Hindawi
Wireless Communications and Mobile Computing
Volume 2019, Article ID 4260359, 15 pages
https://doi.org/10.1155/2019/4260359

Research Article

SensIoT: An Extensible and General Internet of Things Monitoring Framework

Marcel Großmann,[1] **Steffen Illig**,[2] **and Cornelius L. Matějka**[3]

[1] *Computer Networks Group, University of Bamberg, An der Weberei 5, Bamberg 96052, Germany*
[2] *University Library of Bamberg, University of Bamberg, Feldkirchenstraße 21, Bamberg 96052, Germany*
[3] *Computing Centre, University of Bamberg, Feldkirchenstraße 21, Bamberg 96052, Germany*

Correspondence should be addressed to Marcel Großmann; marcel.grossmann@uni-bamberg.de

Received 2 August 2018; Revised 2 January 2019; Accepted 5 February 2019; Published 7 March 2019

Guest Editor: Bing Wang

Copyright © 2019 Marcel Großmann et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

SensIoT is an open-source sensor monitoring framework for the Internet of Things, which utilizes proven technologies to enable easy deployment and maintenance while staying flexible and scalable. It closes the gap between highly specialized and, therefore, inflexible sensor monitoring solutions, which are only adjusted to a specific context, and the development of every other solution from scratch. Our framework fits a variety of use cases by providing an easy to set up, extensible, and affordable solution. The development is based on our former published framework MonTreAL, whose goal is to offer an environmental monitoring solution for libraries to guarantee cultural heritage to be conserved and prevented from serious damage, for example, from mold formation in closed stocks. It is a solution with virtualized microservices delivered by a famous container technology called Docker that is solely executable on one or more single board computers like the Raspberry Pi by providing automatic scaling and resilience of all sensor services. For SensIoT we extended the capability of MonTreAL to integrate commodity servers into the cluster to enhance the ease of setup and maintainability on already existing infrastructures. Therefore, we followed the paradigm to distribute microservices on small computing nodes first, thus not utilizing well-known cloud computing concepts. To achieve resilience and fault tolerance we also based our system on a microservice architecture, where the service orchestration is solved by Docker Swarm. As proof of concept, we are able to present our current data collection of the University of Bamberg's Library that runs our system since autumn 2017. To make our system even better we are working on the integration of other sensor types and better performance management of SD-cards in Raspberry Pis.

1. Introduction

In the last couple of years, advancements in Internet technologies, which enabled networking of everyday objects, significantly increased the popularity of the Internet of Things (IoT). The IoT "describe[s] embedded devices with Internet connectivity, allowing them to interact with each other, services, and people on a global scale [to] increase reliability, sustainability, and efficiency by improved access to information" [1]. Systems, which affect each other, can be interconnected like home and building automation with environmental monitoring to allow information to be shared between these systems. With low powered wireless embedded devices, which require little infrastructure, like the popular Raspberry Pi (RPi), a cheap fully fledged general purpose computer with a small footprint, it is possible for everyone to build low-cost ubiquitous sensing systems, whether low or high scaled, with ease to get cheap access to monitoring systems.

Unfortunately, the deployment and maintenance of such a sensor monitoring system for the IoT is expensive, inflexible, and very costly due to the lack of a general open-source framework, which allows universal application and extensibility and adopts proven technologies to simplify the deployment and management without the burden of costly infrastructures or cloud services in the background.

Like MonTreAL [2], SensIoT is based on the work of Lewis et al. [3], which proposes an environmental monitoring solution in a quality-controlled calibration laboratory. They used the RPi 1 with the Raspbian operating system (OS) in

their architecture while their sensor interfacing is written in the Python programming language. For their web-based monitoring and graphing, they used Cacti, which reads the sensor data. Furthermore, they also introduced parallelism of reading sensor data by using Linux's cronjobs.

Another prototype for the e-health sector, which was implemented by Jassas et al. [4], consists of RPis with attached e-health sensors. These medical sensors measure patients' physical parameters which in turn are collected and transferred to the cloud environment by the RPi to get real-time data. The application running on the RPi is written in C++ and utilizes TCP sockets for data transmissions. Nevertheless, their former prototypes do not consider the privacy of sensor data and miss an architectural concept.

Hentschel et al. [5] introduced a concept with supernodes, which are simply sensor enhanced RPis. "They are capable of local compute operations, as well as transmitting data to a centralized database. Each node can behave autonomously to carry out tasks like sending tweets, processing data, dynamic reconfiguration, and communicating with other devices" [5].

Likewise, Boydstun et al. [6] proposed a drifter node sensor network for environmental monitoring and exposed the potential of the RPi. Their sensor devices are equipped with a camera, a GPS module, and a Wi-Fi USB adapter to broadcast an ad hoc network. The components were sealed within an acrylic case and placed in coastal areas to measure water dynamics and gather data about the seafloor.

In contrast to those approaches, which mostly cover small application areas with very specific conditions and cloud connectivity, the further development of MonTreAL focuses more on being a general sensor monitoring framework fitting more use cases by being easily extensible and applicable. To enable a straightforward updatable, scalable, and manageable framework with appropriate technologies on energy efficient devices, SensIoT continued to make heavy use of lightweight container virtualization. Moreover, obtained sensor data is still processed locally without the need to upload it to cloud services or expensive infrastructures and without the burden to overflow the core network with unnecessary data traffic.

Consequently, there is currently no general sensor monitoring framework similar to SensIoT available and besides MonTreAL none of them makes use of container virtualization to simplify the overall deployment and maintenance of their application.

2. Foundations of SensIoT

MonTreAL's high-level architecture, depicted in Figure 1, consists of Workers, i.e., SBCs like the RPi, which are running services to address a variety of sensor types and collect environmental information of connected physical sensors, which in turn are sent to a server in a uniform format utilizing a messaging queue.

The server, i.e., the second component of the framework called manager, accumulates and stores gathered data and provides a user interface to allow users to view data in a suitable way utilizing a simple web interface [2].

To achieve a simple "to set up distributed system of IoT devices equipped with specific sensors and to gather,

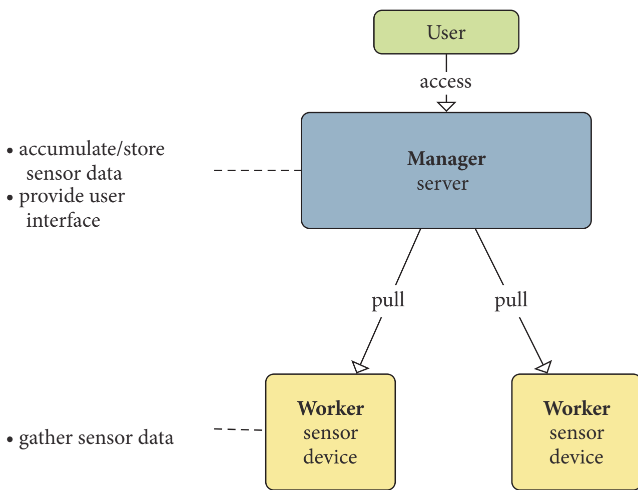


Figure 1: Simplified architectural overview of MonTreAL [7].

accumulate, and store the collected data" [2] MonTreAL takes advantage of Docker and Docker Swarm, which allow the framework to run all of its services within containers. This enables easy distribution and simple management of MonTreAL's components as well as resilience and reliability through the underlying Docker Swarm paradigm [2].

MonTreAL provides an easily manageable solution for monitoring environmental properties and virtualization at IoT level but falls short at providing a solution to implement a wider variety of sensor types besides temperature and humidity sensors and does not mention technologies to efficiently handle steady sensor data.

Handling an incoming deluge of sensor data and efficiently accumulating, storing, and visualizing the same require careful consideration of underlying technologies. Despite common database solutions being capable of handling steady sensor data, they provide no optimization regarding disk space usage, which might grow unrestricted, and optimized queries, which might take an unacceptable amount of time in consideration for hundreds, thousands, or even millions of datasets.

3. Container Virtualization for Use Cases of the Internet of Things

Virtualization is by far not a new concept, with the first steps already done in the 1960s on IBM mainframes. However, its further development and dissemination have been prevented by hardware and software limitations for quite a long time. Today, advances regarding the efficient abstraction of hardware resources led to the availability of many virtualization solutions and turned them into one of the core technologies enabling cloud computing [8, 9].

Virtual machines (VMs) used to be the most common approach of software virtualization for a long time. They enable the concurrent execution of multiple OSs on a single host computer by utilizing a hypervisor. The hypervisor, or also called VM monitor, is an additional software abstraction level, which runs on the host OS (type 2 hypervisor) or

directly on top of the hardware (type 1 hypervisor) and manages the guest OS's concurrent access on the common underlying hardware (CPU, network, storage, etc.). Without a hypervisor, the VMs would compete for the same resources at the same time, which obviously would not work at all. In a common VM scenario, each VM runs a single application based on its individually required binaries or libraries using the guest OS. While it is also possible to run many applications on a single VM, separation is often seen as a major advantage of VMs: it can increase the reliability of applications, as a crash or malfunction inside a single VM only affects this VM and most likely not any other [8]. Nowadays, VMs form the backbone of basically all cloud computing solutions. Content providers like Amazon or Google use VMs for their offerings and also create their services on top of them. So in the end, nearly all cloud workload is executed in VMs [10].

On the other side, a disadvantage of VMs is their resource consumption due to the fixed allocation of virtual CPU cores and RAM (random access memory) to each VM regardless of their actual demand [10]. However, to enable service distribution in a fog computing manner, there is a need for a more lightweight virtualization solution in order to support low power edge or IoT devices but also to allow a fast distribution of applications on different fog nodes [11]. This is where containerization comes into play.

Containers in the context of virtualization are not a completely new approach since they were enabled for Linux by namespaces with Kernel version 2.6.32. Namespaces introduce the ability to separate Kernel resources and assign them to different processes. Based on the host's process tree, the container can fork his own tree with its own root process. Whereas the host can see the whole tree (all processes), the container is only aware of himself (his own subtree) and, hence, totally isolated. The same basic isolation principle also applies for all other current namespaces: IPC (InterProcess Communication), MNT (Mount points), NET (Networking), USER (User IDs), and UTS (system identifiers) [12]. Additionally, the important Linux technology *cgroups* allow it to monitor and limit the resource usage of a specific process and its children.

All in all, namespaces enable concurrent execution of different containers on the same host OS using the same Kernel. This contrasts with VMs, which require a Kernel for each guest OS, and, furthermore, allows containers to bypass the initially mentioned resource consumption handicap of VMs.

Docker (https://www.docker.com/), the world's leading software container platform and "a tool that helps [to] solve common problems like installing, removing, upgrading, distributing, trusting, and managing software" [13] and provides a modern solution to tackle common software problems. By taking advantage of the former virtualization approaches, it achieves an easily maintainable and deployable system.

Applications which need to run a variety of different services on several devices make it economically reasonable to utilize a technology providing easy deployment and high portability within a distributed system environment. Docker containers are lightweight, stand-alone, executable packages of software that include everything to run them (runtime, dependencies, code, system libraries, system tools, and settings) and they can be easily distributed.

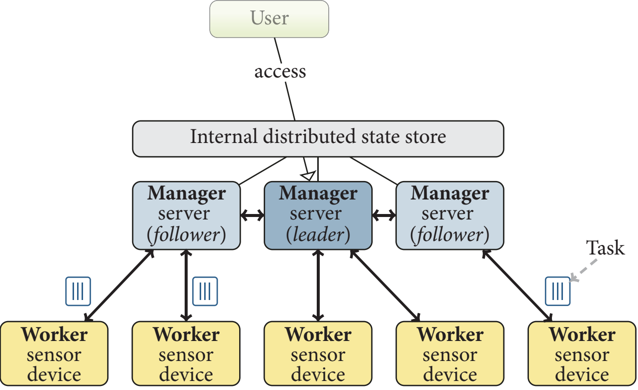


Figure 2: Functionality of Docker Swarm [7].

A model of such a container, bundled with all the files that should be available to run the packaged program, is called a Docker Image. Docker Images represent the "shippable units in the Docker ecosystem" [13] and can be used to create and run an arbitrary number of containers without additional effort [13].

The extension, Docker Swarm (https://docs.docker.com/engine/swarm/), i.e., a Docker Engine in swarm mode, is a cluster of Docker Engines providing a way to simplify building distributed system environments and dynamic-deployment topologies of containers. This goal is achieved by featuring a decentralized design, scaling, automated state reconciliation, multihost networking, load balancing, security, and more. The swarm cluster consists of at least one manager node, a Docker Engine, which takes care of the cluster's topology and maintains the overall cluster state. In case of multiple manager nodes, they start in the follower state and use the Raft Consensus Algorithm [14] to negotiate the global cluster state and elect one leader node for the cluster as depicted in Figure 2.

All manager nodes schedule services, which are blueprints for tasks being executed on worker nodes, respectively. Additionally, the cluster can contain an arbitrary number of worker nodes whose sole purpose is to execute tasks which can be dynamically added or removed at runtime while manager nodes possess worker capabilities too. The manager node will react to other nodes joining the cluster by immediately starting defined services on the new device or compensate inoperative units by recreating respective services on other still operating devices.

Furthermore, Docker's native support for overlay networks allows for a virtual network to span between devices which enable unsophisticated communication between services within the swarm, while the manager node takes care of the service distribution and discovery.

When running services on distributed devices, there is a chance to unintentionally transmit sensitive data over an insecure network. Docker Swarm provides a way to centrally manage sensitive data and securely transmit it to only those

containers that need access to it. Docker secrets are encrypted blobs of data that remain in Docker Swarm. They are only accessible to services, which have been granted explicit access to it.

The synergy of Docker containers and Docker's swarm mode grant major advantages for a system built in a distributed environment. Furthermore, utilizing Docker Compose (https://docs.docker.com/compose/), a tool for defining and running multicontainer Docker applications, the deployment, and installation of services is simplified even more. Actually, Docker Swarm supports composed files and deploys stacks, which are a composition of services, on a cluster.

Deploying and managing via Docker provides major benefits. The overall deployment is much more simplified and can be done without major effort. Adding, removing, and relocating new devices, e.g., RPis, can be done at runtime and does not need additional configuration to work properly. Moreover, Docker Swarm ensures a stable system by relocating and restarting services in case of failures.

4. Architectural Modularity of SensIoT

The principles of SensIoT are the same as the ones of MonTreAL: SensIoT operates with an arbitrary number of small sensor devices like the RPi which can be equipped with sensors to collect environmental information. The RPi was selected because of its relatively small footprint and low power consumption [5]. In addition, the RPi 3 Model B has wireless LAN and Bluetooth Low Energy (BLE) on board, which allows any distribution as long as a power source is available (https://www.raspberrypi.org/products/raspberry-pi-3-model-b/), which might be favorable when running a distributed sensor network. Furthermore, it is a very inexpensive, readily available, and mass-produced SBC providing interfaces for a wide variety of different sensors through GPIO and USB and being able to run Linux distributions [5]. Therefore, it is capable of running all necessary applications, especially Docker, which is nowadays officially supported on ARM architectures. The collected data is sent to the server, which aggregates, stores, and processes the information to create a useful representation for the user as depicted in Figure 3.

While SensIoT keeps most of the principles of MonTreAL, it was improved to fit more use cases and to be easily extensible.

5. Dataflow of SensIoT's Modules

Like its predecessor MonTreAL the framework consists of two logical components as depicted in Figure 3: An amd64-based server, subsequently just called server, which runs services to receive, aggregate, store, and process sensor data and to manage an arbitrary number of connected amd64/arm-based sensor devices, subsequently called sensor(s), which in their turn run services to continuously collect and temporary hold environmental information of physical sensors connected through USB or GPIO (general purpose input/output). Compared to Docker Swarm in Figure 2, every sensor slips into the role of a Swarm Worker and the server into the role of the Swarm Manager. Since Docker Swarm supports multiple manager nodes, all services on them can be distributed and scaled over multiple server instances accordingly.

Every service runs within its own isolated Docker container and all containers together are managed by Docker Swarm to simplify the overall deployment and maintenance.

To maintain data the server runs a database service and a dashboard service to provide the user with an appropriate way to view and analyze the collected information. Furthermore, the server provides a simple web API for other machines to fetch sensor data.

For administrative purposes, users can access several services like the Docker Swarm Visualizer (https://github.com/dockersamples/docker-swarm-visualizer) or Portainer (https://portainer.io/), a simple management UI for Docker, to manage all running containers.

All these services are accessible through Traefik (https://traefik.io/), a reverse proxy. Applications deployed utilizing Docker are configured once beforehand and started without additional effort. Services are distributed automatically among all devices within the system.

Unfortunately, there is a serious limitation within this approach in our case: Docker Swarm does not allow running services in privileged mode, i.e., with root privileges, which might be necessary when reading data from sensors depending on their interface. For example, to access sensors attached to RPi's GPIO interface, the service needs access to `/dev/mem` which is only accessible by the root user. While there are several approaches on the web to avoid these limitations, but not without blowing a hole in both security and system stability protections, SensIoT makes use of the Docker Engine API to communicate with the local Docker daemon on every sensor device to run one or more privileged *sensor reading* services.

Environmental information is continuously collected by these respective sensor services and processed within the sensor device by enriching every record with metadata regarding its origin before it is sent to the server.

6. Data Acquisition by the Sensor's Module

Since SensIoT evolved from MonTreAL, its original focus laid on working with temperature and humidity sensors and, therefore, SensIoT sensor devices can currently be equipped with remote ASH2200 (https://www.elv.de/elv-funk-aussensensor-ash-2200-fuer-z-b-usb-wde-1-ipwe-1.html) sensors through USB utilizing a USB-WDE-1 (https://www.elv.de/usb-wetterdaten-empfaenger-usb-wde1-komplettbausatz-1.html) receiver and low-cost DHT11, DHT22, and AM2302 (https://learn.adafruit.com/dht) sensors through GPIO.

However, the RPi can manage a variety of different sensors through its USB and GPIO interfaces and, therefore, SensIoT can be easily extended with other sensor types.

The sensor service, which implements the specific sensor driver to regularly read from the sensor's interface, pulls data from the attached physical sensor as depicted in Figure 4.

Data provided by physical sensors may consist of different complexity depending on the actual sensor: as plain values like most GPIO sensors provide, e.g., 24.0 for temperature

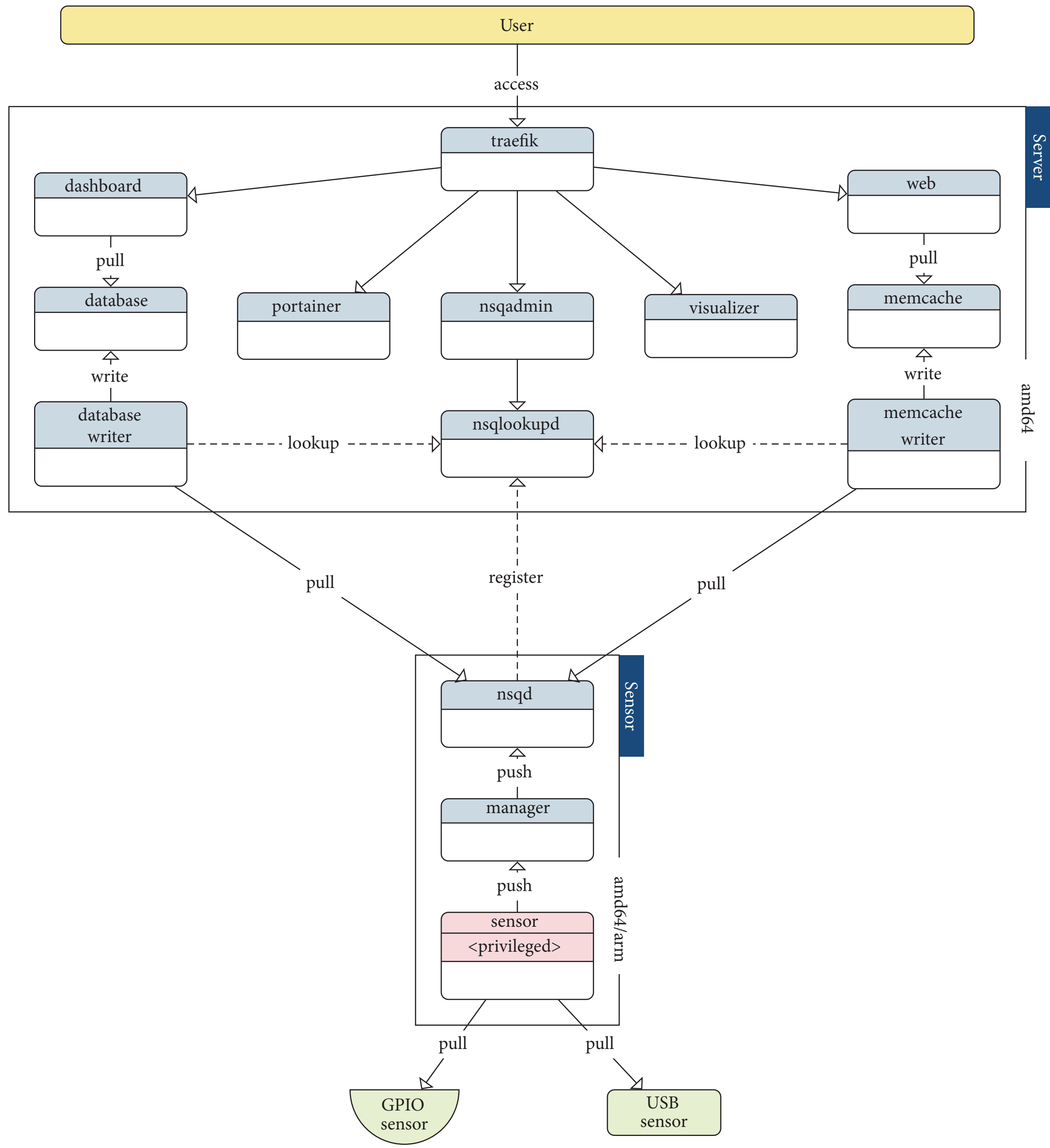


Figure 3: Detailed architectural overview of SensIoT [cf. [7]].

and 60.0 for humidity when using the DHT11/22 sensors, or as a more complex structure like the USB-WDE-1 with attached ASH2200 sensors provides, which includes values for up to eight individual sensors plus mean of all measured values, rain forecast, and several additional data as shown below:

$1;1;;24,2;;;;;;;60,0;;;;;;;24,2;60,0;0,0;0;1;0<cr><lf>

To circumvent problems with these manufacturer specific data formats and to guarantee desired exchangeability between compatible systems [15], every sensor reading service converts its received data into a more universal format utilizing the JavaScript Object Notation (JSON) (https://www.json.org/) format, where a combination of `sensor_id` and `type` forms a unique identifier for every sensor per device containing an arbitrary number of measurements.

```
{
  "sensor_id": 1,
  "type": "DHT22",
  "measurements": [
```

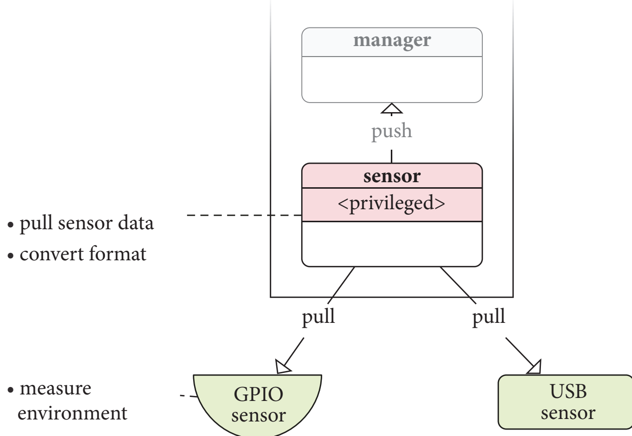


Figure 4: Dataflow of a physical sensor to sensor container (excerpt of Figure 3).

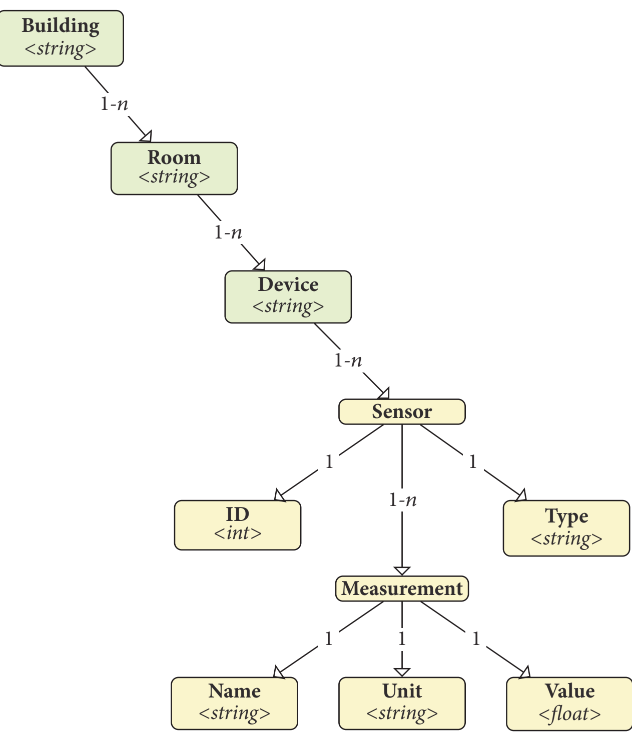


Figure 5: Tree diagram of the context enhanced data format.


```
        {
            "name": "temperature",
            "value": 24.2,
            "unit": "\u00b0C"
        },
        {
            "name": "humidity",
            "value": 60.0,
            "unit": "%"
        }
    ]
}
```


Every individual measurement consists of a `name`, describing the measured value, e.g., temperature, humidity, noise, or movement, the measured `value` and a `unit`, e.g., °C, %, dB, or whatever fits best to describe the specific measurement with an abbreviation. When expanding SensIoT by implementing new sensor drivers, this convention must be ensured.

Since sensor reading containers are not part of Docker Swarm, sensor data is sent to the local manager service through a TCP socket, where it is processed further.

To provide a richer description and context of the measured data, the manager service appends additional information about the sensor's location to each measurement and the time the measurement was done to be able to trace back the data to its origin later since the simple numerical values, which are provided by most sensors, require a rich context to be useful. Later, this metadata allows for more complex queries to filter, group, and analyze data and to be able to draw conclusions from the gathered data [15].

The data format, which is transmitted by the sensor device to be processed on the server side, is the basic unit which every processing service relies on afterward. It contains all necessary information in order to unmistakably trace back every measurement to its origin. Figure 5 shows SensIoT's data format in a logical tree diagram and the hierarchy of the attributes.

Thereafter, the short-lived sensor data is ready to be sent to the server utilizing a distributed messaging queue.

7. Data Transmission by a Messaging Queue

SensIoT implements NSQ (http://nsq.io/), a real-time distributed messaging platform, which offers high scalability and ease of use to realize interdevice communication.

Its components consist of an arbitrary number of messaging queues (*nsqd*), which receive, queue, and deliver messages to clients, and a directory service (*nsqlookupd*), which manages topology information of the network and provides addresses of nsqd instances to possible clients.

NSQ features support of distributed topologies with no single point of failure, high horizontal scalability without brokers allowing for seamlessly adding new nodes, communication over TCP, supporting client libraries in several programming languages, and TLS for secure connections.

NSQ implements the publish-subscribe pattern to provide greater network scalability. Every single nsqd instance is designed to handle multiple streams of data called topics and every topic has one or more channels while every channel receives a copy of all messages for a topic. A nsqd instance maintains a long-lived TCP connection to one or more nsqlookupd instances and periodically pushes its state, which again is exposed on an HTTP endpoint for consumers with regard to a polling policy. Consumers can look up topics

at one or more nsqlookupd instances. They can subscribe to the topic they are interested in by either creating a new channel for themselves, which ensures that messages are delivered to this specific consumer or by joining an existing channel created by another consumer, which leads to the message being delivered to a random consumer attached to this channel.

Topics and channels are not configured beforehand but created on first use by publishing to the specific topic or by subscribing to the specific channel. Both publisher and subscriber are highly decoupled in terms of configuration due to the fact that they never need to know about each other but only about a common nsqlookupd instance.

NSQ allows SensIoT to be deployed in a dynamically distributed environment, to be highly scalable without adding complexity, and it improves overall stability by enabling replication of services. Furthermore, NSQ serves as a write buffer to orchestrate and synchronize write requests to the database of a potentially large number of sensor devices. Without such a write buffer there might be potential performance issues because of too many concurrently open connections to the database when every sensor device is able to issue write requests independently [16].

SensIoT runs several nsqd, nsqlookupd, and consumer instances to maintain high availability even if single devices become inoperable.

The throughput of NSQ is high enough to handle messages of several hundred or even thousands of sensors simultaneously. The dynamic creation of topics and especially channels allows adding and removing new nodes and consumers at runtime without interfering with other nodes and without the need to reconfigure running nsqd or nsqlookupd instances.

SensIoT features NSQ on AMD64 and ARM architecture allowing any user-defined distribution of these services.

Figure 6 shows nsqd instance running on a sensor device which is registered to the nsqlookupd instance running on the server. The sensor device, which has successfully received sensor data from the sensor, sends the converted data to the nsqd instance to enqueue the data. A consumer, which operates on the server, makes use of the nsqlookupd instance to discover available nsqd instances to regularly pull data from the queue by subscribing to the corresponding topic. Depending on the consumer's task, data is then written into a database or cache or otherwise processed.

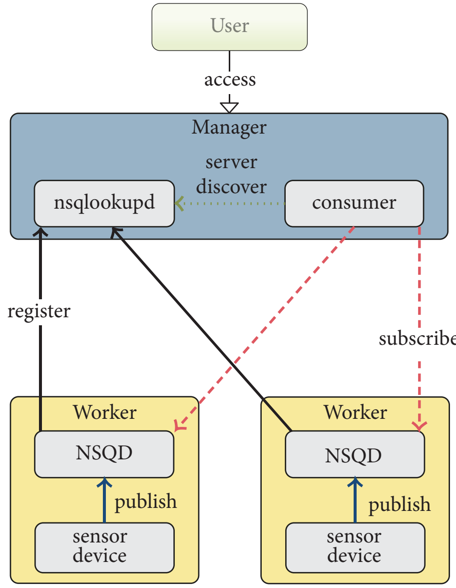


Figure 6: Architecture of NSQ [7].

8. Data Persistence by Databases

When permanently storing data produced by sensors, which are monitoring environmental properties over a long time, there are several circumstances to be noted, which inevitably lead to particular requirements the underlying data store must comply with to work efficiently.

Such data is called time series data, "a sequence of data points measured at uniform time intervals" [17], and its key difference from regular data is that it is uniquely specified by a timestamp and you will always query it over the time dimension. It is "frequently used to represent environmental properties [like temperature, humidity, CPU load, network traffic, RAM usage, etc.] and is fundamental in environmental science" [15].

A monitoring system like SensIoT with a possibly large number of attached sensor devices inevitably creates an immense amount of time series data over time and easily exceeds a few ten thousand data records even though it is not uncommon these days to collect a much higher number of time series data [17].

Furthermore, the deluge of time series data must be enriched by metadata to effectively use, understand, access, and manage it as described above, which increases the complexity and size of each dataset, the required storage to permanently store it, and the amount of time to computationally analyze it even further [15].

Relational databases like MySQL (https://www.mysql.com/) or PostgreSQL (https://www.postgresql.org/), which organize data with uniform characteristics in logically divided tables linked by relationships, "are affected by serious scalability issues when collecting hundreds of thousands (millions) metrics, making them practically unusable for large monitoring systems" [17]. Since every dataset increases the table cardinality and disk space taken with every measurement interval, storing and querying millions of datasets in traditional relational databases have a significant impact on performance because their "write efficiency is positively correlated with the data scale" [18]. Retrieving data might be affected too, "since data access time greatly increases with the cardinality of data and number of measurements" [17] because the unoptimized indexes of relational databases can become too large to be cached and "thus jeopardizing the performance of applications sitting on top of the database" [17].

Nevertheless, time series data has some interesting characteristics, which allow for several optimizations when reasoning about storing massive amounts of time series data, and "an emerging trend towards high performance, lightweight,

databases specialized for time series data" [15] and near real-time monitoring demand and analysis led to the vast growth of time series databases (TSDB).

In Figure 6, a consumer service, which implements a specific database driver, fetches data from one or more nsqd instances utilizing the nsqlookupd's discovery service. In combination with Figure 3, this *database_writer* then stores data in a database.

From the sheer amount of time series databases available, SensIoT currently supports two different databases to store sensor data: *InfluxDB* (https://www.influxdata.com) and cursive (https://prometheus.io/).

8.1. Characteristics of InfluxDB. InfluxDB is one of the most common solutions when storing large amounts of highly time-dependent data and performing real-time analysis of these is a major concern. InfluxDB was created from the ground up including a custom high performance datastore with high ingestion speed and data compression. It is entirely written in Go (https://golang.org/) and compiles to a single binary without external dependencies. InfluxDB can handle a high amount of data points per second, which sooner or later might result in storage concerns mentioned above.

Downsampling data with continuous queries, which periodically and automatically compute aggregated data to make frequent queries more efficient and retention policies, which are unique to each database and determine the data's lifecycle, ensure that the disk space needed to store large amounts of time series data over a long time is limited to a manageable size. InfluxDB can be easily configured to hold high precision data only for a limited time and lower precision data, i.e., aggregated data, for a much longer time or forever.

InfluxDB does not provide a user interface since it is part of the TICK-Stack, which consists of Telegraf that is an extensible server agent for collecting and reporting metrics, InfluxDB, Chronograf, an administration and visualization interface, and Kapacitor, as a data processing engine. All four components form a modern time series platform provided by InfluxData (https://www.influxdata.com/).

The open-source edition of InfluxDB runs on a single node but there is also an enterprise edition for high availability and for eliminating the single point of failure.

8.2. Characteristics of Prometheus. Prometheus was originally built at SoundCloud (http://soundcloud.com/) but later became a stand-alone open-source project and joined the Cloud Native Computing Foundation (https://cncf.io/) in 2016.

Prometheus features a multidimensional data model, utilizing labels, i.e., key-value pairs encoding metadata to identify series of time series data, which allows for grouping, filtering, and matching via queries, and a flexible query language to leverage this dimensionality.

Prometheus runs as an autonomous single server node and does not rely on distributed storage. Time series data is collected via a pull-based approach over HTTP (scraping) or an intermediary Push Gateway. Targets to scrape from are found via service discovery or static configuration.

Most parts of Prometheus are written in Go[17] making them easy to build and deploy as static binaries. Prometheus stores data locally and runs rules to aggregate and record new time series from existing data or generate alerts. Prometheus works well for recording any purely numeric time series and is also applicable in highly dynamic service-oriented architectures. It is designed for reliability and each server runs autonomously, not depending on network storage or remote services. Furthermore, there is no need to set up extensive infrastructure to use it.

8.3. Comparison of InfluxDB and Prometheus. To summarize, both databases address the same problem space but partly follow different approaches and differ in various aspects. While InfluxDB supports various data types and variable timestamps, ranging from nanoseconds to seconds, Prometheus is float-only and has fixed timestamps. Their compression level is similar since they both use the implementation of Facebook's Gorilla Paper [19]. They both support high scalability and high availability through clustering or federating but in InfluxDB's case, these features are restricted to the commercial version. Here is where Prometheus' pull-based approach to scrape data comes in handy: for high availability, you can use an arbitrary number of different cross-federated Prometheus servers scraping the same target since the targets do not need to know the server's address.

A further major design difference might be that Prometheus has no easy way to attach other timestamps than *now* to metrics. In a system like SensIoT, where time series data is not immediately stored in the database because it could remain in the queue for a while, this may be a deal breaker for some use cases.

Nevertheless, while a commercial company maintains InfluxDB following the open-core model and offers premium features like closed-source clustering, hosting, and support, Prometheus is an independent open-source project maintained by a number of companies and individuals.

SensIoT supports both databases since both provide important features needed for a monitoring framework for the IoT. While it is hard to draw conclusions from a deluge of time series data, SensIoT also features proven dashboard solutions to provide a meaningful visualization for the user.

9. Data Visualization through Dashboards

As described above time series databases are mostly used in combination with dashboards to provide an appropriate visualization of the time series data they store and can be queried for. Some databases already come with their own web interface which allows for querying and graphing; others rely on additional applications. Since Prometheus' graphing capabilities are very limited and should only be used for ad hoc queries and debugging, the following sections will introduce Chronograf, InfluxDB's visualization engine and administrative interface, and Grafana, an open-source metric analytics and visualization suite for time series data.

9.1. Chronograf. Chronograf, which is part of the TICK-Stack and responsible for data visualization and administrative

tasks, offers a complete dashboard solution for visualizing data with precreated dashboards and the possibility to create custom dashboards.

Besides its graphing capabilities, Chronograf's web and administration interfaces allow users to easily execute and visualize ad hoc queries utilizing InfluxDB's SQL-like query language, which can be "clicked together", insert data for testing purposes and debugging, and create an arbitrary number of dashboards. Additionally, it allows viewing and managing alerts, which requires Kapacitor (https://www.influxdata.com/time-series-platform/kapacitor/), managing InfluxDB's retention policies for every database individually, and managing users, organizations, and data sources.

For access control, Chronograf supports authentication via OAuth 2.0 (https://oauth.net/2/) providers to offer authorization and authentication of users and role-based access. Chronograf supports OAuth from GitHub, Google, and custom providers with own authentication, token, and API URLs. To provide data confidentiality, data integrity, and server authentication, a secure connection over HTTPS is also supported when accessing Chronograf.

9.2. Grafana. Grafana is an open-source metric visualization and analytics suite, which is most commonly used for visualizing time series data for application and infrastructure metrics including industrial sensors, weather, home automation, and process control. It offers a customized query editor for each data source, which features the particular capabilities the respective data source's query language exposes; i.e., you can "click together" your queries, while Grafana instantly updates the particular graph so you can effectively explore your data in real-time making use of its query inspector, which provides feedback about the current query.

Grafana supports templating allowing a more dynamic and interactive graphing through variables within each query acting as placeholders. Users can choose the variable values by selecting from a drop-down menu above the graph and actively change data that is currently displayed by this graph.

Dashboards can be easily exported and imported, shared via links or snapshots encoded into a static or interactive JSON document, and tagged to provide quick, searchable access to all dashboards of an organization through Grafana's dashboard picker, which allows filtering by name, tags, or its starred status.

Another interesting feature is a scripted dashboard. If new data sources are added or metrics' names are changed in a defined pattern, it might help to dynamically create dashboards using Javascript since every dashboard is represented by a JSON object.

Grafana can be entirely configured through configuration files or environment variables and managed through a basic CLI and, moreover, provides a rich HTTP API for administrators.

9.3. Comparison: Chronograf and Grafana. Although the dashboards shown in the last two sections look very similar, there are considerable differences between these two solutions and it seems to be unfair to equate them since they follow different strategies.

Chronograf is part of the TICK-Stack and heavily integrated with InfluxDB to provide interfaces for visualization and administration; precisely, this makes it impossible to use it with other data storage solutions. Grafana supports many different data sources and is heavily customizable and more of a full-featured dashboard solution in contrast to Chronograf, which feels more like a very basic visualization and administration tool.

Nevertheless, both provide a solid dashboard solution for everyone. Both support "clicking together" dashboards and even queries, so users do not need considerable experience with their underlying technology. But Chronograf misses the high customizability of Grafana, which provides a wide variety of options to customize your graphs and dashboards. You can easily change the range and label of your axis or series or annotate time ranges within your graph. Furthermore, Grafana provides a lot of preconfigured dashboards and official and community built plugins for individually extending and customizing your local Grafana instance and a built-in alerting and notification system.

Even though SensIoT features both dashboard solutions, it is recommended to go with Grafana since it is independent of the underlying data store, i.e., InfluxDB or Prometheus, and provides a feature-rich and fully fledged dashboard solution, which should satisfy almost every need. While a graphical dashboard solution provides meaningful information for users, it is most likely useless for other machines at the same time, which are interested in fetching data from the system for further processing.

10. Accessing SensIoT through Its Web API

For representational purposes, a very basic HTTP API is currently available which allows querying the latest sensor data and a list of all sensor devices available in an unsophisticated way. It is provided by the *web* container that queries a memcache backend to which the data is written by a *memcache_writer* as depicted in Figure 3.

However, further development has been dropped for now since there are already very powerful HTTP APIs included in some database solutions. Nevertheless, there are consumers like PRTG (https://www.paessler.com/prtg), a commercial network monitoring system, which was used as backend during development but later replaced by InfluxDB and Grafana, or Prometheus, which provide a pull approach to fetch data and, therefore, such solutions need an API to get their data.

SensIoT's web API is easily extensible and allows providing data in every form. Table 1 shows a query to get a full sensor list of all current sensors and a query to get the latest data of a specific sensor in JSON.

Nevertheless, SensIoT's HTTP API remains too basic to allow meaningful operations on or with it but it can be enhanced if necessary.

11. Extensibility of SensIoT

One of SensIoT's most important goals, to provide a general sensor monitoring framework for the IoT, is to be easily

Table 1: SensIoT's web API.

Service	URL/Description
Sensor List	web.sensiot.de/json/sensorlist Get a list of all sensor devices and their location available
Sensor Data	web.sensiot.de/json/<device_id>/<sensor_type>/<sensor_id> Get the latest data for the respective sensor

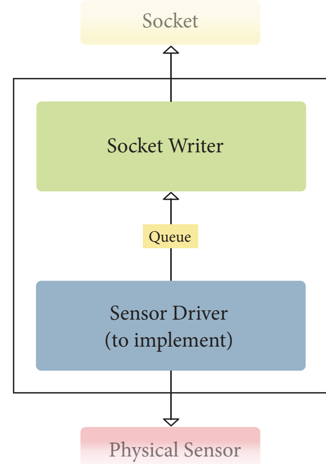


Figure 7: Detailed overview of a sensor implementation.

extensible and enable users to utilize it for almost every application, which involves collecting, storing, and analyzing environmental data with small, distributed SBCs with attached sensors. Therefore, it was designed to allow adding new sensor types and consumers without major expense and trouble.

11.1. Implementation of New Sensors. Theoretically, there are only two restrictions when implementing new sensor drivers regarding their employability in SensIoT, as shown in Figure 7. Firstly, the sensor driver must follow the data format for measurements to allow interoperability with the rest of the system and, secondly, it must send its data through the TCP socket. You are neither restricted to a specific programming language nor restricted to other techniques, but your code must run within a Docker container and be deployed via Docker Swarm. Building sensor containers in other programming languages than Python, which are not directly part of SensIoT, is not covered here.

The implementation of new sensor types only consists of a few steps, which roughly contain implementing the new driver, defining a configuration, and connecting it with a socket writer utilizing a queue and finally declaring it in the framework's service definition. The only restriction is that SensIoT uses the Python programming language.

We begin by taking a look at how services are created with regard to the configuration and sensor declaration. As depicted in Figure 8, the `manager` class gets the global configuration ① from the `ConfigurationReader` class. Based on that configuration, it creates required services ② by calling the `service` class, which holds all service definitions and how they are assembled, i.e., ASH2200 sensor Driver + Socket Writer or NSQ Reader + InfluxDB Writer. The `service` class looks up the service construction plan ③ and creates instances ④ of every needed service, assembles them as required, and returns them to the `manager` class. Then the `manager` starts and monitors all services ⑤ + ⑥.

For the implementation of a new sensor, a new category can be created optionally, which describes the class of the sensor to ease the maintainability of the system. At the current state of the framework, there are only temperature and humidity sensor implementations available. For the sensor, firstly, create a new Python file, within the respective categories' directory, secondly, update the `__init__` function to set attributes as required, and implement the `read` function for your specific sensor. The functionality to periodically read and send data through the socket is inherited by the `AbstractSensor` class and must not be worried about.

```
class My_Sensor(AbstractSensor):
  def __init__(...):
    super(My_Sensor, self).__init__(...)
  def read(self):
    self.event.wait(self.interval)
    logger.info("Reading data... ")
    measurements = []
     """ read data from the sensor """
    ...
    return measurements
```

Thereafter, create a sensor configuration which can be added to SENSIOT's global configuration or the device configuration.

```
"<name>": {
  "service": "<category name>",
  "type": "<type>",
  "image": " <namespace>/sensiot:multiarch-latest",
  "device": ["<device>"],
  "command": "",
  "configuration": {
   <necessary configuration>
  }
}
```

Finally, connect the new implementation with a socket writer in `./src/services.py`. and declare its initialization. If a new category was created beforehand, a new method must be created. Otherwise, if no new category was created, it is sufficient to add a new entry to the existing sensor category.

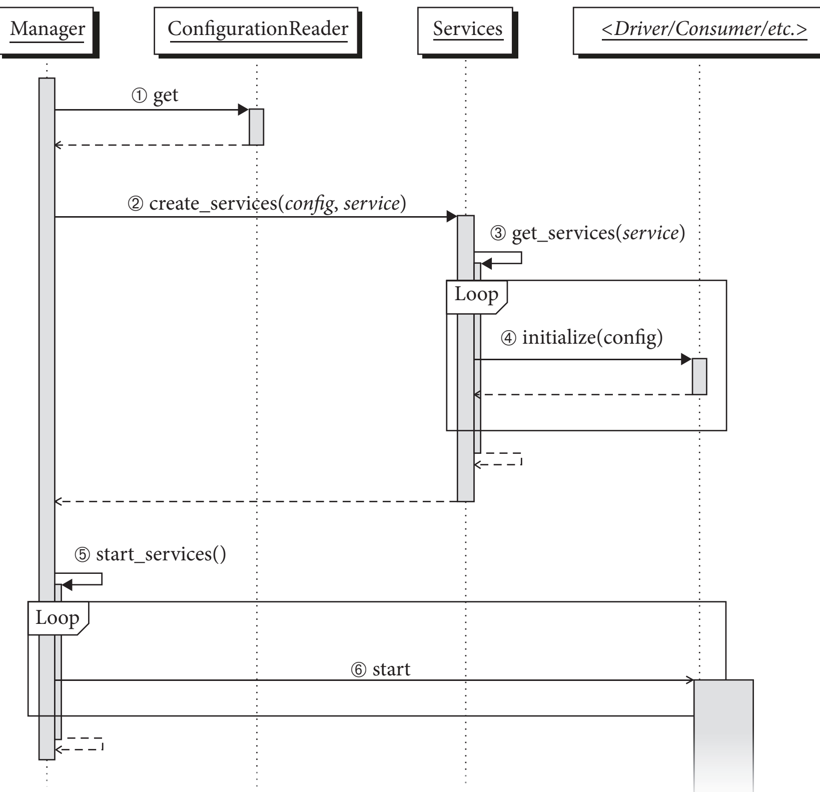


Figure 8: Sequence diagram of service initialization.

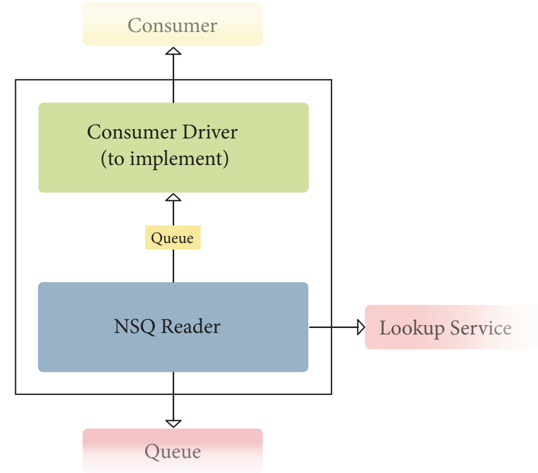


Figure 9: Detailed overview of a consumer service.

11.2. Implementation of New Consumers. It requires similar steps, as mentioned above, to implement new consumer services for SensIoT. The consumer driver must be implemented and connected to an NSQ reader instance utilizing a queue, as depicted in Figure 9. Afterward, any necessary configuration must be added to the framework's configuration file and, finally, it must be declared in SensIoT's service definition.

The whole procedure is very much like implementing a new sensor driver. The implementation should be placed in a suitable directory. For consumers, there is currently no abstract class available to inherit from, since it has been shown that consumers differ much more from each other and have less in common. Adding the consumer's configuration to SensIoT's global configuration file is similar to adding a new sensor's configuration to the configuration file, as described above, declaring the new consumer service. Additionally, you always have to write a new method to connect your implementation with the NSQ reader.

When implementing new consumers, there are additional steps required. For every service except local sensors, there is a *Docker Compose* template that is used by the corresponding *Makefile* to create and start the service. It is necessary for the implementation to create a new *Docker Compose* file, which defines the service's name, the networks it is connected to, and its deploy policies. There is an annotated `template.yml` to help writing *Docker Compose* files for new consumers.

To summarize, SensIoT basically works like its predecessor MonTreAL with several improvements regarding its application area, deployment, and extensibility. It provides a wider area of application and straightforward ways to implement new sensors and consumers and employs proven technologies to effectively handle and visualize sensor data.

SensIoT's repository (https://github.com/uniba-ub/The-SENSIOT-Framework) contains the whole source code, a testing setup which runs on your local machine, and a production setup, subsequently just called Swarm Setup, which utilizes Docker Swarm.

12. Performance Evaluation on a Raspberry Pi Cluster

The evaluation of SensIoT's prototype consists of a laptop serving as Swarm manager, four RPis 3 Model B which served as sensor devices with a USB-WDE-1 receiver connected to each, and four ASH2200 sensors placed in different rooms and outside of the building. It is important to mention that every sensor device with its attached receiver received the data of all ASH2200 sensors; i.e., every sensor device gathered exactly the same data during the evaluation period.

The prototype was configured to run the sensor reading service for the ASH2200 sensor on every sensor device and an nsqd instance on both Swarm Manager and each Swarm Worker, i.e., sensor device. The data was stored in a running InfluxDB instance as well as a Prometheus instance. Furthermore, the web API was enabled to provide current sensor data and a list of all known sensors. Grafana, as well as Chronograf, was used to provide a dashboard to view the collected sensor data whereas Grafana provided one dashboard using InfluxDB and one for Prometheus as data sources.

The prototype was running for 30 days and its functionality was checked every day while Google's cAdvisor (https://github.com/google/cadvisor) analyzed the performance characteristics of all running containers and collected various metrics.

During the evaluation period, 905,512 individual measurements for both temperature and humidity were executed and stored within InfluxDB and Prometheus, i.e., 1,811,024 individual data records in each database. Prometheus constantly uses 258 MiB of disk space. Despite Prometheus' retention policy, which deletes data older than 15 days, the disk space used never changed. Unfortunately, all data is stored in a compressed format so further analyses were not possible.

InfluxDB's disk space consumption is more of what you would expect when permanently collecting data because it shows continuously increasing values depending on the number of stored measurements and decreasing values when data is compressed or deleted due to the database's retention policy. While InfluxDB's disk space consumption can be heavily optimized by adjusting its retention policies and continuous queries, Prometheus should not be used as long-term storage for time series data but rather with other remote storage solutions.

During the test, a cAdvisor instance was running on the server as well as on one sensor device, i.e., an RPi, to analyze how many resources SensIoT's components need and to identify the resource consumption.

The sensor device's sensor service is idle most of the time while the nsqd instance and the manager service list a more frequent but low CPU activity, especially when data is transmitted. The memory usage is permanently low and does not exceed 20 MiB of RAM when added up.

To conclude, SensIoT covers several areas of application which involve monitoring environmental properties. The prototype above simulated a setup with four sensor devices, four physical sensors, and data throughput of 16 sensors since every USB-WDE-1 receiver received the sensor data of all ASH2200 sensors. While it might be sufficient to measure temperature and humidity only every hour, there are also use cases requiring a much shorter measurement interval with much more sensor devices. Therefore, it is necessary to know SensIoT's maximum data throughput which will be evaluated in the following section.

13. Stress Testing of SensIoT

The same setup as mentioned above was used to perform a stress test to evaluate how many messages per second SensIoT would be able to handle and which parts of it might fail considering the deluge of sensor data. To make sure to hit the framework's boundaries, it was configured to start four sensor mocks on every sensor device, which for their part generated sensor data with maximum capabilities. Their `sensor_count` was set to 10,000 and their `interval` to 0. The amount of generated sensor data per sensor mock laid by around 1,000 datasets per second.

From the deluge of sensor data released by the sensors, which together generated around $16,000\,msg/s$, only around 24 messages reached their endpoint in the same amount of time and were saved in the database. In the process, a maximum of 32, a minimum of 0, an average of $24 msg/s$, and in total 87,276 stored messages were achieved.

This high derivation needed further examination since both InfluxDB and Prometheus are capable of handling a much higher amount of incoming data. Unfortunately, SensIoT does not collect statistics about the message throughput for every service which is necessary to find bottlenecks in the system, but NSQ does, at least for its own components.

The total number of messages processed by all nsqd instances together was 228,001 of ~57,600,000 messages $(16,000_{msg} * 3,600_{s})$ generated by the sensor mocks which inevitably leads to the fact that one or more services running on each sensor device are causing the "data-jam".

Nsqadmin reveals more problems, even on the server side. It lists all channels, i.e., consumers, with their total number of ready-to-fetch messages and most remarkably stored messages in memory and on a disk. Depending on the channel, there are enormous numbers of messages temporarily stored on disk that indicates that the underlying consumer implementation is too slow to fetch the same amount of data within the same time this data is enqueued (~ $64\,msg/s$). This leads to an unavoidable overflow after some time. Comparing the InfluxDB consumer, which was able to consume 84,633 $(226,557_{total} - 141,924_{stored})$ messages within an hour $(23,5\,msg/s)$, with the Prometheus consumer, which was only able to consume 3,789 $(228,001_{total} - 224,222_{stored})$ messages within the same time $(1\,msg/s)$, leads to the conclusion that both implementations are unusable for some use cases, which require a high data ingestion rate.

To further evaluate the bottlenecks of SensIoT, a slightly different setup was chosen. Since the consumer implementation for Prometheus seems to have major issues, only the data flow from sensors to InfluxDB was taken into account and all services were updated to regularly log their average message throughput. Furthermore, only one sensor device,

running a single sensor mock instance with a maximum data generation rate, was used to guarantee just one data flow.

The message throughput of the sensor, which was generating around $1,000\, msg/s$, was slowed down by the very first interthread data transmission from the sensor to socket writer through their interprocess connection, Python's built-in queue. The current socket writer implementation can only handle around $125\, msg/s$, which might cause potential data loss if the sensor's throughput is higher. Unfortunately, the data throughput is further decreased since the throughput of the socket reader does not pass $15\, msg/s$. It is hard to say whether the socket reader caused the slowdown or the sending of the data to an nsqd instance utilizing the NSQ writer. But considering that the throughput of the socket reader and the NSQ writer are the same, neither the metadata appending logic nor the NSQ writer can be the cause for the drop from $125\, msg/s$ to $15\, msg/s$. These components are connected in the same way as the sensor and the socket writer through a queue with limited capacity and, therefore, this would also result in data loss and a lower message throughput of the NSQ writer.

We already showed above that the InfluxDB writer's throughput is $24\, msg/s$ and the values of $15\, msg/s$ can be traced back to the slow socket reader. Another look at the nsqadmin's dashboard verified this assumption since no data was stored in memory or on a disk.

So increasing the number of sensor devices to three to generate a higher amount of messages per second ($45 = 3_{devices} * 15_{msg}$) should prove that a higher data throughput can be handled on the server side. We know that the throughput of the InfluxDB writer is $24\, msg/s$. Fortunately, Docker Swarm allows us to easily scale services, so besides increasing the number of sensor devices the number of InfluxDB writer instances was also increased to three.

This setup shows the data ingestion rate of InfluxDB with three sensor devices generating around $45\, msg/s$ and three InfluxDB writer instances which are able to handle up to $72 msg/s$. A look at nsqadmin's dashboard reveals that all messages were processed and no "data-jam" occurred; i.e., NSQ did not have to store messages in memory or on a disk.

Figure 10 draws the final image of SensIoT's data throughput. The sensor service on the sensor device is able to transmit $125\, msg/s$ through the socket independent of the underlying physical sensor. The local manager service is only able to process $15\, msg/s$ because of the slow socket reading procedure. The InfluxDB writer can handle an average of $24\, msg/s$ but can be scaled up if higher throughput is necessary. All NSQ components in between are nothing we have to worry about since they are able to handle a multiple times higher amount of messages even with an enormous amount of sensor devices because every sensor device runs an nsqd instance.

To summarize, there seem to be little possibilities to break SensIoT by overloading since it already blights the cause in its lowest instance: the sensor devices, which drop data exceeding their capabilities. Only a huge amount of sensor devices might overburden the system; i.e., sensor data is bottled-up in the nsqd instances, which might fail sooner or later when disk space becomes low and consumers are not properly scaled.

Figure 10: Message throughput of SensIoT's modules.

All around the evaluation was very successful and shows that SensIoT can be universally used to monitor environmental properties utilizing single board computers, i.e., RPis, and arbitrary sensors, given that they can be accessed by the instated single board computers. In contrast to MonTreAL, SensIoT provides several enhancements across all areas.

SensIoT is universally applicable due to its improved code base and its overall design. MonTreAL was designed for a specific use case with a very monolithic code base, which does not allow implementing other arbitrary sensor types. In contrast, SensIoT is designed to allow new implementations without major effort. Important operations are implemented by the framework so programmers only have to reason about the specific driver implementation, either for sensors or for consumers.

Furthermore, the overall deployment is much more simplified. While there is no solution in MonTreAL to circumvent the problem with the privileged sensor container and the user has to start the privileged sensor reading container manually by logging into the device, SensIoT makes proper use of the local Docker API, not only to start but also to restart the container if it stops. With this solution, the whole deployment is performed utilizing Docker Swarm's capabilities and without starting services on every sensor device individually.

Another advantage over MonTreAL is the awareness of time series data. SensIoT implements proven technologies to handle time series data by utilizing InfluxDB or Prometheus, databases optimized for storing and querying data, which is uniquely specified by time, to provide effective data handling regarding both performance and storage size.

The basic web interface of MonTreAL, which can be described as nearly featureless, is replaced in SensIoT by more powerful solutions, namely, Grafana and Chronograf, whereby Grafana might be the best solution. This applies, regardless of the used data store, much more features and is as easy to use as Chronograf, which is restricted to InfluxDB as a data source.

Compared to MonTreAL, SensIoT provides additional HTTP APIs. On the one hand, both database solutions come with their own HTTP API to fetch sensor data, metrics, and information about targets, which allows for extensive data querying, and, on the other hand, SensIoT provides its own basic web API, which can be easily extended to provide data in arbitrary formats.

Regarding other already existing sensor monitoring solutions, which feature the distribution of single board

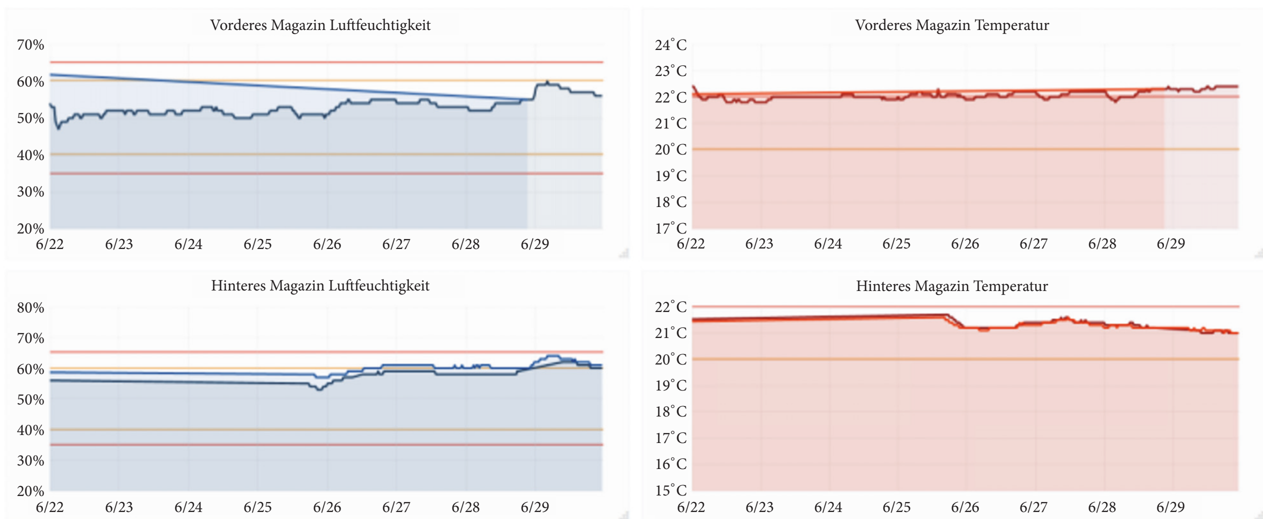


Figure 11: SensIoT's dashboard of a magazine at Bamberg University Library.

computers, SensIoT stands out by providing a general solution not bound to any specific context. Like Hentschel et al. [5], the framework recognizes the potential of the RPi, which is capable of locally computing operations. SensIoT can be configured to transfer the whole NSQ infrastructure as well as consumers to its sensor devices.

14. Conclusion

We introduced SensIoT, a general sensor monitoring framework for the IoT. It most notably stands out by utilizing Docker and Docker Swarm to tackle common software problems and to enable flexible data collection. SensIoT employs IoT devices to collect environmental data and implements proven technologies to efficiently handle sensor data. It provides appropriate database and dashboard solutions for efficient data storage and meaningful data visualization and was evaluated in a real-life scenario to validate its functionality and reliability.

Currently, SensIoT is running in seven magazines, one server room, at the lending desk, and in one carrel using a total amount of 18 sensors. Figure 11 shows continuous temperature and humidity readings of two locations at Bamberg University Library. In April 2017, we started the trial phase until September and since then we are running it as a fully operational system.

SensIoT can easily be deployed without major effort and without expensive infrastructure or the burden of a cloud service. Moreover, the implementation of new sensors and consumers is simplified and done in a matter of minutes for someone who is experienced and knows the details of SensIoT. However, the process can be further optimized to hide internals from the user. Since it is always the same procedure, the coupling of the required components can be done by the framework in the background and users must not deal with implementation details.

Adding new sensor devices works as expected when we reimplemented the ASH2200 and DHT∗ drivers to fit the current design. For someone who regularly worked with MonTreAL and SensIoT and experimented with different solutions for different problems of which some heavily failed and some worked well, we can say that SensIoT promises high potential and with every dismissed solution new ways open up to create a better solution for a general sensor monitoring framework for the IoT. So in the current state, SensIoT is not near to be a finished product since there is much potential for improvements but it is a promising foundation.

Finally, SensIoT is open-source and available on GitHub via https://github.com/uniba-ub/The-SENSIOT-Framework.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

The authors would like to thank the Hypriot Team (https://blog.hypriot.com/crew/) for their relentless effort to port the Docker framework to the ARM platform and their work on Hypriot OS.

References

[1] S. C. Mukhopadhyay, *Internet of Things: Challenges and Opportunities*, Springer Science & Business Media, 2014.

[2] M. Großmann, S. Illig, and C. Matějka, "Environmental monitoring of libraries with Montreal," in *Research and Advanced Technology for Digital Libraries*, J Kamps, G. Tsakonas, Y. Manolopoulos, L. Iliadis, and I. Karydis, Eds., vol. 10450 of *Lecture Notes in Computer Science*, pp. 599–602, Springer International Publishing, Cham, 2017.

[3] A. J. Lewis, M. Campbell, and P. Stavroulakis, "Performance evaluation of a cheap, open source, digital environmental monitor based on the Raspberry Pi," *Measurement*, vol. 87, pp. 228–235, 2016.

[4] M. S. Jassas, A. A. Qasem, and Q. H. Mahmoud, "A smart system connecting e-health sensors and the cloud," in *Proceedings of the IEEE 28th Canadian Conference on Electrical and Computer Engineering (CCECE)*, pp. 712–716, Halifax, NS, Canada, May 2015.

[5] K. Hentschel, D. Jacob, J. Singer, and M. Chalmers, "Supersensors: raspberry pi devices for smart campus infrastructure," in *Proceedings of the IEEE 4th International Conference on Future Internet of Things and Cloud (FiCloud)*, pp. 58–62, Vienna, Austria, August 2016.

[6] D. Boydstun, M. Farich, J. M. Iii, S. Rubinson, Z. Smith, and I. Rekleitis, "Drifter sensor network for environmental monitoring," in *Proceedings of the 12th Conference on Computer and Robot Vision, CRV 2015*, pp. 16–22, Canada, June 2015.

[7] M. Großmann, S. Illig, J. Lappe, and C. Matějka, "Bestandsmonitoring in kulturellen Einrichtungen," *ABI Technik*, vol. 38, no. 4, pp. 344–353, 2018.

[8] M. Portnoy, *Virtualization Essentials*, John Wiley & Sons, 2012.

[9] R. Buyya, C. Vecchiola, and S. T. Selvi, *Mastering Cloud Computing: Foundations and Applications Programming*, Newnes, 2013.

[10] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," in *Proceedings of the 15th IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '15)*, pp. 171-172, USA, March 2015.

[11] OpenFog Consortium Architecture Working Group, "OpenFog reference architecture for fog computing," OPFRA001 20817: 162, 2017.

[12] O. Liebel, *Skalierbare Container-Infrastrukturen: Das Handbuch für Administratoren und DevOps-Teams*, Inkl. Container-Orchestrierung mit Docker, Rancher & Co. Rheinwerk Verlag GmbH, Rocket, Kubernetes, 2017.

[13] J. Nickoloff, *Docker in Action*, Manning Publications, Company, 2016.

[14] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *In Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference*, pp. 305–320, 2014.

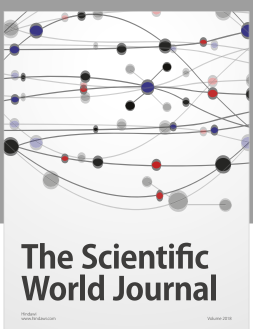
[15] B. Leighton, S. J. D. Cox, N. J. Car, M. P. Stenson, J. Vleeshouwer, and J. Hodge, "A best of both worlds approach to complex, efficient, time series data delivery," in *Environmental Software Systems. Infrastructures, Services and Applications*, R. Denzer, R. M. Argent, G. Schimak, and J. Hřebíček, Eds., vol. 448, pp. 371–379, Springer, 2015.

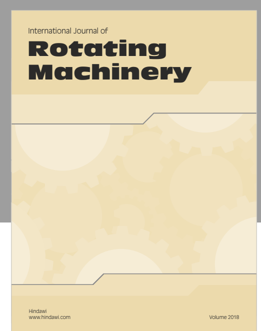
[16] S. Zareian, M. Fokaefs, H. Khazaei, M. Litoiu, and X. Zhang, "A big data framework for cloud monitoring," in *Proceedings of the 2nd International Workshop on BIG Data Software Engineering - BIGDSE '16*, pp. 58–64, Austin, Texas, May 2016.

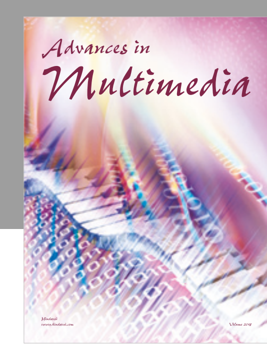
[17] L. Deri, S. Mainardi, and F. Fusco, "tsdb: a compressed database for time series," in *Traffic Monitoring and Analysis*, A. Pescapè, L. Salgarelli, and X. Dimitropoulos, Eds., vol. 7189 of *Lecture Notes in Computer Science*, pp. 143–156, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

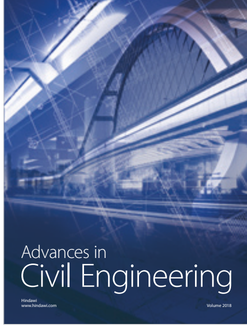
[18] C. Li, J. Li, J. Si, and Y. Zhang, "FluteDB: an efficient and dependable time-series database storage engine," in *Security, Privacy, and Anonymity in Computation, Communication, and Storage*, G. Wang, M. Atiquzzaman, Z. Yan, and K.-K. R. Choo, Eds., vol. 10658 of *Lecture Notes in Computer Science*, pp. 446–456, Springer International Publishing, Cham, 2017.

[19] T. Pelkonen, S. Franklin, J. Teller et al., "A fast, scalable, in-memory time series database," *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, vol. 8, no. 12, pp. 1816–1827, 2015.









Submit your manuscripts at
www.hindawi.com