Measuring, Visualizing, and Optimizing the Energy Consumption of Computer Clusters

Bachelor's thesis

Faculty of Computer Science and Mathematics
at the University of Passau

Nils Steinger
2017-06-16

supervised by
Prof. Dr. Dirk Beyer

Contents

Abstract

Evaluating the effectiveness and efficiency of computer programs, for example software-verification tools, generally requires large amounts of computational resources, to an extent that is most economically provided by setting up computer clusters consisting of several individual nodes. These individual computer nodes often consume large amounts of electrical energy, both during useful operation and while in idle state.

This thesis will examine and compare available technologies for measuring this energy consumption, collecting the measurement data on a central system for later use, and generating visual representations of the measurements to provide users with an insight into the current energy consumption, as well as assist in planning possible improvements.

One such improvement — automatically powering down unused nodes, and starting them on demand — will be implemented as an extension to the `VerifierCloud` task-queueing framework. The potential benefits of this improvement will then be evaluated using data from the measurement instrumentation set up in the context of this thesis.

1. Introduction and Outline

The Chair for Software Systems at the University of Passau [1] focuses its research on methods for formal software verification, using methods such as software model checking and static analysis. [2] One of the main research aspects is to improve the performance of these software verification tools, both in terms of result accuracy and of resource consumption (usage of CPU time, memory consumption, etc.).

The necessary software infrastructure for running automated benchmarks on a computer cluster and collecting performance data is already in place. However, until now one major factor — impacting both cost of operation and environmental impact — has been neglected: the energy consumption of the computer cluster while performing verification runs and also during idle times when no verification runs are executed.

With an energy consumption of more than 31 000 kWh per year (equating to a cost of more than 5 900 € at current prices) from our two main computer clusters alone, this area warrants analysis, as well as exploration of possible improvements to reduce the energy consumed on a daily basis. On a global scale, the issue is even more pronounced: as of 2012, 4.6 % of the worldwide annual energy consumption of TWh were caused by information and communication technology, 29 % of which were due to data centers [13]. This thesis aims to provide an overview of the steps required to analyze the energy consumption of computer clusters, and to help research institutions optimize their computer infrastructure. This will help save on the cost of operation, as well as reduce negative consequences for the global environment, considering 66.7 % of global electrical energy production are still derived from fossil fuels [16].

As the first step, chapter 2 documents the instrumentation that has been set up to measure the energy consumption of the most heavily used parts of the chair's computer cluster, both in idle state and during software-verification runs. These methods should be applicable with little change to any environment where the energy consumption of a small to medium number of computers is of interest to the researcher.

Once the various options for data acquisition have been explored, a number of readily available software packages for storing and visualizing that data is listed and compared in chapter 3 and a suitable selection is made to accommodate our use case.

Finally, chapter 4 presents the implementation of a system for automatically powering off and starting "worker" nodes depending on current resource demand, as well as tests and calculations of the potential savings on energy consumption that can be achieved using this new feature. The power-off and power-on mechanisms have been integrated

[1] As of August 2016, Prof. Dirk Beyer has moved to LMU Munich as head of the Chair for Software and Computational Systems.

[2] Software and Computational Systems Lab – Research, `https://www.sosy-lab.org/research.php`

into the `VerifierCloud`, the existing task-queueing solution developed at and used by our chair, whose characteristics will be outlined in section 4.1.

Systems Overview

While the `VerifierCloud` utilizes a diverse set of computers in our usage scenario, the development and testing in this thesis will focus on the two sets of machines that proved most suitable for being equipped with measurement instrumentation:

- Computer cluster "cayman"
  - 8 nodes (tower form factor)
  - CPU model: Intel Core i7-4770 @ 3.40 GHz
  - 32 GB RAM per node
  - Power consumption measured via single-phase energy meters (Eltako WSZ12DE-32A; see section 2.3)
- Computer cluster "zeus"
  - 24 nodes (rack-mounted as six sets of four blade servers)
  - CPU model: Intel Xeon E5-2650 v2 @ 2.60 GHz
  - 128 GB RAM per node
  - Power consumption measured via rack power distribution unit (APC AP8681; see section 2.2)

These two clusters are operated directly by the Chair for Software Systems and are dedicated to the purpose of testing software-verification tools. This allows us to automatically power them off without impacting users or other parts of our infrastructure and to perform work on their electricity-supply circuitry to add the parts of our measurement instrumentation that require dedicated hardware modifications, as detailed in chapter 2.

2. Measurement Technologies

The decision to measure energy consumption in addition to purely software-based performance data was made after the initial setup of the *cayman* and *zeus* computer clusters, so the measuring equipment had to be retrofitted to match the existing cluster hardware.

Several different approaches proved viable for this, which will be described and compared in this chapter.

Naming of Physical Quantities

A note on the physical quantities referred to in this chapter: for our use case, two units are of particular interest:

- electric **power** consumption, which is a changing, momentary value ($[\mathrm{V} \cdot \mathrm{A}] = [\mathrm{W}]$), and
- electric **energy** consumption, which is the integral of power over time ($[\mathrm{W} \cdot \mathrm{s}] = [\mathrm{J}]$)

These two terms are often used interchangeably in colloquial language, and even in some scientific sources [12]. This thesis makes an effort to avoid this inaccuracy, and as such, "power" will refer to momentary power consumption, while "energy" will refer to an accumulated usage value over a period of time. Independently, however, the terms "consumption" and "usage" will be treated as synonyms.

Note also that the measurement technologies described in this chapter will usually only yield measurements of one of the two quantities. Since the time interval between measurements is known, these can, however, easily be converted to the other quantity by means of differentiation, or integration, respectively.

2.1. Intel Running Average Power Limit (RAPL)

Since the release of their 32 nm processor microarchitecture (codenamed "Sandy Bridge") in 2011, Intel CPUs marketed under the "Core" and "Xeon" brands include an interface to an internal energy-consumption counter named Running Average Power Limit (RAPL). [1] This feature also provides control features in addition to mere reporting, but since our use case is to measure the CPU's energy consumption without artificially slowing it down, reading the counters will suffice.

[1] Running Average Power Limit – RAPL, 01.org Intel Open Source, `https://01.org/blogs/2014/running-average-power-limit-%E2%80%93-rapl`

Integrated directly into the CPU, RAPL has several advantages: it does not require any additional hardware or modifications to the system, which makes it cheaper and easier to set up than the other measurement methods described in this chapter. It also provides measurement values at a resolution unparalleled by any of the other methods (usually on the order of tens of millijoules [5]). Note, however, that the counter value is only updated roughly every millisecond [12].

On the downside, Intel does not guarantee any specific accuracy for the measured values — in fact, the feature is not even guaranteed to be present in future releases of the "Core" and "Xeon" CPU series at all (hence the use of "model-specific registers" as described below). There have, however, been several empirical studies comparing RAPL measurement results to data obtained from external meters, whose results give reason to assume that RAPL yields accurate results [12, 3].

The energy consumption is reported separately for different parts of the CPU and its associated systems, termed "domains" in this context: [1]

- for the entire CPU socket (`package` domain),
- in sum for all CPU cores (`core`, also called `pp0`, i.e. "power plane 0"),
- for additional features like integrated graphics, if present (`uncore`, also called `pp1`, i.e. "power plane 1"), and
- for memory that is local to the CPU package (`dram`).

However, not all domains are supported on all CPU models, so it is important to check which ones are available before trying to obtain measurements from them.

Access to the RAPL counters is implemented via the CPU's model-specific registers (MSRs) and thus requires specialized software, both for accessing the MSRs, and for parsing their values.

The first reference implementation of this was Intel's own Intel Power Gadget [2], released in different versions for Windows, Mac, and Linux. While simplistic, the Linux implementation provided a sufficient basis for our own RAPL-based measurement utility. It relies on the `msr` Linux kernel module to access the MSRs, then performs the necessary calculations (such as multiplying the counter value by the counter's model-specific energy unit) internally.

RAPL functionality can also be accessed from within the Linux kernel itself: there is a read-only implementation [3] (introduced in 2011 and since renamed to `intel_rapl_perf`) that can be used with the `perf` profiler tool [4], as well as the newer `intel_rapl` kernel module that allows applications to both query and control the CPU's energy consumption without requiring direct access to CPU registers, simply by reading from and writing

[2] Intel Power Gadget, `https://software.intel.com/en-us/articles/intel-power-gadget-20`

[3] introduce intel_rapl driver, `https://lwn.net/Articles/444887/`

[4] Tutorial – Perf Wiki, section "Introduction", `https://perf.wiki.kernel.org/index.php/Tutorial#Introduction`

to files under the virtual filesystem path `/sys/class/powercap/intel-rapl/` [5]. Our implementation, however, still uses the direct MSR interface demonstrated by Intel Power Gadget, rather than these version-dependent kernel interfaces.

After some manual testing to ensure RAPL support on our hardware, we modified the Intel Power Gadget source code to include proper error handling, and added an interface to an SNMP daemon. Originally, the Power Gadget would silently ignore any errors reported by its RAPL query functions, potentially leading to incorrect values in its output. This was, however, easily fixed, and the resulting code can be found in appendix A. The program was also modified to conform to the output format required by the `Net-SNMP` SNMP agent's `pass_persist` interface, making it possible to retrieve energy measurements via SNMP (see section 3.2.1 for details on the use of SNMP).

With this software in place, both of our main computer clusters were now equipped with an integrated (and thus cheap and comparatively simple) means of remotely querying and collecting energy measurements.

Keep in mind, however, that RAPL only measures the energy consumption of the CPU. Other electric loads, such as disk drives and cooling systems, are not included in this figure. To achieve a complete view of the computer system's energy consumption, we will require additional measurement instrumentation, as described in the next sections.

2.2. Rack Power Distribution Unit APC AP8681

When the *zeus* computer cluster was commissioned, the planned setup also included a switched rack power distribution unit (PDU), offering several technical benefits to the final rack-mounted computer cluster.

The model AP8681 chosen for our setup is operated from a three-phase electric supply and provides 24 power outlets, eight supplied from each input phase. Each outlet can be switched on and off individually by the PDU's control circuit, either automatically or on user request via a web-based interface. This allows for automated features such as staggered power-on (switching on a large computer cluster one-by-one with delays in between, to avoid overloading the electric circuit with the collective inrush current of the entire cluster).

More importantly for our use case here, the PDU continually measures the power consumed through each outlet. This is used internally as a possible means of failure detection, i.e. alerting an operator when an appliance suddenly draws significantly more or significantly less power than it normally does. Additionally, the measurements can be queried via SNMP, and thus collected by external monitoring systems such as those described in chapter 3.

This makes the AP8681 a convenient tool to monitor the energy consumption of a rack-mounted computer cluster. There are, however, some limitations that need to be taken into account: Most important for our use case are the limits of the PDU's measurement range and accuracy: APC specifies a measurement accuracy of $\pm 3\,\%$ for

[5] introduce intel_rapl driver, `https://lwn.net/Articles/444887/`

their AP8XXX-series PDUs [6], but only when the measured current exceeds 0.5 A. Below this threshold, measurement accuracy is explicitly undefined, and values below 0.3 A are even discarded and reported as zero instead. At the PDU's rated nominal input voltage of 230 V, this corresponds to a power consumption of up to 69 W that will be reported as 0 W, and of up to 115 W that is not guaranteed to be measured accurately. To give an example of where this can be a hindrance, the power consumption of our *cayman* nodes in idle state can be as low as 19 W each. Using only the PDU's integrated measurement capabilities, we would therefore in some cases be unable to tell if a certain PDU outlet has a node connected to it at all. Because of this, care must be taken when using one of these PDUs for measuring purposes, to ensure the results will be reliable and accurate.

Another drawback of using this type of advanced PDU is their high price — the model AP8681 used in our setup is currently quoted at over 2 080 € by APC, with models for lighter electric loads ranging between approximately 1 400 € and 1 600 € [7] — compared to the other devices presented in this chapter.

The PDUs may also require special tooling to install, since they are purpose-built to be mounted in a rack, and (at least in the case of AP8XXX-series PDUs) use the IEC 60309 industrial power plug types [20], rather than the IEC 60083 domestic types [19] more commonly found on commodity computer hardware.

For these reasons, we explore other options for measurement hardware in the following sections.

2.3. Single-phase Energy Meter Eltako WSZ12DE-32A

We examined one possible device for individual-outlet metering in section 2.2, but concluded that it had some significant disadvantages in terms of accuracy and cost. We therefore searched for an alternative that would ideally improve upon both points.

One such option are DIN-rail single-phase energy meters that are designed to be mounted inside a fuse box and measure the instantaneous power and accumulated energy consumption of the load or loads connected to a single electrical phase. Some models are equipped with an electrical output contact that emits signal pulses at a frequency proportional to the power consumption currently detected by the meter, making it possible to automatically collect their measurement data by counting those pulses.

For our setup, we settled for the Eltako model WSZ12DE-32A meter, because it was readily available, comparatively cheap (between 35 € and 40 € at the time of writing), and provides a higher measurement resolution than its competitors (as explained in

[6] APC Technical FAQs: How accurate is the current monitor on an APC Rack PDU (AP7XXX or AP8XXX)?, `http://www.apc.com/us/en/faqs/FA156074/`

[7] Rack-PDU mit Strommessfunktion für jeden Ausgang (Metered-by-outlet Rack Power Distribution Units), `http://www.apc.com/shop/de/de/categories/_/N-ooil4d`

further detail below). [8]

This type of DIN-rail energy meter commonly features a built-in digital display that shows the accumulated energy usage over the lifetime of the meter. For our purpose of automatically collecting measurements from the meter, we do, however, require an additional output called an S0 interface [9].

The S0 interface specification [9] describes a two-wire electrical interface where one device (in this case the energy meter) transmits binary pulses by modulating the current allowed to flow on the interface circuit — a signaling scheme called a "current loop": A binary one is signaled by passing current (a minimum of 10 mA for Class A long-distance transmission, or 2 mA for Class B short-distance transmission) on the circuit. A binary zero is represented by a smaller current ("live zero" or "elevated zero", with at most 2 mA for Class A, and at most 0.15 mA for Class B), rather than no current at all, so the receiving device can detect whether the circuit is intact.

This current-loop signaling has the advantage of being robust against electromagnetic interference and voltage drop on the connecting cable, and can also detect physical connection failures, such as broken cables. It does, however, require particular attention to the correct polarity of the signal voltage, as well as the voltage and current used. Setup mistakes, as well as faults in either of the connected devices, can lead to damage or destruction of the devices.

To simplify the hardware interface, measurement and automation devices commonly use potential-free "floating" contacts — essentially a simple electronic on/off switch — as an alternative to current loops. Our Eltako energy meters belong to this group, which greatly simplified the hardware setup required to interface with them. The lack of a "live zero" state technically eliminates the cable-fault detection found in current-loop setups, but since we will be measuring the power usage of a near-continuously running computer system, a faulty cable would quickly become obvious from the sudden drop in measured power consumption.

The model WSZ12DE-32A signals its measurements in near-realtime by closing the potential-free contact 2 000 times per kWh (corresponding to one electric pulse every 0.5 Wh or 1.8 kJ on average) whereas similar devices from other manufacturers are often limited to 1 000 pulses per kWh. These pulses can then be captured by an external device, yielding data on both the current power consumption (from the current interval between pulses), and the overall energy consumption (from the total number of pulses).

The manufacturer specifies the meters' accuracy as $\pm 1\,\%$ [10], which is significantly more accurate than the AP8681 rack PDU from section 2.2. The minimum measured current [11]

[8] The WSZ12DE-32A is sold as a meter "without approval", i.e. it lacks the official calibration required to be permitted for billing purposes (also called a "revenue-class meter"). This has no impact on the specified accuracy, however, so meters "without approval" are still suitable for academic measuring applications.

[9] Not to be confused with the "S interface" or "S_0 bus" used in ISDN environments.

[10] Single-phase Energy Meters WSZ12DE-32A without approval, `https://www.eltako.com/fileadmin/downloads/en/_datasheets/Datasheet_WSZ12DE-32A.pdf`

[11] Incorrectly labeled as "inrush current" on the manufacturer's English datasheet, but correctly termed "Anlaufstrom" (start-up current) in the German version.

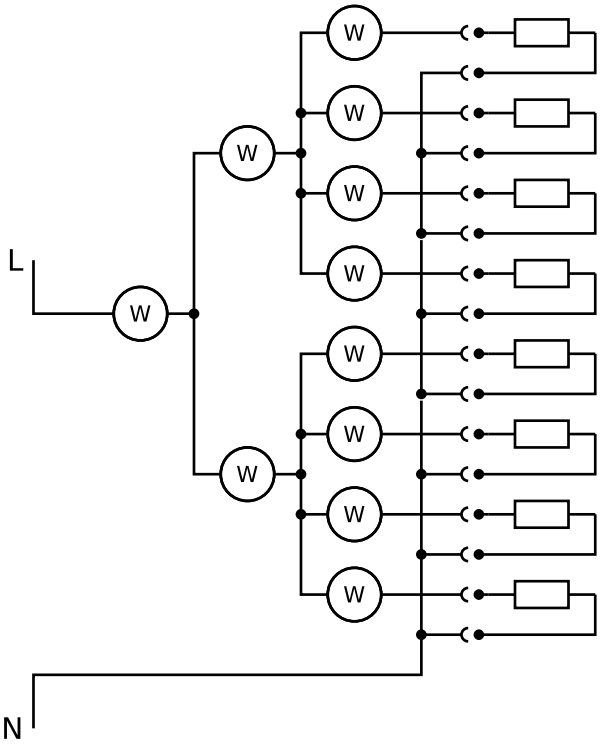

Figure 2.1.: Schematic of the tree-like configuration of our Eltako energy meters (section 2.3), shown here as the standard power meter symbol (W). As usual in an electric schematic diagram, L and N denote the live and neutral conductors of a single-phase electric mains supply, and the resistor symbols on the right symbolize a generic electric load (in our case, the *cayman* computer nodes whose power consumption is being measured).

(similar to the sub-0.3 A cut-off on the AP8681) is given as 20 mA, corresponding to as little as 4.6 W at nominal mains voltage. This makes the WSZ12DE-32A a much better fit for our low-power *cayman* nodes than the AP8681.

Our setup for monitoring the energy consumption of these eight *cayman* nodes consisted of elven identical Eltako energy meters, cascaded in a tree configuration (shown in figure 2.1) to be able to verify the measurement results across multiple meters. The necessary wiring and equipment installation was kindly performed by building-services staff of the University of Passau.

Once the mains-voltage part of the setup had been completed, we proceeded to connect the meters to a Raspberry Pi single-board computer (SBC) to process their signal output. The meters' output contacts were connected to the SBC's ground rail on one side, and individual GPIO pins on the other side, utilizing the GPIOs' internal pull-up resistors to ensure a well-defined input voltage level in the open contact state.

On the software side, the Raspberry Pi ran the default Raspbian Linux distribution, along with a custom Python script to detect and count the incoming pulses from the energy meters. The script would configure the relevant GPIO pins to generate an inter-

rupt and invoke a method whenever they detected a change in voltage (i.e. a pulse from the energy meter), and this method would then increment an internal counter for the appropriate meter. For the sake of simplicity and interoperability, these counters were stored as individual plain-text files on the local filesystem. Those files could then be used concurrently by other programs to provide different interfaces to access the data. Two front ends developed by us are of particular interest:

First, there is a web-based user interface that obtains real-time data from a websocket server on the SBC and uses it to display an estimate of the current power consumption reported by each of the eleven meters, both numerically and as a bar graph. The estimation is done based on the interval between signal pulses from each meter: for example, since one pulse signifies 0.5 Wh, one pulse every five seconds would represent a current power consumption of $\frac{0.5\,\mathrm{Wh}}{5\,\mathrm{s}} = \frac{0.1\,\mathrm{Wh}}{\mathrm{s}} = 360\,\mathrm{W}$. If the next pulse does not arrive within 5 s, however, the power consumption has to be lower than the aforementioned value, and the web front end starts decreasing the power figure accordingly. All calculations are performed in JavaScript within the user's web browser to ensure timely updates to the data display. This real-time interface has proved to be useful to quickly check the current load status of the computer cluster, but does not satisfy the requirement for long-term data collection.

For the latter, there is a separate script that again interfaces with a `Net-SNMP` agent — running on the SBC in this case — and exposes the current counter values via SNMP for consumption by the data acquisition tools described in chapter 3. Note that these are the raw counter values, so the conversion from multiples of 0.5 Wh to the required unit needs to be performed by the client querying the data. This approach was selected to avoid any inaccuracies introduced by the conversion, and also has the advantage of making the same script usable for any energy meter, including meters with a different measurement resolution.

Both tools can be found in the software list in appendix A.

2.4. Electronic Electricity Meter EMH ED300L

The WSZ12DE-32A energy meters described in section 2.3 proved to be well-suited for our use case, providing us with comparatively cheap (especially compared to the rack PDU from section 2.2) and accurate measurement instrumentation. However, external assistance was required to have them installed and connected professionally, so we decided to also investigate another possibility to facilitate replicating our setup elsewhere: recently, many German electric companies [12] have started to deploy so-called "smart electricity meters", particularly for their residential customers. These electronic electricity meters feature some useful interfaces not present in their (usually analog) predecessor models. Their increasing deployment by electric companies also makes it likely that a

[12] A cursory web search yields several electric companies providing ED300L usage instructions for their customers — Energieagentur Göttingen, E-Werk Mittelbaden, infra fürth, KWH Netz (Haag i. OB), Netzgesellschaft Forst (Lausitz), Stadtwerke Augsburg, Stadtwerke Gießen, Stadtwerke Landau a. d. Isar, and Stadtwerke Wiesbaden, to name just a few.

comparable meter is already present in the hardware infrastructure or can be installed more easily than the multi-device solution from section 2.3.

A meter model that is widely used is the electronic household electricity meter ED300L manufactured by EMH metering. Like most household electricity meters, its measurement accuracy given as an EN 50470-3 class index, ranging from A to C.[13] In our case, the ED300L is specified to comply with class A, corresponding to an accuracy of ±2.0 % when operated within its rated electric current range [10].

As a fully electronic meter, it uses a digital display as its main user interface for the basic functionality of measuring the total energy consumption over the lifetime of the unit, as well as its additional consumption counters. These include a set of counters that covers historic time periods (past 1, 7, 30, and 365 days), and multiple independent counters selectable via an external electric contact. The selectable counters are intended primarily for scenarios with different electricity tariffs (e.g. depending on the time of day) that can then be metered separately by switching to a different counter (e.g. using an electronic timer connected to the counter selector contact).

The ED300L also features an interface similar to the S0 interface described in section 2.3, in this case implemented as an infrared LED on the meter's front panel. Depending on the exact device model, the LED emits either 5 000 or 10 000 light pulses per kWh, potentially exceeding the resolution of the Eltako WSZ12DE-32A's S0 interface by an order of magnitude. However, the manufacturer recommends that this light pulse output should only be used for testing purposes, and does not provide an accuracy specification for it.

Instead, the primary machine-to-machine interfaces (referred to as "INFO interface" by EMH) is another infrared LED that is used to relay more detailed information from the meter. The INFO interface periodically emits data packets using a unidirectional serial protocol similar to that commonly used on RS-232 serial interfaces. These packets contain a variety of information about the meter and its measurements, encoded as Smart Message Language (SML) "files".

SML is a generic "communication protocol for applications in the field of data acquisition and device parameterization" [24] commonly used to transmit measurements from electricity meters [7], but also capable of encoding other data, such as measurements of gas, heat, or water usage, as well as information about the state of the meter itself.[14] To unambiguously identify the meaning and unit of these data points, SML uses the Object Identification System (OBIS), a standard that defines a hierarchical numbering scheme to categorize data points based on the medium that is being measured (electricity, gas, heat, etc.), the physical quantity (current, power, energy, etc.), as well as any sub-categories required by the metering application (such as the multi-tariff counter feature of the ED300L) [8].

[13] Classes A, B, and C correspond to nominal accuracies of ±2.0 %, ±1.0 %, and ±0.5 % respectively [10].

[14] EDI@Energy OBIS-Kennzahlen-System, Bundesverband der Energie- und Wasserwirtschaft e. V., https://www.bundesnetzagentur.de/DE/Service-Funktionen/Beschlusskammern/Beschlusskammer6/BK6_31_GPKE_und_GeLiGas/Mitteilung_Nr_40/Anlagen/OBIS-Kennzahlensystem%202.2b.pdf

Multiple implementations of the SML protocol already exist, such as one in C [15] and an independent one in Java [16]. Of these two, `jSML` already includes functionality to interface with an electricity meter via a hardware serial port, so it was chosen as a basis for our SML software.

Interfacing with the ED300L's INFO interface also requires hardware instrumentation to receive the optical signals emitted by the meter. The circuitry required for this is relatively simple [17] and the developer of the "IR TTL read head" also sells pre-assembled devices for approximately 20 €. For convenience and to ensure reliable operation, we decided to use this well-established device as our hardware-interface solution.

To facilitate testing and debugging of the SML interface, one of `jSML`'s reference programs was modified to convert SML data to a human-readable format. After successfully testing our electricity meter in combination with the optical receiver circuit, both were installed alongside the existing measurement instrumentation and the optical receiver was connected to the single-board computer already monitoring the output from the single-phase energy meters described in section 2.3. To be able to continuously receive and collect measurement data from the meter, the SML interface program was then extended to implement `collectd`'s plugin interface, as described in appendix A.

However, the ED300L meter did not see extensive usage in our particular setup, due to some disadvantages compared to the other measuring devices that were already present: for one, this model of energy meter only provides a measurement accuracy of 2.0 %, compared to the 1.0 % offered by the Eltako single-phase meters (though it does exceed the 3 % accuracy claimed by our rack PDU). But more importantly for our use case, the ED300L can only report the total power and energy consumption of all devices connected to its three-phase electric supply. Especially when using different cluster nodes for different tasks, this does not provide the fine-grained measurement results offered by its competing technologies.

Ultimately, this type of meter can therefore only be considered a possible supplement for an existing setup, or a compromise when more extensive solutions cannot be installed. If possible, using individual meters such as the ones examined in section 2.3 provides significant benefits in terms of measurement accuracy and granularity.

2.5. Alternatives

In some situations, installing fixed measurement equipment might not be a viable option, for example because the professional installation of mains-voltage equipment would be too costly. For those cases, it is useful to have a "plug and play" alternative to the fuse-box-mounted meters from section 2.3 and section 2.4 that is also more compact and less costly than the rack PDU from section 2.2.

---

[15] `libsml`: Implementation in C of the Smart Message Language (SML) protocol, `https://github.com/dailab/libsml`

[16] jSML Overview – openmuc.org, `https://www.openmuc.org/sml/`

[17] IR-Schreib-Lesekopf, TTL-Interface, `http://wiki.volkszaehler.org/hardware/controllers/ir-schreib-lesekopf-ttl-ausgang`

Unlike the aforementioned solutions, this kind of device is more likely to be found in the consumer market, since the idea of measuring energy consumption is also gaining traction among private individuals. These consumer-grade power meters are, however, often designed to only provide a direct human interface — usually an electronic display on the device, or a proprietary service hosted by the device manufacturer. Devices with publicly available specifications for their machine-to-machine interfaces are sparse, making it difficult to find one that can be used to collect measurement data points and store them over an extended time period.

While this category of devices was not part of the research focus of this thesis, the author is using multiple AVM FRITZ!DECT 200 "smart plugs" [18] in his home setup. They are pre-programmed to connect to a router from AVM's FRITZ!Box series via the DECT radio standard commonly used for cordless telephone handsets. The router then collects data samples from all connected plugs and displays them via its default web interface. Additionally, AVM publishes a regularly maintained specification of a custom HTTP-based protocol [19] that can be used to retrieve these data samples programmatically.

A cursory search also yields open-source software implementations for the EDIMAX Smart Plug family of devices. [20][21] However, these are based on analysis and reverse-engineering of EDIMAX's proprietary network protocol, [22] so there are no guarantees regarding their reliability and continued support.

Both these meter models have an advantage in their compact "wall plug" form factor, which makes it possible to install them with minimal effort. Unfortunately, like many other devices intended for consumer use, their interfaces do not follow any of the widely used standards described in the previous sections (such as SML, SNMP, or the S0 interface). Additionally, both require manufacturer-specific hardware (an AVM router) or software (EDIMAX's Internet service and smartphone app) to function in their default setup.

In conclusion, the meters listed earlier in this chapter, being explicitly targeted at industrial and enterprise use, proved to be a more accurate and more reliable way to measure the total power consumption of our computer cluster, compared to consumer-grade hardware that lacks stable and well-defined interfaces, or integrated measurement features that only provide a partial view of a computer's power consumption.

[18] The term "smart plug" is often used to describe devices with a compact form factor that plug directly into a wall socket and can be controlled via a web interface or smartphone app (hence "smart") provided by the manufacturer.

[19] AVM Home Automation HTTP Interface, `https://avm.de/fileadmin/user_upload/Global/Service/Schnittstellen/AHA-HTTP-Interface.pdf`

[20] `ediplug-py`: Simple Python class to access a "EDIMAX Smart Plug Switch SP-1101W", `https://github.com/wendlers/ediplug-py`

[21] `ediplug`: Python interface to Edimax Smartplug, `https://github.com/bablokb/ediplug`

[22] Deobfuscating the Edimax SP-2101W cloud protocol, Guntram Blohm, `http://blog.guntram.de/?p=37`

3. Data Storage and Visualization

To make use of the large number of data points acquired in chapter 2, they need to be stored in a scalable and easily accessible way. Additionally, it is desirable to provide a human-readable, i.e. preferably graphical, representation.

This kind of performance-data collection and visualization is a very common problem for administrators of computer clusters of any size, so numerous solutions already exist. Some of these tools (`MRTG`, `routers2`, `collectd`, and `Grafana`) were already in use at our institution when the energy-measurement systems were set up. In this chapter, we will examine their individual advantages and disadvantages, as well as those of some other commonly used performance-data visualization tools.

Ultimately, our measurement system was configured to use `collectd` to transmit data to a central `Graphite` instance, mainly for the very flexible data post-processing options provided by the combination of `Graphite` with `Grafana`. A more detailed rationale for this decision is laid out in section 3.3.

3.1. General Architectural Structure

Acquiring, storing, and visualizing time-series data involves several distinct steps, shown as a generalized abstraction in in figure 3.1.
In order, data points need to be:

- **generated** by some (physical or virtual) process
- **acquired** in a machine-readable format (preferably in an automated fashion)
- **transferred** from the acquisition program to the storage program (locally or over a network)
- (optionally) **aggregated** or otherwise adapted to fit a certain storage format
- **stored** persistently
- **retrieved** from storage, while selecting only the data points required at the time
- (optionally) **post-processed** (e.g. combined or transformed)
- **rendered** into a visual representation for human use

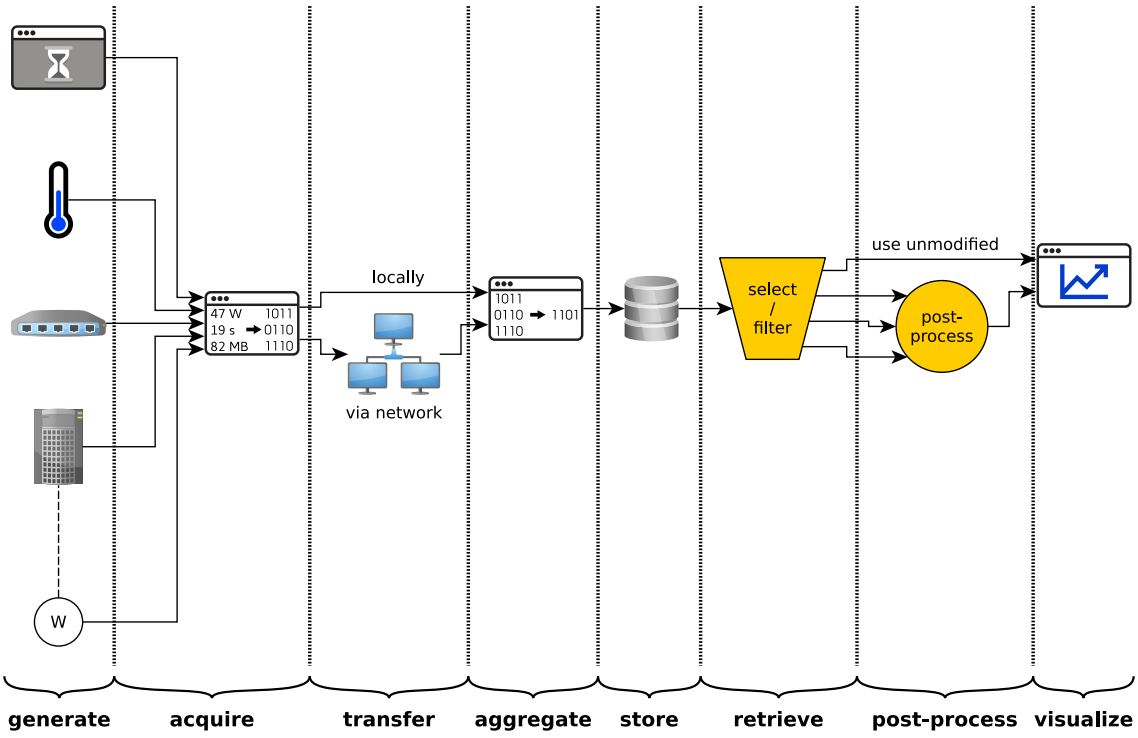

Figure 3.1.: General structure of the data processing chain of a monitoring system, as described in section 3.1

The tools listed in section 3.2 each perform different parts of this processing chain. As a general synopsis, an overview of their respective features is provided in table 3.1.

Note that in many cases, these functions are delegated to external libraries or programs. To provide a complete overview, they are marked as "supported" (✓), and their specific implementation is explained in the appropriate textual description.

Development of Monitoring Paradigms

It is important to note that the approach to implementing a full monitoring stack has changed over time. For the purpose of this overview, we will divide them into three "generations":

Originally, both the systems being monitored and the computers doing the monitoring were few in numbers and rarely changed, as computer hardware of any scale was expensive to purchase and virtualization technology was still in its infancy. [1] This made it the most sensible choice to configure a single central instance to gather data by polling a fixed set of nodes — see for example `MRTG` (section 3.2.1) and `Munin` (section 3.2.4).

This approach remained suitable for years, but eventually large-scale systems with a greater number of individual nodes became more common. Especially in cases where these nodes were inhomogeneous, i.e. yielded different kinds of data, pre-configuring

[1] Consider as an example VMware, which was only founded in 1998, and released its first virtualization solution in 1999.

	generate	acquire	transfer	aggregate	store	retrieve	post-process	visualize
MRTG	–	–	✓	✓	✓	–	–	–
RRDtool	–	–	–	✓	✓	✓	✓	✓
routers2.cgi	–	–	–	–	–	✓	✓	✓
Munin	–	✓	✓	✓	✓	✓	–	✓
collectd	–	✓	✓	–	–	–	–	–
Graphite	✓	–	✓	✓	✓	✓	✓	✓
Grafana	–	–	–	–	–	✓	✓	✓
Telegraf	–	✓	✓	–	–	–	–	–
InfluxDB	–	–	✓	✓	✓	✓	✓	–
Chronograf	–	–	–	–	–	✓	✓	✓
Kapacitor	–	–	–	–	–	✓	✓	–

Table 3.1.: Data processing steps performed by the tools described in section 3.2

all their intended data outputs centrally could be quite cumbersome. To solve this new challenge, new tools like `Graphite` (section 3.2.6) were created, that would automatically store all data they receive, without requiring each individual time series to be known and configured in advance. However, this second generation of tools still assumes that data samples will arrive at regular intervals and that samples of the same time series will keep arriving for an extended period of time after they are initially discovered.

While not an issue in our particular use case, assuming time series to be long-lived can create another set of limitations, which when disregarded can cause issues such as excessive resource consumption — consider a tool that preallocates a relatively large amount of storage for each time series being used in an environment where time series are often short-lived and new ones appear frequently. For this type of scenario, a new approach to time-series data storage was introduced, which will be exemplified by the `TICK` monitoring stack in section 3.2.8.

Ultimately, each of the three general scenarios above still exists today, and it is up to the system administrator to choose a suitable tool for their use case. The following section will provide some information to be used in this decision process.

3.2. Available Technologies

Automated performance monitoring has been a topic of interest to system administrators ever since the inception of computer networks; after all, it is important to know what one's hardware is doing to be able to detect and fix anomalies, as well as plan expansions when resource utilization is approaching the hardware's physical limits.

The software available for this purpose has evolved greatly over time, both in terms of their approach to configuring data sources (as detailed in section 3.1) and in terms

of usability features and visual appearance: from `MRTG`, first released in 1995 [2], simply generating static PNG graphs, to `Grafana`, a feature-rich Javascript-based web panel released as recently as January 2014 [3].

Additional features often come at the expense of increased complexity, resource usage, or both, so this section will give an overview of the strengths and drawbacks of some of the more established and well-known performance monitoring and visualization tools.

3.2.1. MRTG

The conceptually oldest tool used in our setup is the Multi Router Traffic Grapher (`MRTG`), developed primarily by Tobi Oetiker and first released in 1995.

As the name suggests, it was originally written to visualize network traffic across one or multiple routers, as a graph of network throughput over time. This is also reflected in the data storage format (further described below) that stores pairs of measured values — usually incoming and outgoing traffic figures — in the same logfile.

To acquire the real-time throughput data, `MRTG` uses the standardized Simple Network Management Protocol (SNMP) implemented by many enterprise networking devices. Since SNMP exposes not only network throughput data, but can also provide arbitrary other data (if supported by the device), `MRTG` can also monitor "things such as System Load, Login Sessions, Modem availability and more" [4].

`MRTG` stores its acquired data points in a plain-text file format (called a "logfile" by the developer). To prevent the file from growing indefinitely, data points are automatically aggregated over certain fixed timespans, with decreasing resolution as the data ages (e.g. 5-minute aggregates of the most recent data, down to 24-hour aggregates for data from two years ago), and dropped from the database after about two years. This means the logfile will have a nearly constant size, varying only by a few bytes as the numeric values within it change. For reference, the default configuration produces a logfile of 2 536 lines and about 48 KiB for each pair of measured values.

3.2.2. RRDtool

While easy to create and parse, the plain-text format from section 3.2.1 limits how data can be consolidated (since the logfiles only store means and maxima) and according to its developer can also have an impact on performance.

To address this, Oetiker eventually created a more versatile storage format that he simply named Round Robin Database (RRD). Oetiker's `RRDtool` combines functions for data acquisition (interpolating data points when the timestamps of the acquired data don't match the fixed schedule of the RRD), consolidation (allowing to choose between calculating the data's mean, minimum, maximum, or simply storing the most recent value for a given time slot), and even generating static PNG graphs from the

[2] MRTG Download, `http://oss.oetiker.ch/mrtg/pub/old/`

[3] Grafana 2.0, the future, and raintank, `http://staging.grafana.org/blog/2015/01/12/grafana-2.0-the-future-and-raintank/`

[4] What is MRTG ? [sic], `http://oss.oetiker.ch/mrtg/doc/mrtg.en.html`

data.[5] Again for reference, an RRD for a pair of values created with the default config (two-year data retention) generates a binary file about 93 KiB in size.

MRTG natively supports using RRDtool to store and graph its data. When configured to use it, however, it disables its internal graph and index file generation, instead requiring an external tool for this purpose. This separation of concerns actually vastly improves MRTG's performance (Oetiker himself claims an 80 % reduction in runtime per invocation), since the human-readable output no longer has to be pre-written every time new data is acquired and can instead be generated on demand only when it is actually of interest to the user.[6]

3.2.3. routers2.cgi

The MRTG developer recommends three scripts that each implement this on-demand visualization: 14all.cgi ("one for all"), mrtg-rrd (which is meant to be a successor of 14all.cgi), and routers2.cgi.[6]All three use the Common Gateway Interface (CGI) to interface with a web server and dynamically generate web pages.

Of these three alternatives, routers2.cgi[7] is the most versatile, offering features that neither 14all.cgi nor mrtg-rrd provide, such as grouping graphs into a custom hierarchy, and changing their rendering size and time scale from the web interface. It is also the most actively maintained (with the most recent release being from 2014, compared to 2003 for both other scripts), and was therefore chosen as the MRTG front end on our infrastructure.

As mentioned in section 3.2.1, routers2.cgi is not limited to displaying network traffic, and was in fact mainly used by us for measurements such as CPU load and some of our energy consumption data (see sections 2.1 and 2.3). It was, however, later supplemented and eventually replaced by Grafana, for reasons that will be explained in section 3.3.

3.2.4. Munin

First released in 2003, Munin bears many similarities to its predecessors, but at the same time distinguishes itself through a new approach at gathering data.

Like MRTG, Munin runs on a central monitoring server, which then collects data from multiple nodes by connecting to them at regular intervals. Its data visualization also looks very similar to that of MRTG by default, as it uses RRDtool to generate its graphs.

The underlying architecture for data acquisition is, however, vastly different: where MRTG uses the standardized SNMP to connect to its monitored systems, Munin developed a custom plain-text communication protocol.[8] This protocol intends to make it easier to extend Munin with additional data sources ("plugins"), as plugin developers no longer

[5]About RRDtool, http://oss.oetiker.ch/rrdtool

[6]How to use RRDtool with MRTG, http://oss.oetiker.ch/mrtg/doc/mrtg-rrd.en.html

[7]routers2.cgi, http://www.steveshipway.org/software/rrd/f_routers.html

[8]Writing a munin plugin, http://guide.munin-monitoring.org/en/latest/plugin/writing.html

have to obtain OIDs to represent the data reported by their plugins (as would be the case with SNMP). While this makes it easier to write plugin, it means the user no longer has the convenience of relying on a preconfigured SNMP service on their monitored devices and instead has to manually install and configure a `Munin` "node" on each of them. It does, however, have the advantage of providing greater freedom to configure what data is collected and how it is combined into graphs, as `Munin` eliminates the restriction to (incoming, outgoing) tuples enforced by `MRTG`.

Its easy extensibility has lead to a wealth of user-contributed `Munin` plugins becoming available, some of which are shipped with the main Munin release. `Munin` has also introduced the concept of "autoconf" plugins that try to automatically determine all parameters required for their operation (e.g. the location of the system load average file, or the credentials required to monitor an SQL database daemon), and either activate themselves automatically or notify the user what caused their auto-configuration to fail. This system brings `Munin` very close to being a "plug and play" solution that the user merely needs to install and point at a set of nodes to start collecting useful data.

`Munin` plugins can freely define the name they are displayed under, as well as the labels, graph type (e.g. lines, stacked, etc) and RRD data-source type (see section 3.2.2) of their measurement value or values. The plugin interface is also programming-language independent, as the `Munin` node simply runs each plugin as an executable and parses the text emitted on its `stdout` stream. That same textual data and metadata format is also used in the network communication between the master and each node. [9]

From there, the process of acquiring data is again simple polling: the `Munin` master regularly contacts each node sequentially, asks for a list of available plugins, and fetches the configuration and current values for each of them. [10]

As mentioned above, Munin uses `RRDtool` as its data-storage back end, with a separate RRD file for each plugin on each node. To generate graphs from the raw data, `Munin` needs to parse each of them, which can put a significant strain on the resources of the monitoring server, especially with `Munin`'s default configuration of querying its nodes every five minutes and regenerating all graphs afterwards. Like with `MRTG`, there is the option to instead use a CGI script interfaced with a web server to generate the graphs on demand, thus avoiding unnecessary load on the system. Conveniently, Munin already ships the necessary CGI scripts in its default distribution. In addition, the CGI approach enables advanced features such as viewing data from arbitrary time periods, instead of just the past day, week, month, or year.

Overall, `Munin`'s data acquisition and visualization features exceed those of the monitoring systems listed thus far, but it still lacks a way to combine and compare data from multiple time series. This shortcoming will be addressed by the set of tools described in the following sections.

[9] Data exchange between master and node, `http://guide.munin-monitoring.org/en/latest/master/network-protocol.html`

[10] The Munin Protocols, `http://guide.munin-monitoring.org/en/latest/architecture/protocol.html`

3.2.5. collectd

A different approach to the kind of versatile plugin-based monitoring introduced by `Munin` in section 3.2.4 is taken by the "system statistics collection daemon" `collectd`: rather than relying on a simplistic text-based plugin interface, it strives to implement as much of its functionality as possible in natively compiled libraries loaded by a single C program. This avoids the overhead of having to spawn new processes for every single query to every single plugin (as is the case with `Munin`). `Collectd` is therefore arguably more efficient in collecting measurements and goes as far as shipping with a default measurement interval of ten seconds.

`Collectd` also takes the modularisation approach one step further: not only the acquisition of data is split into individual plugins; the same is done for forwarding or storing it, and even for logging of `collectd`'s own events. This expands the possibilities beyond merely storing values in an RRD and generating static graph images from there. The output options implemented by so-called "write plugins" shipped with collectd include CSV files, `RRDtool` (section 3.2.2), `Graphite` (section 3.2.6), and numerous others.

This plugin-centric architecture goes hand in hand with an important distinction of `collectd`: many of the write plugins can store data locally, on the same system where it is collected, so it is not always necessary to have a remote "master" system poll it. Instead, collectd instances will usually be configured to write data of their own accord – either to a local file, or to a remote service over the network.

By default, `collectd` will try to use RRD files as its storage back end, since the format is widely used and requires very little configuration (essentially just a path where the files should be written). The files can then converted to a visual representation using third-party tools like the `router2.cgi` script described in section 3.2.3 (`collectd` explicitly considers it out-of-scope to include a visualization solution). This does however reintroduce the relative inflexibility that is common to all monitoring solutions described so far: data is averaged over a number of fixed time periods and left to later be rendered into a static graph.

Unlike the previously explored tools however, `collectd` provides other, more flexible, options. Specifically, when `collectd` was first considered for our chair's monitoring needs in 2014, a very common solution was a stack of `collectd`, `Graphite`, and `Grafana`, which we will examine in more detail below.

3.2.6. Graphite

`Graphite` may be considered a member of the second generation of monitoring tools described in section 3.1, as it automatically accepts and stores new measurement time series without prior explicit configuration for them. It, however, does not perform any measurements by itself; instead it expects to be "fed" via one or multiple of its network-based ingestion protocols.

Internally, `Graphite` is split into three independent components: [11]

[11] Graphite – The Architecture in a nutshell, `http://graphite.readthedocs.io/en/latest/overview.html#the-architecture-in-a-nutshell`

- a network daemon (`carbon`) that receives time-series data and manages access to the database,
- a library for working with databases using the purpose-built `whisper` file format, and
- a web-based application (`Graphite-web`) for rendering and viewing the data on-demand.

Carbon Network Protocol

`Graphite`'s network interface supports multiple protocols for feeding in data: for one, there is an application-specific plain-text format; its existence justified by the simplicity of its use. This is the protocol used by `collectd`'s `Graphite` plugin.

To reduce overhead (such as having to specify the full names of data points repeatedly, and encoding numeric data as sequences of ASCII characters), `carbon` adds support for "pickling", Python's standard library for "serializing and de-serializing a Python object structure" [12], i.e. converting a high-level data-representation object into a byte stream, and back again. This allows more efficient encoding of data, albeit at the expense of requiring an implementation of Python's pickling process. While third-party implementations exist for some languages [13], their continued interoperability with the native Python implementation is usually not be guaranteed.

A different option was introduced when `Graphite`'s version 0.9.6 added support for the standardized Advanced Message Queuing Protocol (AMQP) [14]: As a "wire-level" protocol specification, AMQP ensures interoperability between any software implementing the standard, similar to HTTP's role for exchanging hypertext on the Internet. However, AMQP competes with other protocols such as the ISO-standardized Message Queue Telemetry Transport (MQTT), and — especially for the use case of transmitting measurements for monitoring purposes — various application-specific protocols like those mentioned previously. AMQP is supported by `collectd` (via an included plugin) as both a data source and data sink, and thus provides an additional option for connecting collectd to `Graphite` as its data storage.

Whisper Storage Format

Data storage in `Graphite` is in some ways similar to the storage solutions explored previously: it is limited to numeric values, structures time series into a hierarchy of directories and files on the filesystem, and automatically aggregates data based on a fixed configuration to maintain a constant size for each time-series storage file.

---

[12] `pickle` – Python object serialization, `https://docs.python.org/3/library/pickle.html`

[13] Pyrolite: Java and .NET interface to Python's pickle and Pyro protocols, `https://github.com/irmen/Pyrolite`

[14] Graphite 0.9.6 release notes, `http://graphite.readthedocs.io/en/latest/releases/0_9_6.html`

Conceptually similar to RRDtool, the whisper storage format has some design goals that required the creation of a different format when Graphite was first written in 2006. Most importantly, RRDtool had no native support for caching and aggregating write operations at that time: every incoming data point had to be written to the database immediately and individually. On large deployments, this would impose extreme requirements on the performance of the underlying storage on the central monitoring node. Whisper intends to mitigate this by caching data points in memory and only writing them to disk at (larger) fixed intervals, thus reducing the number of write operations required. This makes the format more suitable for traditional rotary magnetic storage (hard disks), where frequently repositioning the head (to execute large numbers of write operations of individual files) takes a non-negligible amount of time and thus decreases disk throughput and performance. Even for solid-state drives, this can result in a performance improvement, as their number of input/output operations is still limited by the implementation of the hardware message bus.

RRDtool has since gained a similar facility in the form of the rrdcached daemon [15] that also accumulates a certain number or timespan of data before writing it to disk. Unlike carbon (the persistent daemon that wraps all access to Graphite's database), rrdcached does not provide a way to access the contents of the cache; data has to be flushed to the storage file to become accessible for graphing and other kinds of post-processing.

Graphite is strongly geared towards using its default whisper database as a storage backend. There are plans to develop a new back end named Ceres that will support distributing the database across multiple servers and will replace whisper's fixed-size format with something more flexible, but at the time of writing, Ceres is marked as being "not actively developped [sic] at the moment". [16] Graphite theoretically also supports using alternate databases, including for example InfluxDB and OpenTSDB, but at this point the official documentation only mentions them in passing, without giving details on how to integrate them with Graphite. [17]

Graphite-web Interface

One particularly noteworthy set of features lies within Graphite's web interface. In addition to merely plotting graphs using unmodified values from its data files (like all previously mentioned tools), Graphite-web supports "functions".

Functions allow the user to process and combine time-series data retrospectively. They can change the way graphs are drawn (e.g. stacked, or with multiple y-axes), and can also perform complex transformations and calculations on the data, from simply summing up multiple series to calculating integrals and derivatives, to performing forecasts on

[15] incidentally written by Florian Forster, the primary developer of collectd, and added to RRDtool's version 1.4 in 2008 — see "2008-09-14 09:49" in the RRDtool change history at https://oss.oetiker.ch/rrdtool/pub/CHANGES

[16] The Ceres Database, http://graphite.readthedocs.io/en/latest/ceres.html

[17] Tools That Work With Graphite: Storage Backend Alternates, http://graphite.readthedocs.io/en/latest/tools.html#storage-backend-alternates

their development. [18]

All this functionality is exposed in the `Graphite Browser` web application which provides a tree structure of available time series and menu options to configure the graph rendering options and apply functions, and displays the finished graph as a PNG image. The `Graphite Browser` effectively only serves as a user interface to an API that accepts queries (names of time series, optionally combined with function invocations) and returns either a PNG image or JSON data in response.

Unfortunately, while powerful, the `Graphite Browser` user interface lacks convenience when composing graphs: measurements can only be accessed via a tree structure grouped by their source, which makes it cumbersome to execute common tasks like selecting the same measurement from multiple different nodes for comparison: the measurements are leafs at different positions in the tree, thus forcing the user to click through multiple layers of the tree. In addition, the usefulness of the generated graphs is limited because they can only be rendered as static PNG images, which do not allow the user to accurately read measurement values from the graph.

3.2.7. Grafana

`Grafana` can provide a solution to the lack of convenience features described in section 3.2.6: Instead of being a simplistic viewing tool included with a monitoring solution, Grafana's single purpose is to provide a user-friendly (and visually appealing) front end to a number of data-storage back ends.

`Graphite` is one of those supported back ends, integrated via the JSON output format offered by `Graphite-web`. Since `Graphite`'s advanced post-processing functions are part of the underlying query API, `Grafana` can make use of them as well, and even enables the user to chain functions and combine time series using a graphical representation of the available post-processing steps.

The processed data can then be displayed using the `Flot` JavaScript plotting library, which adds useful features such as hovering over points in the graph to display the exact value at that point, or showing a common time marker on multiple plots on the same page when one of them is hovered over. [19]

Grafana also supports user-editable "dashboards" that combine multiple graphs on a single web page, potentially complemented with numerical representations of histograms of any available time series. Features like these make Grafana a particularly user-friendly tool to explore data that has already been recorded, without having to decide in advance how the data should be plotted.

Its support for several different data-storage back ends, even multiple at the same time, also means that it can be used to combine data from monitoring systems that might otherwise not be compatible with each other. This can be useful when migrating to a different monitoring solution, such as the comparatively new `TICK` stack described in the next section.

[18] Graphite: Functions, `http://graphite.readthedocs.io/en/latest/functions.html`

[19] Flot: Attractive JavaScript plotting for jQuery, `http://www.flotcharts.org/`

3.2.8. Telegraf, InfluxDB, Chronograf and Kapacitor (TICK)

When beginning work on the measurement systems in this thesis, our chair was already in the process of setting up the tool stack of `collectd`, `Graphite`, and `Grafana` to eventually replace the older and less flexible combination of `MRTG` and `routers2.cgi` (see section 3.3), so it was prudent to use the existing infrastructure in this case. While working on this thesis, however, another monitoring tool stack with yet another different approach to storing measurement data increasingly gained popularity, so it warrants a look at its design goals and the implications they have on operating and using it.

The toolchain we will be looking at is known under the acronym `TICK`, named for its components `Telegraf`, `InfluxDB`, `Chronograf` and `Kapacitor`. Developed by the commercial organisation InfluxData (originally named Errplane), it is intended to provide a "complete platform for metrics and events" [20]. Its components span all parts of the data collection (`Telegraf`), storage (`InfluxDB`), processing (`Kapacitor`), and visualization (`Chronograf`) chain.

`Telegraf` bears much similarity to `collectd` in that it delegates all functionality implementation to plugins, both for collecting data (either from direct measurements or from other networked services) and for passing it on to other services for processing and storage. It supports a number of input formats, including those used by the de-facto industry standard service-monitoring tool `Nagios` [21], and also the network data formats of `collectd` and `Graphite`.

The paradigm change that makes `TICK` worth mentioning above the large number of similar tools lies in `InfluxDB`'s way of storing time-series data: the tools we examined so far always operated under the assumption that time series would keep receiving data points for extended periods of time, so it made sense to use fixed-size storage formats that preallocate space and aggregate data to lower resolution over time. For observing indicators of system performance, this is usually sufficient, as fine-grained time resolution becomes less important for time periods that lie further in the past.

`InfluxDB` changes this design parameter, instead aiming at time series that are not guaranteed to be long-lived. It therefore removes the mandatory aggregation of data, instead storing all data points individually, with a timestamp attached to each one. This makes it far more space-efficient for "sparse" time series, i.e. ones that only receive data points sporadically. Since data is now no longer guaranteed to be uniform, `InfluxDB` also replaces the linear numeric query methods of the previously mentioned databases with a query language more closely resembling SQL. Beyond this major characteristic however, `InfluxDB` closely resembles what we already know from previous tools: it supports multiple ingestion protocols (both custom ones and third-party protocols like those of `collectd` and `Graphite`), caches data in memory to optimize read/write access to the underlying file storage, and supports "retention policies" to govern how long data is stored and hence how large the database is allowed to grow.

[20] InfluxData (InfluxDB) – Open Source Time Series Database for Monitoring Metrics and Events, `https://www.influxdata.com/`

[21] claimed to be used by "9000+ customers" by its developer Nagios Enterprises (`https://www.nagios.com/`), and also forked into several open-source projects, such as `Icinga`, `Shinken`, and `Naemon`

In the TICK tool stack, data post-processing is actually available in two places: InfluxDB itself offers "continuous queries" that "run automatically and periodically on realtime data and store query results in a specified measurement [time series]" [22]. On the other hand, Kapacitor is purpose-built to receive data streams or query data from InfluxDB and apply arbitrarily complex transformations and even user-defined functions. [23] Both provide the options of writing the processed data back into InfluxDB so it does not have to be processed repeatedly, and Kapacitor can also trigger alerts or arbitrary handlers based on the conditions applied to the data, opening it up to additional use cases, such as alerting or load-balancing.

Finally, Chronograf integrates with the features of the other tools in the stack, particularly the post-processing and alerting options of Kapacitor, in addition to providing JavaScript-based graphs and dashboards similar to those offered by Grafana.

Overall, InfluxDB and its related tools are a versatile monitoring solution and exemplify the ongoing development of new monitoring paradigms, as outlined in section 3.1, but its improvements over existing tools did not warrant the effort of setting up a new toolchain in our particular use case. Instead, our requirements were already satisfied by a combination of some of the tools described earlier in this chapter, the setup of which will be detailed in section 3.3.

3.3. Selected Toolchain for our Use Case

The previous section offered an insight into the different options for storing and visualizing time-series data, and particularly the assumptions they made about the environment they would be used in. This information now needs to be applied to the use cases in our environment.

As described in chapter 1, our chair operates a number of clustered computer systems for the purpose of software verification. In addition to these single-purpose machines, there are several individual machines used as workstations by members of staff.

Originally, a solution was required to monitor the performance and usage intensity of our software-verification cluster, and it was an obvious step to also include performance data from the workstations in the same monitoring system. Our system administrators at the time had already successfully used MRTG and router2.cgi on other systems, and the standardized SNMP interface made it easy to include custom data sources, such as the VerifierCloud software-verification environment.

While the data ingestion options were sufficient, the visualization was unfortunately inflexible. If a user required a graph comparing verification-task throughput to CPU load, for example, the administrator would have to set up logging to a custom RRD file, which would also start out empty, with no feasible way to import the existing data. At the time, time-series storage tools with flexible post-processing options were not yet as ubiquitous as they are at the time of writing this thesis, so small-scale testing quickly

[22] InfluxDB Documentation: Continuous Queries, https://docs.influxdata.com/influxdb/v1.2/query_language/continuous_queries/

[23] Kapacitor Version 1.3 Documentation, https://docs.influxdata.com/kapacitor/v1.3/

converged on the combination of `collectd`, `Graphite`, and `Grafana`, that had already seen use and documentation by a number of system administrators on Internet forums and blogs.

As an added advantage, `collectd` made it possible to configure the acquisition of measurements once and deploy the configuration to all nodes (both the verification cluster and the workstations) via the pre-existing automated software-deployment system at our institution, without requiring explicit per-node configuration on the central monitoring server.

Most measurements that were previously collected by `MRTG` via SNMP were already supported by `collectd`'s default plugins, and the custom integrations could easily be adapted to interface with `Graphite` via its plain-text protocol, or in some cases rewritten as native `collectd` plugins for an added performance increase (see the matching technology descriptions in chapter 2 for the approaches taken for each of the energy-measurement sources in particular).

One question that warrants further explanation is why we did not choose the `TICK` stack (or parts of it) over `Graphite` and `Grafana`, considering its advantages in terms of storage format and post-processing options. First and foremost, the components of `TICK` lacked renown at the time our monitoring system was restructured: `collectd` and `Graphite` were already established and mature (being ten [24] and eight years old in 2014), and `Grafana` — while newly created — outpaced its pendant `Chronograph` in terms of development speed. Even now, with `TICK` maturing, its main feature of flexible data storage does not pose an advantage on our infrastructure, as our pool of machines and their set of measurements does not usually change. Customizable alerting for anomalies is a helpful addition, but on our infrastructure has at this point already been implemented using the `Bosun` alerting system [25] (the specifics of which however expand beyond the scope of this overview of storage and visualization solutions).

Overall, `Graphite` and `Grafana` together have been able to satisfy all our requirements and have been intuitive to work with. Only `Graphite` required some modifications to its configuration so it would keep a larger cache of data before flushing it to disk: otherwise, it was prone to overloading the system with write requests to the disk. Since `Graphite` allows access to its in-memory cache, this modification had no noticeable impact on the access latency for existing time series; it did however cause new time series to only appear after a certain delay (the time until they were flushed to disk for the first time) when they were initially populated with data (e.g. when new nodes or new monitoring features were added).

When the choice of monitoring solution was finalized and the software had been tested to be operating reliably, the measurement solutions from chapter 2 were updated to deliver their output to `Graphite`, and work on using their data for optimizations could begin.

[24] considering the source files from collectd's v1.0 release on `https://collectd.org/files/`

[25] Bosun, `https://bosun.org/`

4. Reducing the Energy Consumption of a Software-Verification Computer Cluster

The Chair for Software Systems focuses its research on developing and improving methods for formal software verification. The algorithms used for this often require large amounts of computer memory and CPU resources [1] and need to be run repeatedly, often in quick succession, to test the effects of implementation changes on their runtime and accuracy.

This makes it crucial to have extensive computational resources readily available, the most cost-effective solution to which is setting up a dedicated computer cluster to run these software verification tasks. Tasks can then be assigned to nodes within the cluster by a task queueing system, as will be described in section 4.1.

However, these resources are not used continuously, since the influx of new tasks depends on the activity of human users (which tends to be governed by factors such as the time of day, weekends or holidays, and project schedules). When all items in the task queue have been processed, the cluster sits idle, not running any useful calculations, but still consuming electric power. It is an obvious step to evaluate the possibility of powering down the cluster or some of its individual nodes when there is no current use for them, thus reducing power and overall energy consumption.

Automating the process of powering down idle systems, as well as powering them up when they were needed again, constituted an integral part of this thesis. Section 4.2 details the process of designing and implementing an "auto-shutdown" feature within our existing `VerifierCloud` task-queueing framework.

Chapters 2 and 3 described our work on setting up an infrastructure for measuring, storing, and visualizing the power consumption of our computer clusters, originally with the goal of comparing the energy consumption of different software-verification algorithms. Conveniently, this infrastructure is also well suited to analyze the potential energy savings achievable by dynamically powering cluster nodes up and down based on current resource demands. Section 4.3 will examine these potential savings in more detail.

4.1. The VerifierCloud Software-Verification Task-Queueing Framework

The introduction to chapter 4 already hinted at the infrastructural challenges associated with testing software-verification tools: forming an accurate picture of a tool's performance and accuracy requires a large number of individual tests that each require a significant amount of computational resources. The most efficient and economical way to provide these resources is usually to set up a cluster of individual nodes, as the number of CPU cores and the amount of memory on a single node are restricted by limitations of the available hardware platforms.

This does mean, however, that resources are no longer available on a single device and are instead split across multiple nodes, each of which is running an independent operating system kernel. Programs therefore cannot rely solely on multi-threaded local execution (i.e. running separate threads of the same program on the multiple CPU cores of the local device) to fully utilize the capabilities of the cluster. Instead, the intended work load has to be split into "tasks" that are then executed on each node individually. While possible, performing this work-load distribution manually would be inefficient and time-consuming, so an automated solution should be found instead.

Requirements for a Task-Distribution Solution

A suitable automated task-distribution solution has to account for specific requirements imposed by the intended application of testing software-verification tools: in general, software-verification tools analyze a given program to prove or disprove whether it conforms to a given specification. This means that they require both the source code of the program under test, as well as a representation of the specification, to operate. Additionally, their results need to be collected and made available to the user.

The distribution of the computer clusters in our use case also creates challenges for the task-distribution process: the Chair for Software Systems collaborates with teams at other institutions, some of which contribute computer resources to the overall cluster system. The geographic diversity and accompanying high network latency of these locations rules out some means of transfering the input and output files (such as the commonly used Network File System (NFS), which is known to perform poorly in high-latency scenarios [1]).

After considering all of these application-specific requirements, it was decided to implement a custom software solution, the `VerifierCloud`, in 2011.

System Architecture of the VerifierCloud

The `VerifierCloud` serves multiple purposes: in addition to distributing tasks to nodes of the cluster, it also serves as a queueing system when all nodes are currently busy pro-

[1] File Sharing on the WAN: A Matter of Latency, EE Times, `http://www.eetimes.com/document.asp?doc_id=1272058`

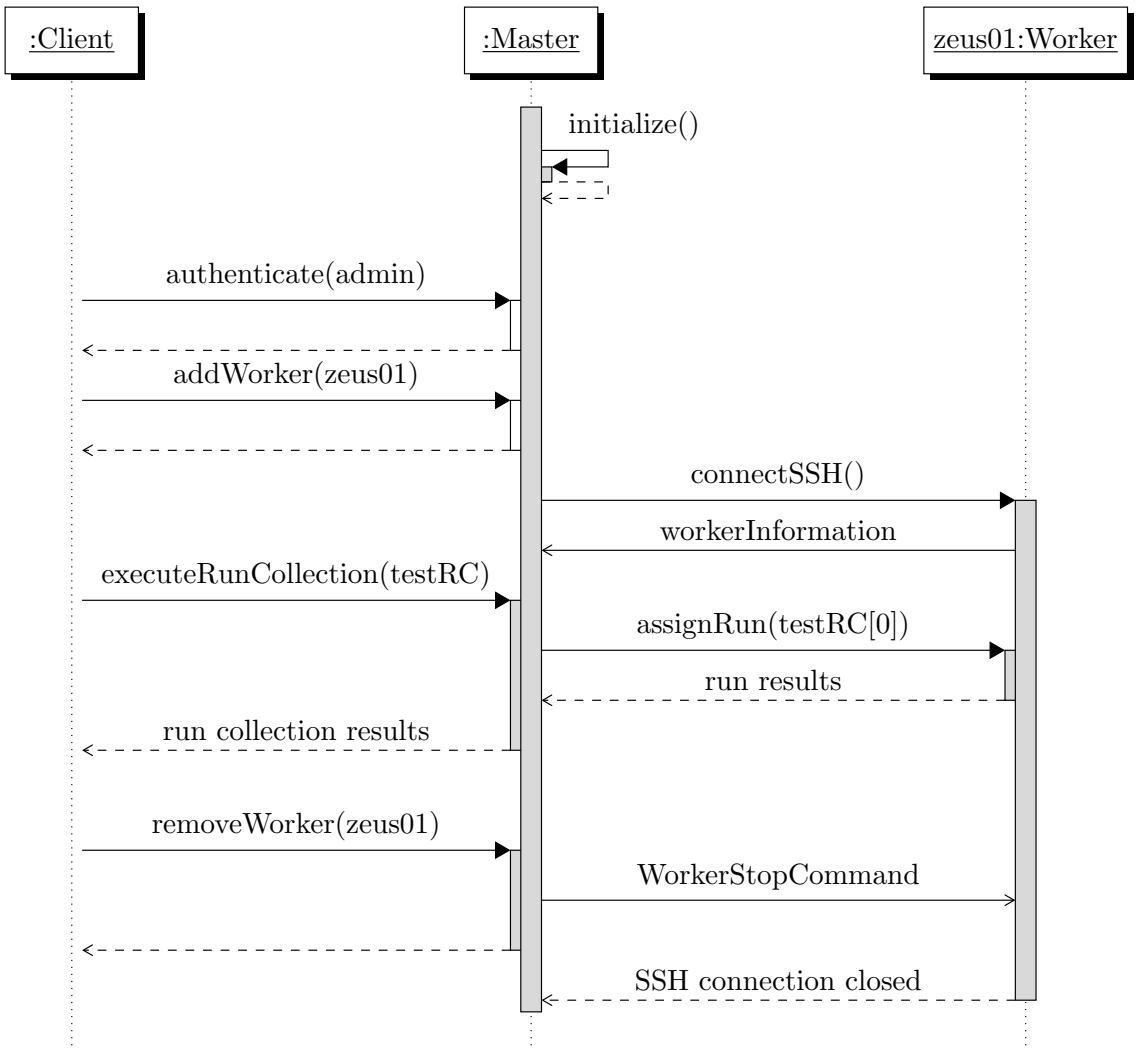

Figure 4.1.: Communication sequence of basic `VerifierCloud` operations (described in section 4.1)

cessing their assigned tasks. The queueing functionality offers some additional features (such as requirements and limitations) that will be described later in this section.

All network communication is conducted via a central instance, the *master*, that accepts *run collections* (a set of *runs* that each contain a *task* and its required input files) and *commands* from *clients*, starts *worker* instances on a given set of nodes, and assigns *runs* to each of them. Some parts of the implementation depend on functionality provided only by the Linux operating system kernel (such as the `cgroups` interface), so the *master* and *worker* are only intended to run on Linux-based systems.

Figure 4.1 shows a typical communication sequence where a user operates an instance of the `VerifierCloud` interactively:
As the core component of the system, the master has to be started first. Once it has finished initializing, the user can use one of the various client implementations (described

in more detail below) to connect to the master and instruct it to start a worker on a node with a given hostname. The master then attempts to establish an SSH session to the given node and launch a worker instance on it. If the SSH connection and worker launch succeed, the worker performs its own initialization sequence and then connects to the master through a port-forwarding tunnel established over the SSH connection. Upon receiving a connection from the worker, the master adds it to an internal pool of available workers, thereby making it available for task execution.

Regardless of the number of available workers, the client can submit a run collection, or even multiple run collections successively. Whenever a new run collection is submitted or a worker becomes available (either after connecting or after finishing execution of a run), the master performs a scheduling algorithm that determines if any runs from the pending run collections can be assigned to a worker for execution. When all runs of a run collection have been processed, the master notifies the client that submitted the run collection and delivers the output produced by the runs it contained.

Beyond this basic example, the `VerifierCloud` has several additional features that increase its usefulness for software-verification applications: its implementation is designed to be as modular as possible, using a well-defined set of serialized Java objects for all network communication between its components (master, workers, and clients). This makes it possible to have multiple separate client implementations, each targeting a specific use case. These include an interactive command-line interface to access the `VerifierCloud`'s internal state (e.g. to manually start or stop worker instances), a command-line tool that can automatically perform verification-tool benchmarks on the cluster, and also a web-based client that provides external collaborators with access to the cluster [18].

Since workers and clients only communicate directly with the master, any special network configuration (such as for cluster nodes at remote locations that need to be connected via network tunnels) only needs to be done once, on the node running the master. The master will then use network port forwarding over SSH connections to ensure that the workers it starts can connect back to the master without further configuration.

The software also provides features for selecting what nodes runs from a given run collection will be assigned to, and what amount of resources each run may occupy. The first is achieved via *requirements* that can optionally restrict a run collection to nodes with a certain CPU model, number of available cores, and amount of available memory (RAM). When a run is assigned to a worker, it will use the full amount of available memory and CPU cores by default, but this can optionally be regulated by specifying a set of *limitations* for memory and CPU usage. The worker will then execute the run in an isolated `cgroups` hierarchy that ensures that it cannot exceed the given limitations — a necessary prerequisite for reliable and reproducible benchmarks [2] and also a useful mechanism for enforcing resource restrictions in competitions like the Competition on Software Verification (SV-COMP) [1]. The implementation of requirements will become relevant when discussing the implementation of on-demand power-up for worker nodes in section 4.2.

4.2. Implementing Automatic Worker Power Control

Automatic power control involves the two distinct functionalities of powering nodes down when they are unused and powering them up when they are needed again. This requires the addition of several features to the master: first, the master has to be made aware of nodes that have been powered down intentionally, so it can consider them for use as workers. Second, the master requires a method of powering on nodes remotely. For this, we utilize the Wake-on-LAN (WoL) standard, as described in section 4.2.2. When all requirement for powering up nodes are in place, the master can then begin powering down worker nodes. In our case, detailed in section 4.2.3, nodes will be powered down when they are used exclusively by the `VerifierCloud` and have been idle for a set length of time. However, choosing a sensible value for this idle timeout requires some considerations that will be explained in section 4.2.4.

4.2.1. Keeping Track of Available Worker Resources

As mentioned in section 4.1, runs are assigned to workers by an algorithm (referred to as the *scheduler* in the context of the `VerifierCloud`) that is triggered whenever a new run collection is submitted or a worker becomes available. This scheduler examines the requirements associated with each queued run collection, compares it to the unused resources on all available worker nodes, and assigns an appropriate number of runs to a worker when it finds a match. If no suitable workers are available, the run collection simply stays queued and is re-examined the next time the scheduler is triggered.

For its suitability check, the scheduler uses two kinds of information about each worker: a map of attributes ("worker information"), such as the worker's CPU model and current amount of unused system memory, and a "worker state" attribute that represents the worker's current condition (e.g. whether it has not yet finished initializing or its node is currently in use and cannot be used by the cluster).

Of the worker states shown in figure 4.2, only workers that are `AVAILABLE` are considered in the suitability check. Workers that have not finished their initialization, are faulty or in the process of terminating, or whose node is currently running processes started by a local user (i.e. is `USER_OCCUPIED`) are excluded.

The original implementation was written without controlled power-down in mind, instead assuming that all nodes would be running continuously and that only the worker instances on them could be terminated and restarted on demand. This meant that the master would simply discard any information it had about a worker whenever the connection to the worker was lost, and would re-add the worker from scratch when the connection was re-established.

With automatic power-down and especially automatic power-up in mind, however, this behavior poses a problem: since the master discards the worker's information (including, for example, its CPU model) as soon as the worker disconnects, it has no way of checking whether the powered-down worker would be suitable for a given run collection. It is therefore essential to modify the master to retain the worker information of any workers that were intentionally powered down.

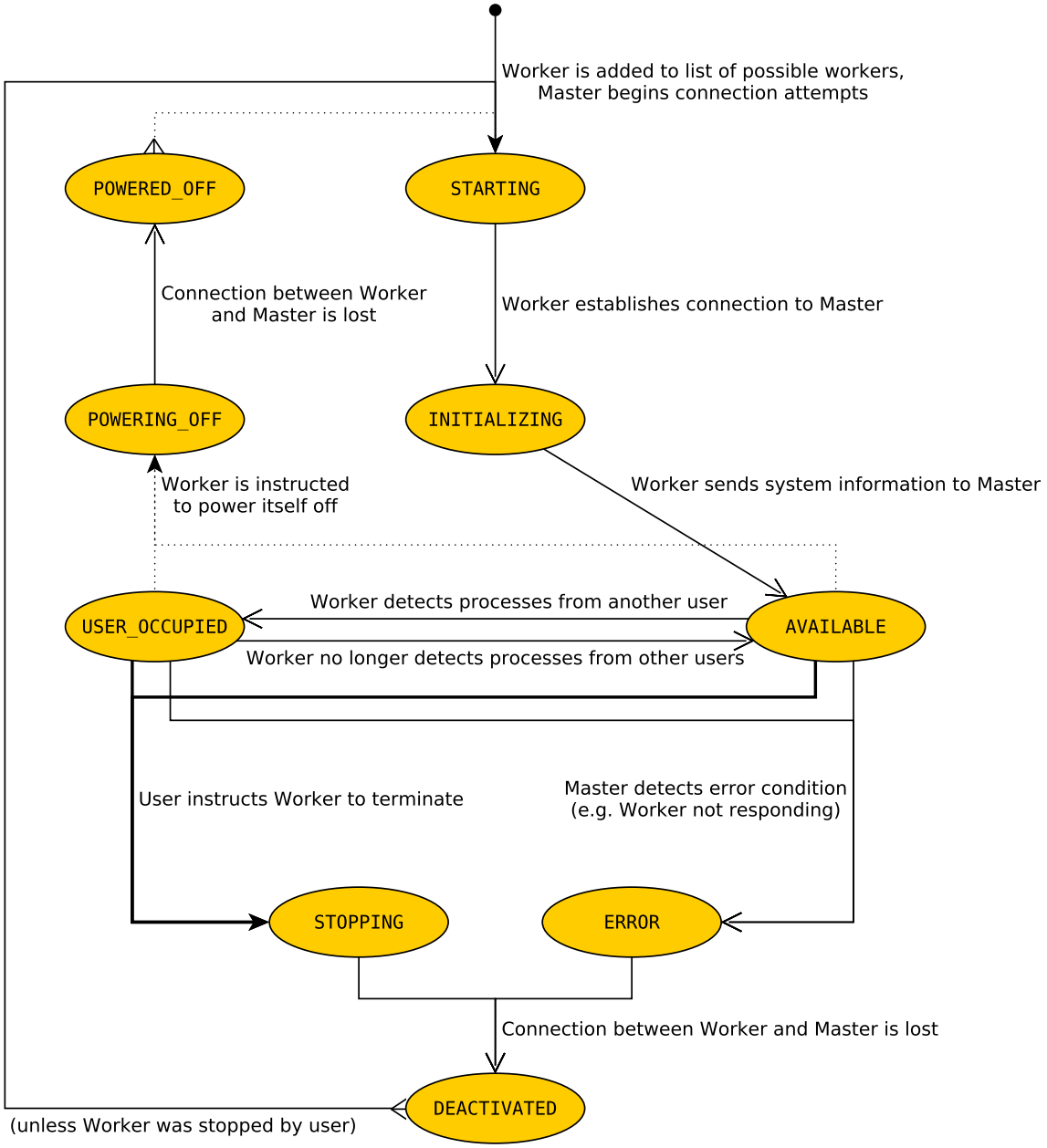

Figure 4.2.: Transitions between the worker states used by the `VerifierCloud` master to track which workers are "`AVAILABLE`" for task execution.
Thick solid edges denote actions explicitly initiated by the user, while thin solid edges denote state transitions triggered automatically when a certain condition is fulfilled. Dotted edges are used for transitions that depend both on a certain user-defined configuration and a condition being matched. The latter is currently the case for the automatic power-down (based on idle time) and power-up (based on resource demand) of workers, as described in section 4.2.

As far as retaining worker information is concerned, it would not strictly be necessary to distinguish between intentional power-downs (initiated by the master) and unintentional connection losses (e.g. network connectivity issues, power outages, or system shutdowns initiated by a user outside the `VerifierCloud` controls). We do, however, require a way to distinguish between these two cases when it comes to checking for suitable workers, as we cannot expect an unintentionally disconnected node to be available at our command (contrary to a node who has performed an intentional, orderly shutdown via the worker that was running on it).

One important step towards automatic power control is therefore to introduce new worker states to mark workers that are currently executing an intentional shutdown (`POWERING_OFF`) and ones that have disconnected after an intentional shutdown and are therefore assumed to be powered down, but available on demand (`POWERED_OFF`). These new states are already included in figure 4.2 and will be further explained in section 4.2.3.

4.2.2. Using Wake-on-LAN to Power up Worker Nodes on Demand

The modification described in the previous section allows the `VerifierCloud` master to consider all cluster nodes when searching for suitable workers, including ones that have been shut down to conserve energy. To complete the power control process, we now require a means to automatically power up nodes when the master determines that they are required to execute runs.

Operational Principles of Wake-on-LAN

A standardized and commonly supported method for this is the Wake-on-LAN (WoL) feature: WoL-capable devices that have their WoL feature enabled listen for a so-called "magic packet" on their network interfaces, containing a specific data payload based on the device's MAC address. When this magic packet is received, the device initiates its normal power-up sequence.

WoL uses MAC addresses (rather than IP addresses) so the magic packets can be processed directly by the network interface card without requiring a TCP/IP protocol stack. This means, however, that the magic packet cannot be sent as a unicast packet addressed to the IP address of the powered-down device, as the network interface has no knowledge of IP addresses and therefore will not send the ARP response necessary to receive a unicast packet. Instead, magic packets are usually sent as broadcast packets to all devices on the local network segment. The intended recipient device will detect the magic packet and initiate a power-up, while all other devices will simply discard the packet. While it is technically possible to send broadcast packets over a routed network like the Internet, the necessary configuration is not universally deployed, so broadcast-dependent features like WoL are usually limited to use within the same local network. In our case, this precludes the use of the automatic power control feature on the nodes hosted remotely by other institutions, but since those nodes are outside our administrative domain, the were not included in our measurement setup anyway.

Aside from the technical requirements at the network level, the master also needs to know a node's MAC address to be able to send a magic packet with the correct contents to "wake" that node. This information can be obtained automatically by any software running on the node, such as our worker instances, so it can simply be included in the map of attributes (CPU model, etc.) that is already collected from each worker during the worker start-up process. Care must be taken when collecting a node's MAC addresses since it can have multiple physical network interfaces, each with a different MAC addresses. As it would require extensive parsing of the node's network configuration to determine which interfaces are reachable from the master, the approach with the least programmatic overhead is to simply list the MAC addresses of all available network interfaces and later send magic packets for all of them. At approximately 100 B per magic packet, the additional network traffic caused by this will usually be negligible compared to the traffic produced by normal `VerifierCloud` operation.

Obtaining the MAC address automatically has the advantage of keeping configuration overhead low, but it means that the master does not have enough information to remotely power up a node until it has successfully started a worker instance on it. The cluster operator therefore has to make sure that nodes are powered up before they will be initially recogized and used by the `VerifierCloud`. On our setup, the `VerifierCloud` master usually runs continuously for several weeks at a time and cluster nodes are very likely to be powered up when the master is restarted, so our use case did not justify the additional complexity of adding MAC addresses to the configuration file format. If the need ever arose, this option could easily be added in the same way as the idle timeout value introduced in section 4.2.3.

Initiating a Node's Power-up Sequence from the Master

With all required information present in each worker's attribute map, the master can now generate magic packets to trigger a node's power-up sequence via WoL.

As mentioned in section 4.2.2, the only requirement for the magic packet is that it must contain a specific payload. The type of packet, as well as any header data prepended to this magic payload, are of no consequence to WoL functionality, so the user is free to choose the specific protocol used for the packet. A common approach is to send a UDP packet to port 9, which has been assigned by the IANA to be used for the Discard Protocol, i.e. to discard all incoming packets at operating-system level [21]. Network cards check for WoL-relevant packet contents before packets reach the operating system, so the Discard Protocol provides a convenient target port for WoL packets.

This approach was also chosen for the `VerifierCloud`, using an open-source implementation by Matt Black [2]. For debugging purposes, a new type of `VerifierCloud` command packet was added, to make the master send a WoL packet by issuing a command on the `VerifierCloud` command-line client.

During manual testing, nodes would sometimes not react when sent a magic packet, likely either due to limitations of the network interface card, or due to congestion and

[2]Wake On Lan: Android wake on lan application, `https://github.com/mafrosis/Wake-On-Lan`

packet loss on the network. While the exact cause is unknown, the stateless and unreliable nature of UDP makes it prudent to re-send WoL packets in any case.

When a node is added to the list of potential workers on the master, the master automatically tries to connect to the node via SSH and launch a worker instance on it. Since this connection attempt might fail due to temporary reasons (network problems, the master is programed to regularly re-attempt a connection to any worker that is not currently connected. At the time of writing, the default interval for this feature is set to 2 min, regulated by a list of upcoming attempt times maintained by the master.

This timer also provides a convenient way to send magic packets repeatedly (to mitigate the reliability problems hinted at above), while minimizing the additional code complexity and resourse usage: the master can simply send a WoL packet before each connection attempt: if the node is already powered on, the packet will simply be ignored and the worker start will proceed normally; if on the other hand the node is currently powered down, the packet will trigger a power-up and the node will likely be available for connections by the time the next attempt is made. Unfortunately, this causes a delay between triggering a power-up sequence and the node actually becoming available for `VerifierCloud` use. This matter requires extensive deliberation and will therefore be elaborated separately in section 4.2.4.

Automating Node Power-up from the VerifierCloud Run Scheduler

To make use of the power-up functionality described above, the master now needs to be modified to account for powered-down nodes when selecting nodes suitable for executing a given run collection.

Previously, the run scheduler would only consider workers in the `AVAILABLE` state (refer to figure 4.1). With the new automatic power-control feature, the scheduler will still perform one worker suitability check using only `AVAILABLE` workers, to avoid powering up any nodes unnecessarily. If that check yields workers that can satisfy the requirements of the current run collections, its run are scheduled and no further action is done. If, however, the requirements cannot be satisfied by the currently `AVAILABLE` workers, the scheduler extends its search to also include `POWERED_OFF` workers, i.e. ones that were running on a node that has since been powered down. If this extended search yields a positive result, the master then proceeds to trigger power-ups of all nodes that have been determined to be suitable, but are currently powered down.

One consequence of this is that such a condition will cause the power-up of all nodes matching the requirements, regardless of the number of runs in the run collection. Theoretically, this may mean powering up more nodes than there are tasks in the run collection, leaving some nodes unused (and thus powered up to no purpose). However, our specialized use case of testing software-verification tools means that run collections will usually be of such large sizes (on the order of 10^2 to 10^4 runs each) that all powered-up nodes will be used. While there are possible improvements that could be made to this approach, they are associated with additional challenges, as outlined in chapter 5.

4.2.3. Automatic Power-down of Idle Workers

While it was possible to implement the node power-up functionality using a widely used remote-control standard, no such standard exists for the purpose of initiating system shutdowns. We therefore need to target platform-specific solutions for the second part of our automation. `VerifierCloud` workers already require the `cgroups` interface that is specific to the Linux operating system, so this will not create any new limitations in terms of operating-system compatibility.

On Unix-based operating systems, such as Linux, powering down or rebooting the system was historically performed by executing the `shutdown` command with parameters selecting the intended action (power off, reboot, etc) and the time when the process should be triggered. However, the exact command-line parameters have changed over time and can have different meaning on different Linux-based operating systems ("Linux distributions") today. Many major distributions [3] have therefore added a `poweroff` command that invokes `shutdown` with the appropriate parameters to power off the system in a safe way (i.e. giving programs a time window to terminate cleanly and write all changed data to persistent storage).

To simply deployment, the `VerifierCloud` worker will directly execute this universally available command to initiate a power off. While it would have been possible to delegate the actual implementation to an external script provided by the system administrator, the `poweroff` command is universally available and already triggers all standard mechanisms to ensure a clean shutdown of the system. Introducing an additional layer of abstraction would therefore add complexitity without providing any obvious additional benefit to flexibility.

There is, however, one indirection that has to be made, since initiating a system power-off requires super-user privileges: rather than run the entire worker process with elevated privileges (which could pose a security risk in case of certain programming errors), privileges can be granted selectively by using the well-established [4] `sudo` command [5]. This requires the system administrator to add the necessary permission grant to the sudo configuration on each worker node, but since `sudo` supports multiple configuration files and even centralized (e.g. LDAP-based) configuration, this change can be easily automated.

Once this configuration is in place, the worker process can initiate a power-off by executing the command line `sudo poweroff` whenever it is instructed to do so by the master. The master in turn keeps track of the current state of a worker (according to the transitions shown in figure 4.2) to be able to distinguish between an intentional shutdown (initiated by the master) and an unexpected disconnection (caused by a local user switching off the node, a network issue, power outage, or other unforeseen circumstances). As previously mentioned, the `POWERING_OFF` and `POWERED_OFF` states have been newly introduced with the implementation of automatic power-off functionality.

To decide when to trigger a node power-off, the `VerifierCloud` needs to keep track

[3] Including Arch Linux, CentOS, Debian, RedHat Enterprise Linux, and Ubuntu

[4] A Brief History of Sudo, `https://www.sudo.ws/sudo/history.html`

[5] Sudo in a Nutshell, `https://www.sudo.ws/intro.html`

of how long each worker has been idle, and issue the power-off command when a certain configured *idle timeout* is exceeded. The accounting for this is ideally implemented within the master instance, since the master also needs to update its list of worker states when a worker is being powered off intentionally (as explained in section 4.2.1).

As noted in section 4.2.2, the master already maintains a list of upcoming event times (internally referred to as a *schedule*) that indicate when the next connection attempt to each unconnected worker should be made. This approach has shown to be reliable, and is therefore also used for the new power-off feature: whenever a worker becomes idle (by finishing processing a run when there are no further runs that can be assigned to it), the master calculates the time when the worker should be powered off — assuming it stays idle — and inserts it into a separate schedule. The schedule is then regularly checked for events that have come due in the same way that is already used for the connection attempt schedule.

The timeout after which an idle worker is instructed to power itself off is freely configurable for each worker when it is added to the master's list of available workers. However, choosing a sensible value for this timeout is a non-trivial task and will therefore be discussed in more detail in section 4.2.4.

4.2.4. Considerations for Choosing Idle-Timeout Values

To make use of the automatic power control feature, the administrator of the `VerifierCloud` master has to configure the amount of time a worker is allowed to be idle before it is instructed to power itself off.

An intuitive approach might be to set this "idle timeout" as short as possible, to reduce the amount of energy wasted by idle worker nodes. However, letting a worker idle is not the only waste of energy that can occur: When a worker is instructed to power off its node, the node's operating system has to ensure that all processes are terminated cleanly before proceeding to power off the system. Likewise, after a node has been issued a power-on signal, it takes time for it to load the operating system and all required programs and services, before it becomes available for network connections and thus usable by the `VerifierCloud`.

No verification runs can be executed during these shutdown and boot periods. For the user, this manifests as a delay in run execution. Also, whenever a node is powered on without perform useful work, the energy it consumes can be considered wasted. This is exacerbated by the fact that shutting down and booting an operating system requires more system resources than are used in the system's idle state, and thus also increases the power consumption of the system.

These considerations show a scenario with two conflicting optimization routes: on the one hand, a short idle timeout would reduce the energy wasted on idling nodes. On the other hand, a longer idle timeout would avoid needlessly expending the additional energy required to shut down and boot a node. To strike a balance between the two ends of the spectrum, the characteristics of the individual group of nodes need to be analyzed.

While the shut-down and boot-up times of different computer systems will be vastly

different, we can use our main computer cluster *zeus* as an example. Through empirical testing, we were able to determine the average duration of a power-off sequence to be 28 s, while powering on a *zeus* node took 59 s on average. Unfortunately, these tests were performed after a number of changes in the organizational structure of our institution, following the move of Prof. Beyer from the University of Passau to the Ludwig Maximilian University of Munich. As part of these changes, administration of the cluster and the associated monitoring systems was transferred, resulting in the loss of the power-consumption measurement data from the relevant time period. It is possible, however, to estimate the cluster's power consumption during shut-down and boot-up sequences from its average and peak consumption, operating under the assumption that both shut-down and boot will require additional power compared to the system's idle state.

At an average power consumption of approximately 2.5 kW for the entire cluster during idle periods and approximately 4.9 kW while in use, we can calculate the amount of idle time that results in the same energy consumption as a power-down and power-up sequence (plus any amount of time in the powered-off state, since no energy is consumed during that time) as follows:

$$t_{idle} \cdot P_{idle} = (t_{shutdown} + t_{boot}) \cdot P_{use}$$

Inserting the measured values from before ($P_{idle} = 2.5\,\mathrm{kW}$, $P_{use} = 4.9\,\mathrm{kW}$, $t_{shutdown} = 28\,\mathrm{s}$, $t_{boot} = 59\,\mathrm{s}$) yields

$$\begin{aligned} t_{idle} \cdot 2.5\,\mathrm{kW} &= (28\,\mathrm{s} + 59\,\mathrm{s}) \cdot 4.9\,\mathrm{kW} \\ t_{idle} &= \frac{(28\,\mathrm{s} + 59\,\mathrm{s}) \cdot 4.9\,\mathrm{kW}}{2.5\,\mathrm{kW}} \\ &= 170\,\mathrm{s} \end{aligned}$$

We can now use this value as a lower bound for the duration a cycle of power-off, off state and power-on would have to last to be efficient in terms of energy consumption: only for durations longer than this would powering off the system yield any energy savings.

Note, however, that t_{idle} also includes the 59 s required to power on the node when the master receives a run collection that requires it. We require the time between the beginning of a power-off sequence and the beginning of the subsequent on-demand power-on, so we we need to subtract t_{boot}:

$$\begin{aligned} t_{unused} &= t_{cycle} - t_{boot} \\ &= 170\,\mathrm{s} - 59\,\mathrm{s} \\ &= 111\,\mathrm{s} \end{aligned}$$

For our use case, this means that the power control automation would have to predict when a worker node was likely to remain unused for at least $t_{unused} = 142\,\mathrm{s}$ before it would be sensible to power it off. Unfortunately, the irregular nature of run-collection

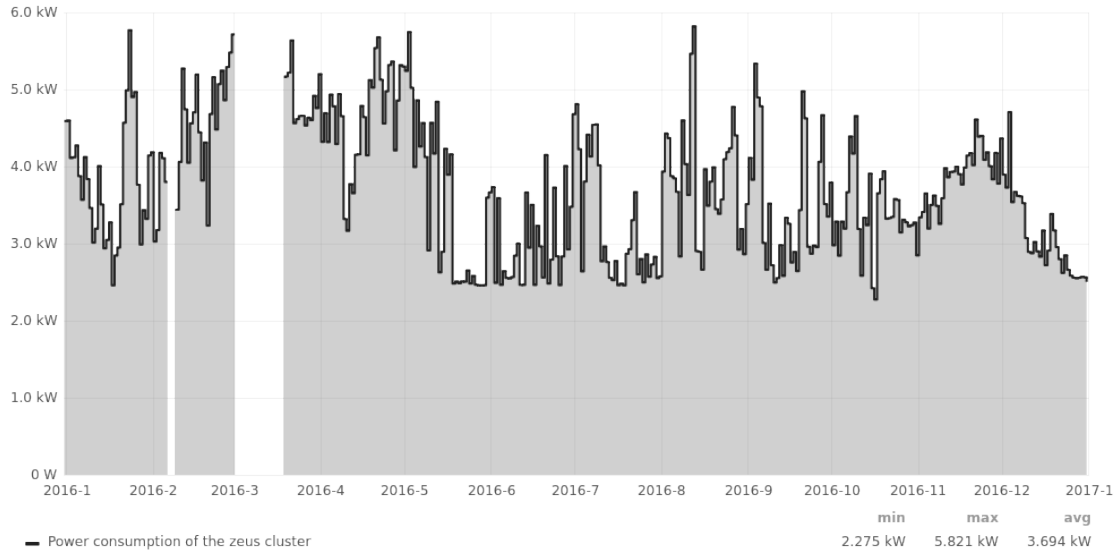

Figure 4.3.: Average daily power consumption of the *zeus* computer cluster from 2016-01-01 to 2016-12-31

submissions makes it very difficult to reliably predict the time of the next submission. Theoretically, it would be possible to analyze historic data on the execution time of run collections to determine the likelihood of another run collection being submitted as a function of the time passed since the last submission. However, examining the historic power consumption data of the *zeus* cluster (shown in figure 4.3) reveals no obvious pattern in the cluster usage frequency and intensity. This can likely be attributed to the human factor already mentioned in the introduction to chapter 4, namely the varying influence on work hours exerted by holiday and project schedules.

In conclusion, optimizing the idle-timeout setting in our use case would require extensive data analysis at a level that would exceed the scope of this thesis. Instead, an administrator may choose a conservative timeout value based on the estimate above, and adjust it manually based on his or her observations and the users' expectations with regard to run execution delays.

For future reference, chapter 5 will list some possible approaches to this kind of data analysis, to be explored in future work on the subject.

4.3. Resulting Savings on Energy Consumption

In addition to the theoretical considerations described in the previous section, the finished power control functionality was tested in a multi-node environment separate from the main cluster. The decision to use a separate testing environment was made to avoid any negative impacts on normal operations. While preferable for operational reasons, this does limit the scope of the test results, especially due to the fact that no nodes with external energy-measurement instrumentation were available for the test. Nonetheless,

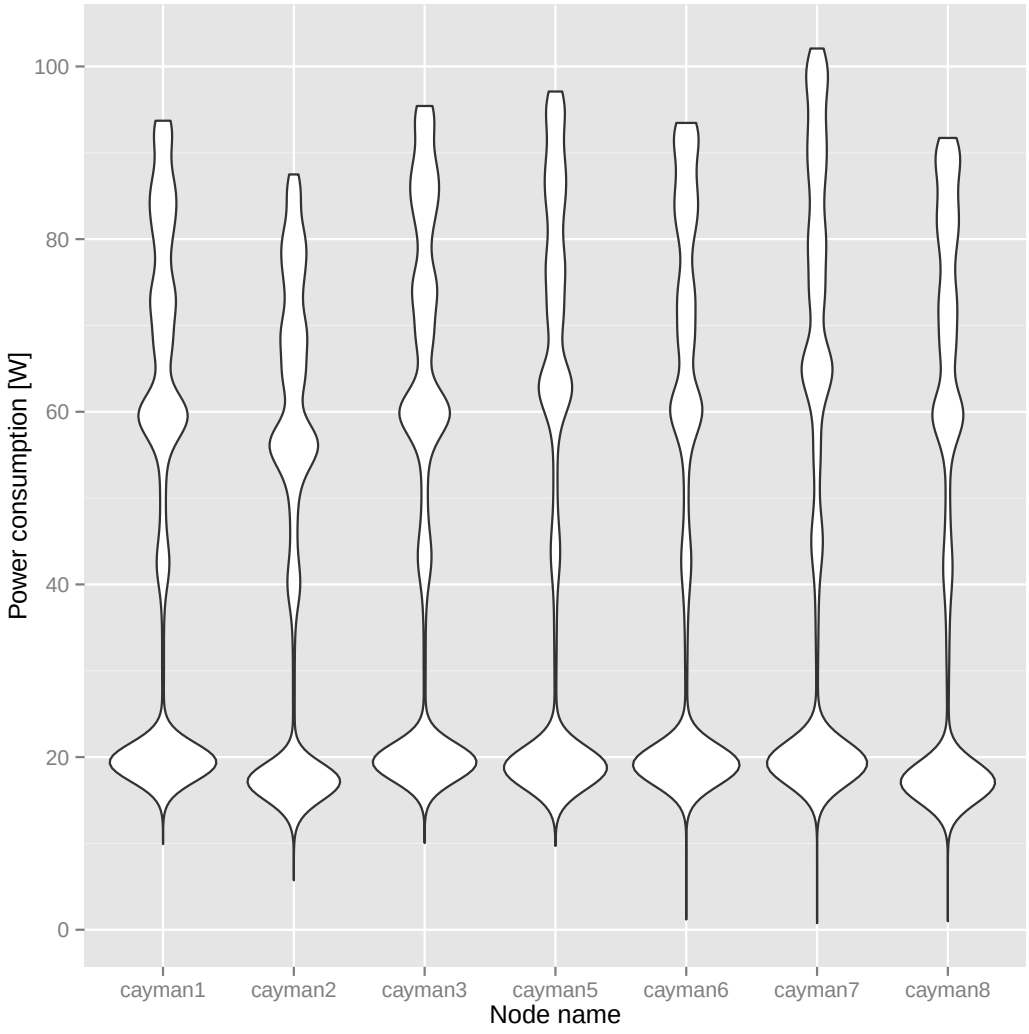

Figure 4.4.: Distribution of momentary power consumption of the *cayman* computer cluster, recorded from 2016-12-25 to 2017-03-23.
cayman4 suffered a malfunction during the measurement period and has been excluded from this figure.

the test was sufficient to demonstrate the full functionality of the new feature and yielded valuable insights into the factors that need to be considered when configuring it (such as the effectiveness threshold described in section 4.2.4).

To estimate the potential savings on energy consumption that could be achieved on our cluster infrastructure, we can instead utilize the power consumption data that has been collected by our monitoring systems. Of particular interest for this purpose is the distribution of the discrete power-consumption values: figures 4.4 and 4.5 show a clear clustering of measurements in the lower power-consumption segment, indicating that the nodes spent a significant portion of their operational time in an idle state, thus yielding low power-consumption measurements.

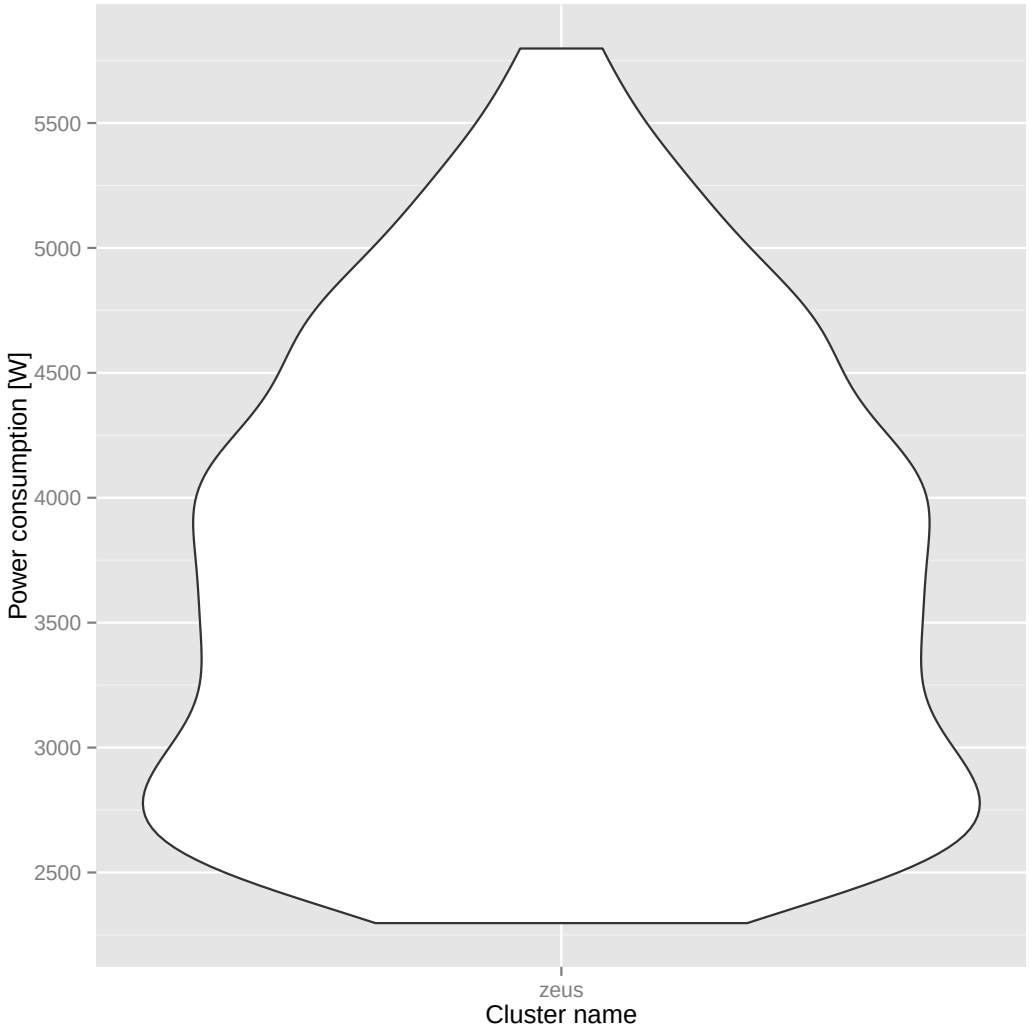

Figure 4.5.: Distribution of the average daily power-consumption values shown in figure 4.3

The effect is particularly pronounced in the data concerning the *cayman* cluster. This is in part due to the different use cases for the *zeus* and *cayman* clusters, with *zeus* being primarily used for large software-verification run collections, while *cayman* is often excluded from these and used for continuous-integration software builds instead (which are only triggered upon changes to the code base, can be completed relatively quickly, and therefore usually do not extend into times of day with little user activity).

Another factor influencing the difference between the two diagrams stems from the means of data acquisition and aggregation: The nodes of the *cayman* cluster are each connected to a single-phase energy meter and monitored by a custom monitoring daemon that retains individual data points for an extended period of time. The *zeus* cluster on the other hand is monitored by a rack PDU that accumulates the power consumption of multiple nodes due to the layout of its electrical distribution. The resulting data

has also been pre-aggregated to daily averages by `Graphite`, so the gap between idle power consumption and consumption under load is less pronounced than it would be in higher-resolution data.

Using these daily data samples obtained from `Graphite`'s database (also shown graphically in figure 4.3, we can extract several useful data points: first and foremost, the *zeus* cluster consumed a total of 31 000 kWh in 2016 alone. Even at the discounted electricity price offered to our institution, this corresponds to 5 800 € for a cluster of 24 nodes over the course of one year. For comparison, the 8-node *cayman* cluster with its average power consumption of approximately 51 W will consume less than 450 kWh per year, bringing it to a mere 1.4 % of *zeus*'s consumption.

Considering its large yearly energy consumption and significant amount of idle time, optimizing the energy consumption of the *zeus* cluster by employing the automatic power-control functionality could yield significant savings on operational cost. To estimate this number, we need to determine the range of power consumption that corresponds to the system's idle state (as represented by the large cluster around 2 800 kWh in figure 4.5). Even with conservative estimation, assuming *zeus*'s maximum power usage in idle to be 2.8 kWh, its idle state accounts for 13 % (3 900 kWh) of its yearly energy consumption, corresponding to 740 € spent on powering an unused system. If we refer to the very pronounced clustering of *cayman*'s power consumption values as an example and assume an idle power usage of 3.25 kWh on *zeus*, the amount of wasted energy — and thus potential savings — is as high as 27 % (8 300 kWh) or 1 600 € per year.

These figures can be taken to indicate that automatic power control, as demonstrated in this thesis, can indeed lead to significant savings on energy consumption, which benefits both the operator (through reduced cost of operation) and the global environment (through reducing the emission of harmful pollutants from electricity production).

5. Caveats and Possible Future Extensions

In summary, the technologies examined in this thesis already provide extensive means of energy measurement, visualization, and optimization. There are, however, some limitations that need to be stressed. Wherever possible, this section will provide suggestions to mitigate them in scenarios where they might pose a significant drawback to the system operator.

Measurement Technologies

In the area of measurement equipment, close attention must be paid to the accuracy of the measurement results that can be obtained from a meter. This includes the accuracy figure itself — ranging from 1 % to 3 % for the devices examined in this thesis, and even including RAPL technology with no guaranteed accuracy at all — but also extends to the conditions that are required for the accuracy specification to hold: many devices, such as the rack PDU described in section 2.2 or the single-phase meters described in section 2.3, require a certain minimum load to measure accurately and might even discard measurements below that threshold. The operator must also take note of what parts of the system's energy consumption are actually measured to avoid incomplete results: while useful through its simplicity, measurements obtained from Intel RAPL do not represent the full impact of a computer on overall energy consumption and, by extension, cost of operation and environmental consequences.

Unfortunately, more flexible and accurate measurement equipment often tends to have more complex installation requirements, such as the setup detailed in section 2.3 that consists of eleven individual devices and required a separate fuse box to be installed by a professional electrical technician.

While it is possible to sidestep this complexity issue by using consumer-grade hardware, those devices often do not offer the standardized interfaces required for reliable data collection.

Setting up energy measurement instrumentation will therefore usually be a trade-off between the three competing factors of cost, complexity, and flexibility.

Data Storage and Visualization Solutions

When it comes to storing and visualizing the acquired measurement data, the user is presented with a large number of alternative tools and solutions, each with different advantages, but also different drawbacks. Due to the large amount of data they are

required to process on a regular basis, monitoring solutions tend to consume significant amounts of computational resources, including CPU usage and disk throughput. The operator must therefore make sure to either use a sufficiently powerful — and perhaps dedicated — system to host these services, or to manually configure them to optimize their resource usage in the particular use case at hand.

Even then, it is difficult to find a single monitoring solution that serves all possible use cases: while `Graphite` was sufficient for everyday monitoring purposes, section 4.3 showed that its automatic aggregation of data points could pose a problem when trying to make decisions based on long-term

In general, one cannot reliably predict the ways in which measurement data might be used in future projects, so it is advisable to store that data at as high a resolution as possible, while accounting for opposing factors such as disk space usage by the time-series database.

Reducing the Energy Consumption of Computer Clusters

While there was no opportunity to test the `VerifierCloud`'s new power-control feature in full-scale productive use, we can already identify some areas that could be targeted for further improvement:

As detailed in section 4.2.4, it is very difficult to choose a idle-timeout value for the power-control feature that maximizes energy savings without hindering the `VerifierCloud`'s speed and effectiveness.

To find such a value, the operator would need to record the sizes of all submitted run collections, as well as their times of submission, and analyze that data to reveal any pattern in run-collection submission times that could be factored into an optimum timeout value. In particularly complex cases, it might even be necessary to replace the static timeout value with a function that determines the timeout value based on other factors, such as the time of day, current project schedules, or historic cluster utilization data. This could pose potential research opportunities for other fields of computer science, such as context recognition and machine learning.

Potential for optimization possibly also exists within the worker power-on process: currently, the master always starts all worker nodes that have been determined to be suitable for a given run collection. In the context of software verification, this is usually appropriate, since verification run collections tends to consist of hundreds, if not thousands, of individual runs, thus providing a sufficient work load for clusters of nearly any size.

If the `VerifierCloud` was selected to be used for other tasks, however, the power-on mechanism should be extended to account for the number of runs within a run collection, to avoid starting more workers than there are runs.

However, starting worker nodes selectively introduces a new issue that would need to be addressed: while it is likely that workers that have been powered off intentionally will be able to be powered on again, this is by no means guaranteed. If the master were to only ever send power-on signals to the first few workers in its worker list, it might get stuck in a scenario where it targets only workers that happen to have suffered a fault

and will not power on. To account for this, the master would have to keep track of the power-on attempts for each worker, and eventually exclude them from the worker list if they were determined to have become unreliable.

The general worker start procedure also poses another limitation in its current implementation: the master attempts to start worker instances at a fixed interval (defaulting to 2 min unless overridden by the operator). These connection attempts happen in quick succession of the transmission of Wake-on-LAN signals, so the worker node will usually still be in the process of powering on during the first connection attempts. After that, the next connection attempt only happens after a full two minutes have passed, even when the node only requires a much shorter time to boot up (compare the empirical value of 59 s from section 4.2.4). This results in an unnecessary idle period that increases both the waiting time incurred by users, as well as the energy consumption of the cluster.

One possible approach to mitigating this could be a "back-off" feature that starts with short intervals between connection attempts and continually increases them after each failed connection attempt. This would prevent excessive load on the network from too many connection attempts, while still reducing the delay between powering on a node and starting a worker instance on it.

With these optimizations, the automatic power-control feature could arguably result in better energy efficiency than it provides in its current form.

It has to be noted, however, that automatic power-control cannot be applied to all kinds of computer clusters. Many data centers mainly operate servers that are required to be online and running continuously, since they are directly accessed by end-users, rather than managed from an integrated task-queuing framework like the `VerifierCloud`.

These computer systems contribute a significant portion to the world's overall electrical energy consumption, calculated to be as high as 4.6 % in 2012 and rising by 4.4 % each year [13].

Since they cannot simply be powered off when idle, other means of reducing their energy consumption need to be explored. An obvious area of improvement is the efficiency of their cooling systems: for the past years, cooling has consistently accounted for about 60 % of data centers' total energy consumption. This distribution can be attributed to the use of localized cooling solutions over centralized ones, as is still common in many devices: localized cooling increases overhead due to the use of multiple small cooling devices and additionally tends to encourage designs that do not pay attention to optimized air or coolant flow through the system. While out of scope for this particular thesis, investigating optimized cooling techniques could provide valuable insights into further potential energy savings.

Irrespective of such hardware-based optimizations, however, the software-based approach presented in section 4.2 can already contribute significant savings on energy consumption and operators should evaluate whether it can be applied to their use case.

6. Conclusion

This thesis has aimed to provide the reader with insights into the primary challenges associated with measuring the energy consumption of computer clusters and utilizing that data to plan, apply and validate optimizations.

The energy consumption of computer infrastructure has become an increasing focus of research due to its extensive effects: constituting 4.6 % of worldwide electricity consumption as of 2012 and rising by 4.4 % each year [13], information and communication technology is a significant contributor to the negative environmental effects of the global production of electrical energy. It is therefore an important goal to minimize this figure by avoiding any unnecessary consumption of electrical energy. This in turn requires means of measuring a system's energy consumption to analyze the possibilities for optimization and later confirm and quantify the results of those optimizations.

For the first step of acquiring measurements, available options range from integrated solutions such as Intel RAPL (section 2.1) that offer minimal installation complexity, to single devices with a variety of features (such as the rack PDU from section 2.2 and also some consumer-grade devices described in section 2.5), to highly customizable instrumentation that provides the operator with highly accurate and fine-grained observations of the individual nodes in a cluster (as implemented using single-phase energy meters in section 2.3). These options have been compared in detail in chapter 2 and have been successfully employed in the hardware infrastructure at our institution.

Continuing the processing chain, we then evaluated the advantages of several monitoring software solutions, spanning multiple different approaches to acquiring, storing, and processing data. A focus was put on well-established solutions such as `MRTG` (section 3.2.1), `Munin` (section 3.2.4), and `Graphite` (section 3.2.6) that are based on fixed-size database formats, such a RRD and `carbon`, to store time-series data from a large number of sources. The scope was also extended to the more recent development of tool stacks like `TICK` that substitute fixed database sizes with other database formats, allowing for unrestricted data resolution. Ultimately, both approaches still have use cases today and operators can freely choose the tools that best suit their needs, referring to the information in chapter 3 as a guideline.

Most of the listed monitoring solutions provide means of extending them with custom data sources. This functionality was used to integrate them with our energy measurement instrumentation and the resulting software has been made available in appendix A.

Several of the presented data storage and visualization tools have been used in our environment, yielding insights into their usability and their behavior under the load of real-world usage. Based on these insights, a suitable tool stack for our use case has been chosen and documented in section 3.3. Consisting of multiple tools, our monitoring solution combines the efficiency, extensibility, and wealth of plugins of `collectd`

with the advanced post-processing features offered by `Graphite` and the powerful and user-friendly interface of `Grafana`. It also includes alerting functionality based on the measurement data using the `Bosun` alerting tool, making it a very useful tool for researchers and system administrator alike.

Finally, with appropriate energy measuring and monitoring solutions in place, we proceeded to devise and implement a means of reducing the energy consumption of our software-verification computer cluster, as documented in chapter 4: the master and worker implementations of the `VerifierCloud` task-queueing framework have been extended to allow worker nodes to be powered off and powered on remotely, either manually by the user or on demand using an automated feature. This new power-control feature was programmed to continually monitor the utilization of its worker nodes and — depending on user configuration — power them off automatically when they are not in use. When queued run collections require the computational resources of powered-off nodes, the power-control functionality on the master then proceeds to power them back on automatically. The feature was successfully tested in an environment similar to that of the main cluster, making it ready to be deployed and put to use in a real-world scenario. We also identified that could be used as starting points for potential future extensions.

Using the data from our measurements, this automatic power-control feature has been confirmed to provide significant savings on the cluster's energy consumption, reaching up to 27 % of its total consumption and potentially saving more than 1 600 € in operational costs every year for a single cluster. In addition to saving costs, the energy consumption optimizations that have been presented also reduce the negative effects that electricity production still has on the global environment.

Overall, the author is confident that this thesis can serve as a toolkit for researchers striving to gain a better understanding of their hardware and its energy-consumption properties, and to find new ways of improving the efficiency of their setup.

Acknowledgments

The author would like to give particular thanks to Ludwig Zistler of the University of Passau's technical operations department for his valuable input regarding the University's power consumption, as well as possible optimizations to cooling technology for data centers.

A. Software Developed in the Context of this Thesis

The following software has been developed to complement the work described in this thesis. The relevant source code is available in the digital distribution for this thesis.

Details on the required hardware setup can be found in chapter 2.

A.1. Intel RAPL Implementations

A.1.1. SNMP Agent Interface

`power_gadget_snmp` is based on `Power Gadget`, Intel's reference implementation of an interface to the RAPL energy measurement functionality of modern Intel "Core" and "Xeon" CPUs.

It implements the `pass_persist` interface used by the `Net-SNMP` SNMP agent to expose the RAPL measurement values (converted to Joules) via an SNMP subtree rooted at the OID `.1.3.6.1.4.1.47670.5`.
Its OIDs follow the format `.1.3.6.1.4.1.47670.5.<cpu_number>.<rapl_domain>`, where `cpu_number` is the one-based CPU number returned by the RAPL interface, and `rapl_domain` is either `1`, `2`, `3`, or `4`, signifying the `package`, `core`, `uncore`, and `dram` RAPL domains, respectively.

Written in C, it can be compiled using the supplied Makefile.
Note that the RAPL interface requires the `msr` and `cpuid` kernel modules to be loaded.

A.1.2. Collectd Integration

`collectd-plugin-intel_cpu_energy` is a native plugin for the `collectd` "system statistics collection daemon".

It uses the same reference implementation as `power_gadget_snmp` to access the RAPL measurement values, converts them to Joules, then passes them on to `collectd` for further processing. Its output follows the recommended `collectd` plugin structure, with the following mapping:

- `hostname = <hostname provided by collectd>`
- `plugin = intel_cpu_energy`
- `plugin_instance = cpu<N>` (`N` ≥ 0)

- `type = energy`
- `type_instance = [ package | core | uncore | dram ]`

Therefore, the resulting query path will look similar to `zeus01.intel_cpu_energy-cpu0.energy-package`.

Written in C, it can be compiled using the supplied Makefile.
Note that the RAPL interface requires the `msr` and `cpuid` kernel modules to be loaded.

A.2. Reading Measurements from Single-phase Energy Meters

The data acquisition and processing for the single-face energy meters is split into multiple programs for increased flexibility.

A.2.1. Data Collection Daemon

A central daemon — `emeterd` — collects and counts pulses from the energy meters and stores the results in realtime.

Two storage formats are used: first, all events (both pulses and starting or stopping the daemon) are recorded to `raw.log` as plain-text lines. Pulses are logged in the format `<date>,<time>,event,[ start | stop ]`, while pulses are logged as `<date>,<time>,<input pin number>,0`. Additionally, the daemon maintains separate text files — one for each configured input pin — that contain a monotonically increasing value signifying the number of pulses received on that pin so far. The values of these counters are preserved across restarts of the daemon, or indeed the system it runs on.

Refer to `pins.txt` for the mapping between GPIO pin numbers, meter numbers, and node names in our setup.

The `emeterd.py` executable file includes a standardized configuration block to be parsed by SysV-style init systems, so it can be directly installed as a system service. A `logrotate` configuration file to regularly `raw.log` is also included in the distribution.

A.2.2. Command-line Interface

`emeter-live` is a command-line utility that reads lines from `raw.log` (as written by `emeterd`), converts the timestamps and recorded pulses within into the most recently observed power consumption of each connected node, and displays that information both numerically and as a set of bar graphs.

`emeter-live.py` expects continous input on its standard input stream, so `emeter-live.sh` is provided as a wrapper to automatically supply the required data.

A.2.3. Websocket-based Interface

`emeter-ws` operates in a manner similar to `emeter-live` in that it parses lines from `raw.log` to calculate current power consumption.

In this instance, however, the process is split into two parts: a daemon serves a web page via HTTP on port 9000 and simultaneously detects and broadcasts any changes to `raw.log` to all clients currently connected via the Websocket protocol. These broadcasts are received by a JavaScript function integrated into the web page, which then calculates the current power consumption and displays it to the user (again both numerically and as bar graphs).

Additionally, the JavaScript function regularly checks whether the time passed since the last pulse on a certain pin is still consistent with the power consumption reported for that pin. When the time period exceeds the one previously observed (equating to a lower power consumption than before), it begins extrapolating the current consumption value based on the time period since the last pulse. As soon as the next pulse is received on that pin, the estimate is replaced with a newly calculated exact value and normal operation resumes.

A SysV-compatible service file is provided in `emeter-ws`.

A.2.4. SNMP Agent Interface

Finally, the distribution contains two Python scripts that implement the `Net-SNMP` SNMP agent's `pass` and `pass_persist` interfaces, respectively. As explained in more detail by the `snmpd.conf` manual page, the `pass_persist` interface has the advantage of being more resource-efficient, since it re-uses the same interface process repeatedly, instead of launching a process for each individual SNMP query.

Both scripts use an SNMP subtree under the OID `.1.3.6.1.4.1.47670.2` to provide the current contents of the counter files written by `emeterd`. The OIDs follow the format `.1.3.6.1.4.1.47670.2.<meter_number>`, using the meter numbering described in `pins.txt`.

A.3. SML Electricity Meter Interface

Both implementations of the SML electricity meter interface are based on the `jSML` Java library [1] developed by the department "Intersectoral Energy Systems and Grid Integration" at the Fraunhofer Institute for Solar Energy Systems in Freiburg.

`jSML` supports a number of features, including both encoding and decoding SML message files. The following two programs use its decoding functionality and format the results according to their intended use case.

[1] jSML Overview – openmuc.org, `https://www.openmuc.org/sml/`

A.3.1. Command-line Interface

To facilitate debugging the optical interface on SML-capable electricity meters, a `jSML` example program was modified to receive SML messages via the serial interface of a Raspberry Pi single-board computer and format it in a human-readable way.

A.3.2. Collectd Integration

Expanding upon the `jSML`-based program modified for human-readable output, `collectd-plugin-sml-electricity-meter` integrates with `collectd`'s Java plugin interface and submits all relevant SML message contents to `collectd` for further processing. The fields in `collectd`'s data packets are set as follows:

- `hostname = <set at compile-time>`
- `plugin = smlreceiver`
- `plugin_instance = <SML server ID>`
- `type = [ watts | watt_hours ]`
- `type_instance = <OBIS code>`

Refer to section 2.4 and the glossary for details on the use of OBIS codes.

A.4. VerifierCloud Automatic Power Control

The `VerifierCloud` is a sophisticated system for queueing and distributing tasks — primarily verification runs — to nodes of a computer cluster.

The automatic power-off and power-on functionality described in this thesis was implemented in a separate feature branch "automatic-shutdown" so it could be tested without impacting productive use on the main cluster.

All information required for powering worker nodes off and on is collected automatically. However, initiating a power-off sequence requires the worker process to have elevated privileges on the node it is running on. For this, the node's `sudoers` configuration needs to be amended to allow the worker process to execute the command `sudo poweroff` non-interactively (i.e. without entering a password).

After privilege escalation has been configured on the node, the user can then use the `VerifierCloud` command-line client to manually initiate a power-off of that node, as well as a subsequent power-on.

To enable the automatic power-off functionality, the worker needs to be started with a non-empty "shutdown-delay" value, either via the command-line client or from the master's `WorkerInformation` file. The configuration syntax is unchanged and described in more detail in the official `VerifierCloud` documentation.

Bibliography

[1] D. Beyer. "Software Verification with Validation of Results (Report on SV-COMP 2017)". In: *Proc. TACAS*. LNCS 10206. Springer, 2017, pp. 331–349.

[2] D. Beyer, S. Löwe, and P. Wendler. "Benchmarking and Resource Measurement". In: *Model Checking Software - 22nd International Symposium, SPIN 2015, Stellenbosch, South Africa, August 24-26, 2015, Proceedings*. 2015, pp. 160–178.

[3] G. Calandrini, A. Gardel, I. Bravo, P. Revenga, J. L. Lázaro, and F. J. Toledo-Moreo. "Power Measurement Methods for Energy Efficient Applications". In: *Sensors* 13.6 (2013), pp. 7786–7796.

[4] J. Case, M. Fedor, M. Schoffstall, and J. Davin. *Simple Network Management Protocol (SNMP)*. RFC 1157 (Historic). RFC. Fremont, CA, USA: RFC Editor, May 1990.

[5] S. Desrochers, C. Paradis, and V. M. Weaver. "A Validation of DRAM RAPL Power Measurements". In: *Proceedings of the Second International Symposium on Memory Systems*. MEMSYS '16. Alexandria, VA, USA: ACM, 2016, pp. 455–470.

[6] *Digital Enhanced Cordless Telecommunications (DECT); Common Interface (CI); Part 1: Overview*. ETSI EN 300 175-1. European Standard. Sept. 2005.

[7] *Electricity metering - Data exchange for meter reading, tariff and load control - Part 61: Object identification system (OBIS)*. IEC 62056-61:2002. International Standard. Feb. 2002.

[8] *Electricity metering data exchange - The DLMS/COSEM suite - Part 6-1: Object Identification System (OBIS)*. IEC 62056-6-1:2015. International Standard. Nov. 2015.

[9] *Electricity metering equipment (a.c.) - Particular requirements - Part 31: Pulse output devices for electromechanical and electronic meters (two wires only)*. IEC 62053-31:1998. International Standard. Jan. 1998.

[10] *Electricity metering equipment (a.c.). Particular requirements. Static meters for active energy (class indexes A, B and C)*. EN 50470-3:2006. European Standard. Dec. 2006.

[11] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616 (Draft Standard). RFC. Obsoleted by RFCs 7230, 7231, 7232, 7233, 7234, 7235, updated by RFCs 2817, 5785, 6266, 6585. Fremont, CA, USA: RFC Editor, June 1999.

[12] M. Hähnel, B. Döbel, M. Völp, and H. Härtig. “Measuring energy consumption for short code paths using RAPL”. In: *SIGMETRICS Performance Evaluation Review* 40.3 (2012), pp. 13–17.

[13] W. V. Heddeghem, S. Lambert, B. Lannoo, D. Colle, M. Pickavet, and P. Demeester. “Trends in worldwide ICT electricity consumption from 2007 to 2012”. In: *Computer Communications* 50 (2014), pp. 64–76.

[14] *Information technology – Procedures for the operation of object identifier registration authorities: General procedures and top arcs of the international object identifier tree*. ITU-T X.660 (07/2011). ITU-T Recommendation. July 2011.

[15] *Information technology – Procedures for the operation of object identifier registration authorities: General procedures and top arcs of the international object identifier tree*. ISO/IEC 9834-1:2012. International Standard. May 2012.

[16] *Key World Energy Statistics 2016*. International Energy Agency (IEA), 2016.

[17] *Message Queuing Telemetry Transport (MQTT) v3.1.1*. ISO/IEC 20922:2016. Standard. June 2016.

[18] S. Ott. *VerifierCloud: Implementierung eines Web-Service zur Software-Verikation*. Bachelor’s thesis. 2014.

[19] *Plugs and socket-outlets for domestic and similar general use standardized in member countries of IEC*. IEC 60083:2015. International Standard. Oct. 2015.

[20] *Plugs, socket-outlets and couplers for industrial purposes*. IEC 60309:1999. International Standard. Feb. 1999.

[21] J. Postel. *Discard Protocol*. RFC 863 (Internet Standard). RFC. Fremont, CA, USA: RFC Editor, May 1983.

[22] D. Robinson and K. Coar. *The Common Gateway Interface (CGI) Version 1.1*. RFC 3875 (Informational). RFC. Fremont, CA, USA: RFC Editor, Oct. 2004.

[23] J. Sermersheim. *Lightweight Directory Access Protocol (LDAP): The Protocol*. RFC 4511 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, June 2006.

[24] *SML – Smart Message Language*. BSI TR-03109-1, Anlage IV, Teil b. Technische Richtlinie (Technical Guideline). Mar. 2013.

Glossary

Address Resolution Protocol

Link-layer network protocol to determine which MAC address a given IP address is associated with by broadcasting an ARP query and waiting for a response from the intended destination systems. 57

Advanced Message Queuing Protocol

"An open standard for passing business messages between applications or organizations"[2]. Can be used by `collectd` (described in section 3.2.5) as a transport protocol for measurement data. 24, 57, *see also* Message Queue Telemetry Transport

American Standard Code for Information Interchange

Character-encoding standard that uses 7-bit words to encode characters (including the English alphabet, punctuation, and certain control codes). This set of characters is still present at the same code points in most character encodings currently in use on the Internet. 57

AMQP

see Advanced Message Queuing Protocol 24

API

see Application Programming Interface 26

Application Programming Interface

A well-defined set of methods that allow external programs to interact with the program providing the interface. 57

ARP

see Address Resolution Protocol 36

ASCII

see American Standard Code for Information Interchange 24

[2]AMQP is the Internet Protocol for Business Messaging, `https://www.amqp.org/about/what`

Blade server

Server with a compact form-factor, multiple of which are combined in a blade enclosure to be mounted in a rack. 6

Central processing unit

The electronic circuitry inside a computer that executes instructions according to its programming. 58

CGI

see Common Gateway Interface 21, 22

Cgroups

Official abbreviation of "control groups". A feature included in the Linux kernel since 2008 that provides a unified interface to isolate processes and both monitor and limit their use of system resources, such as CPU time, disk throughput, or system memory. 32, 33, 39

Comma-separated values

A file format to store tabular data as plain text, with the data records (columns) separated by commas and rows separated by line breaks. 58

Common Gateway Interface

"A simple interface for running external programs [...] under an information server", such as a HTTP web server. While not formally standardized, a description of the "current best practice" for its use does exist [22]. 21, 58

CPU

see Central processing unit 5, 7–9, 21, 28, 30, 47, 51, 64

CSV

see Comma-separated values 23

Daemon

A program designed to be run without continuous user input [3], such as a web or mail server. 9, 44, 52, 53

DECT

see Digital Enhanced Cordless Telecommunications 16

[3] FreeBSD Basics: Processes and Daemons, `https://www.freebsd.org/doc/handbook/basics-processes.html`

Digital Enhanced Cordless Telecommunications

A telecommunications standard designed to "provide cordless communications, both for voice traffic and for data traffic" [6], primarily for use with cordless telephone handsets and other domestic equipment. Originally developed by the European Telecommunications Standards Institute (ETSI), "[DECT] has also been selected by the ITU as one of the radio interfaces for 'International Mobile Telecommunications 2000'" [6]. 58

DIN-rail

Refers to several types of standardized metal rail profiles, used for mounting equipment (such as circuit breakers) inside equipment racks and fuse boxes. The name derives from the original standardization by the German standardization organization Deutsches Institut für Normung e.V. (DIN). The type most commonly used in fuse boxes has a hat-shaped cross section and is therefore also referred to as "top hat rail" or "Hutschiene" in German. 10, 11

Formal software verification

The process of proving or disproving that a given computer program satisfies a given specification. 30

General-Purpose Input/Output

A "pin" (electric connection point) on an integrated circuit that can be configured in software to act as either an input or output for electric signals. GPIO pins often also have additional features associated with them, such as internal software-configurable pull-up or pull-down resistors, or capabilities to trigger a software interrupt when a certain state or change of signal level is detected. Application Note 11496 by NXP Semiconductors, albeit not related to our hardware setup, provides a generic overview of the electrical characteristics of GPIO pins on modern integrated circuits: `https://www.nxp.com/documents/application_note/AN11496.pdf` 59

GPIO

see General-Purpose Input/Output 12, 52

HTTP

see Hypertext Transfer Protocol 24, 53, 58

Hypertext Transfer Protocol

Application-level protocol for exchanging hypertext — such as HTML web pages — and other data over a computer network. HTTP has been in use by the World-Wide Web global information initiative since 1990 [11]. 59

IANA

see Internet Assigned Numbers Authority 37

IEC

see International Electrotechnical Commission 10

Integrated Services Digital Network

A set of link-layer communication standards used on the public telephone network. 60

International Electrotechnical Commission

The international standards and conformity assessment body for all fields of electrotechnology. [4] 60

International Organization for Standardization

International non-governmental organization connecting 163 national standards bodies. [5] 61

Internet Assigned Numbers Authority

Organization coordinating global Internet protocol resources, such as IP addresses and port numbers. [6] 60

Interrupt

A signal sent by a hardware or software component to cause the computer system to temporarily [7] interrupt normal program flow and execute the matching interrupt service routine (ISR) as soon as possible. 12, 59, 60

Interrupt service routine

A software routine registered to run when a specific interrupt is triggered. 60, 61

IP (Internet Protocol) address

Numeric identifier assigned to a network device either locally or by a central management instance and used to specify the destination of network-layer packets (such as UDP datagrams). 36, 57, 60

ISDN

see Integrated Services Digital Network 11

---

[4] About the IEC, `http://www.iec.ch/about/`

[5] ISO: About us, `https://www.iso.org/about-us.html`

[6] IANA – About us, `https://www.iana.org/about`

[7] The system normally continues execution of the interrupted program at the same point after the ISR has been executed.

ISO

see International Organization for Standardization 61

ISR

see Interrupt service routine 60

JavaScript Object Notation

A textual representation of structured data; derived from JavaScript, but actually language-agnostic and implemented by libraries in many different programming languages. 61

JSON

see JavaScript Object Notation 26

LDAP

see Lightweight Directory Access Protocol 39, 62

Lightweight Directory Access Protocol

A network protocol that "provides access to distributed directory services" [23]. Commonly used as a standardized interface to connect applications to a central user database for authentication purposes. 61

Logging

Recording information (such as measurement values or diagnostic messages) and storing it persistently. The resulting storage file is usually called a "logfile". 23, 28

MAC (Media Access Control) address

Unique numeric identifier of a network interface card, used as the link-layer address to specify the destination of packets on the network. 36, 37, 57

Magic packet

see Wake-on-LAN

Message Queue Telemetry Transport

An ISO-standardized messaging protocol designed to efficiently transmit sensor data over a computer network [17]. 24, 62, *see also* Advanced Message Queuing Protocol

Model-specific register

Sometimes also called "machine-specific register". A register (internal memory location) that allows accessing "experimental" features in some Intel CPUs. 8, 62

MQTT

see Message Queue Telemetry Transport 24

MRTG

see Multi Router Traffic Grapher 20

MSR

see Model-specific register 8, 9

Multi Router Traffic Grapher

Software for collecting and storing computer performance data. See section 3.2.1. 62

Network File System

Distributed file system that uses UDP to serve files from a central server to clients over a computer network. 31, 62

NFS

see Network File System 31

Node

One computer in a network, set, or cluster of computers. 5, 10, 12, 18, 30, 31, 52

OBIS

see Object Identification System 14, 54

Object Identification System

Specifies identification codes for "commonly used data items in electricity metering equipment" [8], including both measurement and configuration values. These identification codes mark both the data type and meaning of a data item. 14, 62

Object identifier

Standardized system for uniquely identifying objects or concepts via a series of integers that represent a node in the International Object Identifier Tree [14, 15]. Administration of the tree is structured hierarchically, so organizations that have been assigned a node in the tree are free to generate OIDs under their branch as they see fit. OIDs are used in computer protocols such as SNMP and LDAP, but also outside the computing field in areas such as health services[8]. 62, 64

OID

see Object identifier 22, 51, 53, 62

[8]PHIN Vocabulary Access and Distribution System, Centers for Disease Control and Prevention, `https://www.cdc.gov/phin/tools/PHINvads/index.html`

PDU

see Power distribution unit 9–11, 13, 15, 44, 46, 49

Phase

In the context of electrical power distribution using alternating current, one of the energized electrical conductors. Power distribution can either utilize a single phase (with a voltage potential to a single conductor called "neutral"), or multiple phases that reach their peak voltages sequentially at different times (and therefore have a voltage potential both to the neutral conductor, and to each other). 10

Pin

see General-Purpose Input/Output

Plug and play

Describes devices that can be begin operation without requiring explicit configuration input from the user, i.e. by simply "plugging in" the device. While technically a hardware-specific term, it is also used for software that satisfies the same requirement (i.e. does not necessarily need to be configured by the user). 15, 22

PNG

see Portable Network Graphics 20, 26

Portable Network Graphics

A lossless raster graphics format. 63

Power distribution unit

A device with one or more electrical input connectors, and multiple electrical output connectors. The term is usually used for the more advanced members of the group (such as the one described in section 2.2), although a domestic power strip is technically also a power distribution unit. 9, 63

Rack

Usually refers to 19-inch equipment racks, which are commonly used for mounting servers, or other electronic equipment. 6, 9, 11, 13, 15, 46, 49, 58

RAPL

see Running Average Power Limit 7–9, 46, 49, 51

Recursion

see Recursion

Round Robin Database

A fixed-size textual file format, used by the eponymous `RRDtool` (see section 3.2.2) for storing time-series data. 20, 64

RRD

see Round Robin Database 20–23, 28, 49

Running Average Power Limit

An integrated capability for measuring and controlling the power consumption of modern Intel "Core" and "Xeon" CPUs (available since the Sandy Bridge microarchitecture generation). See section 2.1. 7, 63

S0 interface

Pronounced "S-zero interface". A two-wire electrical interface for transmitting measurement and control data. 11, 14, 16

SBC

see Single-board computer 12, 13, 15, 54

Secure Shell

Protocol to establish a secure data connection over an insecure network. Originally targeted at providing shell access to remote systems, the SSH protocol has been extended with additional features, such as supporting file transfers and forwarding of TCP ports over an SSH connection. 65

Simple Network Management Protocol

Standard protocol for communicating "management information" between network devices ("agents") and a controlling device. Used for both monitoring and controlling device parameters, which are uniquely identified by object identifiers (OIDs) [4]. 20, 65

Single-board computer

A computer that contains all required components (CPU, memory, etc.) on a single circuit board. This design has some advantages over the traditional modular computer construction, particularly in terms of size and cost. 12, 15, 54, 64

Single-phase

10, 12, *see* phase

Smart Message Language

“A communication protocol for applications in the field of data acquisition and device parameterization” [24], such as on electronic electricity, gas, heat, or water meters. 14, 64

SML

see Smart Message Language 14–16, 53, 54

SNMP

see Simple Network Management Protocol 9, 13, 16, 20–22, 28, 29, 51, 53, 62

Solid-state drive

A storage device that uses direct electronic means to store data, rather than the rotating magnetized disks used in traditional hard-disk drives (HDDs). Due to the absence of moving mechanical parts, SSDs can often exceed the performance of HDDs in terms of access latency and data throughput. 25, 65

SQL

see Structured Query Language 22, 27

SSD

see Solid-state drive 65

SSH

see Secure Shell 33

Structured Query Language

A domain-specific language for querying and manipulating the contents of a database. 65

Three-phase

15, *see* phase

UDP

see User Datagram Protocol 37, 38, 60, 62

User Datagram Protocol

Stateless network-layer protocol. Unlike the Transmission Control Protocol (TCP), UDP has no notion of connections and does not guarantee the order in which packets arrive at the destination, or indeed that they will arrive at all. 65

Wake-on-LAN

Standardized protocol to remotely trigger a system's power-on sequence by sending a "magic packet" with specific contents (as defined by the standard) to the system's network interface. 34, 36, 48, 65

WoL

see Wake-on-LAN 34, 36–38, 48

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Zuhilfenahme der ausgewiesenen Quellen und Hilfsmittel angefertigt habe. Sämtliche Stellen der Arbeit, die im Wortlaut oder dem Sinn nach anderen gedruckten oder im Internet verfügbaren Werken entnommen sind, habe ich durch genaue Quellenangaben kenntlich gemacht. Die Arbeit wurde in gleicher oder anderer Form noch keiner anderen Prüfungsbehörde vorgelegt.

Nils Steinger, Passau, 2017-06-16