

[20] “autopower/client/client_db_schema.sql at main · nsg-ethz/autopower · GitHub.” [Online]. Available: https://github.com/nsg-ethz/autopower/blob/main/client/client_db_schema.sql

[21] “11.2. Index Types,” May 2024, publication Title: PostgreSQL Documentation. [Online]. Available: https://www.postgresql.org/docs/16/indexes-types.html

[22] “autopower/proto/api.proto at main · nsg-ethz/autopower,” publication Title: GitHub. [Online]. Available: https://github.com/nsg-ethz/autopower/blob/main/proto/api.proto

[23] “Asynchronous-API tutorial,” publication Title: gRPC. [Online]. Available: https://grpc.io/docs/languages/cpp/async/

[24] “Add flag to skip execution of workload - osmhpi/pinpoint,” Jun. 2024. [Online]. Available: https://github.com/osmhpi/pinpoint/pull/15

[25] “autopower/client/README.md at main · nsg-ethz/autopower · GitHub.” [Online]. Available: https://github.com/nsg-ethz/autopower/blob/main/client/README.md#deployment

[26] “Design — Gunicorn 22.0.0 documentation.” [Online]. Available: https://docs.gunicorn.org/en/latest/design.html

[27] “gevent compatibility · Issue #4629 · grpc/grpc,” publication Title: GitHub. [Online]. Available: https://github.com/grpc/grpc/issues/4629

[28] “psycopg/psycogreen,” Apr. 2024. [Online]. Available: https://github.com/psycopg/psycogreen

[29] “Switch.” [Online]. Available: https://www.switch.ch/en

[30] “LMG670 - Channel Power Analyzer - ZES ZIMMER.” [Online]. Available: https://www.zes.com/en/Products/Predecessor-Products/Energy-and-Power-Meters/LMG670

[31] “MCP39F511N Datasheet.” [Online]. Available: https://ww1.microchip.com/downloads/aemDocuments/documents/OTH/ProductDocuments/DataSheets/20005473B.pdf

[32] “8.6. Boolean Type,” May 2024, publication Title: PostgreSQL Documentation. [Online]. Available: https://www.postgresql.org/docs/16/datatype-boolean.html

[33] “Raspberry Pi OS 2024-03-15 download.” [Online]. Available: https://downloads.raspberrypi.com/raspios_lite_arm64/images/raspios_lite_arm64-2024-03-15/2024-03-15-raspios-bookworm-arm64-lite.img.xz

[34] “Webgl.” [Online]. Available: https://plotly.com/python/webgl-vs-svg/

[35] “Ansible Collaborative.” [Online]. Available: https://www.ansible.com/

[36] "Accelerate your CI/CD with Arm-based hosted runners in GitHub Actions," Oct. 2023. [Online]. Available: https://github.blog/changelog/2023-10-30-accelerate-your-ci-cd-with-arm-based-hosted-runners-in-github-actions/