MACHINE LEARNING & DATA ANALYTICS

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Node.js-based Sensor Library Framework for a Data Concentrator in Home Monitoring

Project Thesis in Mechanical Engineering

submitted
by

Felix Tuchnitz

born 12.03.1994 in Gräfelfing

Written at

Machine Learning and Data Analytics Lab (CS 14)
Department of Computer Science
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)

in Cooperation with

Institute for Factory Automation and Production Systems (FAPS), FAU Erlangen-Nürnberg

Advisors:
Arne Küderle, M. Sc., Robert Richer M. Sc., Nils Roth M. Sc., Prof. Dr. Björn Eskofier
(Machine Learning and Data Analytics Lab, FAU Erlangen-Nürnberg)
Jochen Bauer M. Comp. Sc., Prof. Dr.-Ing. Jörg Franke
(Institute for Factory Automation and Production Systems (FAPS), FAU Erlangen-Nürnberg

Started: 01.11.2018

Finished: 01.04.2019

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Die Richtlinien des Lehrstuhls für Bachelor- und Masterarbeiten habe ich gelesen und anerkannt, insbesondere die Regelung des Nutzungsrechts.

Erlangen, den 01.04.2019

Abstract

Wearable computing technologies have attracted a lot of attention from the healthcare sector in recent years. Driven by the rapid rise in healthcare costs and an aging population, sensor-based home monitoring systems are expected to play an increasingly important role in modern healthcare infrastructure. Worn inside, on, or near the human body, wearable devices are able to seamlessly capture vital parameters, such as heart rate, blood pressure, and movements. In order to derive medical benefits from these sensor readings, the data must be included in clinical decision-making processes. For this reason, a suitable technical infrastructure is required to make the data available. Leveraging wireless communication technologies and cloud computing, data concentrators are therefore used to aggregate the sensor data and store it in a cloud. Smart algorithms utilize the advances in artificial intelligence to gain insights from this data. Integrating this new personalized health knowledge into the existing healthcare infrastructure has the potential to improve the quality of prevention and treatment techniques.

In such an ecosystem, the communication with the wearable sensors is mainly controlled by the application running on the data concentrator. In existing home monitoring infrastructures, these applications are often limited to a specific *Operating System (OS)*, such as Android or Linux. Therefore, a sensor library framework for cross-platform data concentrator applications was developed within the scope of this work. In consequence of using the Node.js runtime, it allows for higher flexibility regarding the *OS* of the data concentrator: The sensor library can be integrated into any software running on one of the three major desktop *OSs* macOS, Widows, and Linux. The central task of the library is the data exchange between the sensor nodes and the data concentrator, from simple connection setup to data streaming. For this purpose, the framework uses *Bluetooth Low Energy (BLE)*, a widely used short-range wireless communication standard for low-power applications. However, due to its modular design, the library can easily be extended with additional protocols, such as ZigBee or Wi-Fi. As demonstrated in this thesis, the sensor library framework can significantly simplify the development of a Node.js-based data concentrator application. Thus, it could support the entire development process – from prototype to final applications – of data concentrators in the patient-centered healthcare system of the future.

Contents

Chapter 1

Introduction

The *Internet of Things (IoT)* has become an essential part of today's world. According to IHS Markit, a leading technology analyst firm, it is considered as one of the top technology trends of our time [IHS19]. The *IoT* is a global application domain that encompasses many different technological and social fields, from personal electronics to industrial machines. Despite the large number of research institutes and companies working in this domain, no uniform definition exists [Min15]. The analyst firm Gartner characterizes the *IoT* as a "network of physical objects that contain embedded technology to communicate and sense or interact with their internal states or the external environment" [Gar19].

One of the biggest drivers of the *IoT* is the rapid progress in the underlying technologies: Improvements in wireless networking technologies, advances in the microprocessor, chip, sensor, and data storage industries and the broader standardization of communication protocols enable ever smaller and cost-effective devices, which collect data almost anywhere at any time [Chu10]. In the year 2016, IHS forecasted an increase of connected devices from 31 billion by 2020 to 75 billion by 2025 [Luc16].

At the same time, increasingly powerful algorithms can process and analyze more data in less time using neural networks and cloud computing. These advancements make the development of new business models in all fields of industry and society possible: Ground sensor data in combination with satellites images allow for individualized, autonomous farming. Millimeter-sized cameras are used to detect sources of illness in the human digestive tract. Billboards are instantly changing their displayed messages based on the comparison of passersby and consumer profiles [Chu10]. Infrared and ultrasound sensors are used by train operators to observe the temperature, sound, and quality of train wheels to prevent derailments [Mur12].

Most of the current applications of the *Internet of Things* can be assigned to one of the

following market segments: Monitoring and controlling the performance of homes and buildings, automotive and transportation systems, automation and self-regulation of industrial processes, sport and health self-tracking, as well as personal environment monitoring [Swa12].

In the recent years, a wide range of portable devices have been developed featuring various sensors such as accelerometers, body temperature sensors, pulse and blood level sensors, or skin response sensors. Thereby, they have access to a person's motions as well as vital signals like heart rate or oxygen level [Dav15]. Wearables offer different functionalities, from simple data collection, data pre-processing, and temporary data storage up to data transmission to internet-connected data concentrators, such as smartphones or remote servers. At the same time, wearable devices are still facing certain challenges, such as short battery life, a low transmission bandwidth or limited computing power. Nevertheless, the rapid progress of wearable technologies is giving the *IoT* a new dimension and thus forming the "*Wearable Internet of Things (WIoT)*" [Hir14].

The healthcare sector is one of the fields that has the highest anticipation of the *WIoT* development [Amf18]. In the light of increasing healthcare costs, aging population, as well as the global prevalence of chronic diseases, a transition from a hospital-centered to a more patient-centered, cost-effective healthcare system is needed. The *WIoT* has the potential to make a substantial contribution to this transformation. Due to their light and unobtrusive design, wearable sensors can be worn for several hours without discomfort. Equipped with appropriate sensor packages, wearable devices are thereby able to track a patient's motions and biomedical signals. This development enables relevant medical data, such as heart rate or blood glucose level, to be captured on a large scale outside the controlled clinical environment. However, in order to derive medical benefit from this data, the wearable devices still have to be connected to the clinical infrastructure. This is made possible by an integrated home health monitoring system, which links the sensor data with the physicians. As shown in Figure 1.1, such an ecosystem consists of the following four basic elements [Kum17]: (1) *Wearable human activity tracking devices* can monitor a patient's movements and biomedical signals. They either transfer the data directly to a nearby data concentrator using a short-range communication protocol such as Bluetooth® or the device first stores the data in its own temporary memory. (2) A *data concentrator* acts as a gateway between the wearable devices and the cloud enabling bidirectional data exchange. As it is often located inside a patient's house, it is also called "edge computing device" [Kum17]. Data concentrators usually run with a light *OS*, such as Android or Raspbian. (3) In a *cloud*, the data is analyzed by intelligent algorithms that allow the physicians to gain knowledge from human activity. (4) The last step of the home health monitoring system is the *clinical integration*. Medical insights from the data in the cloud support physicians in making clinical decisions and

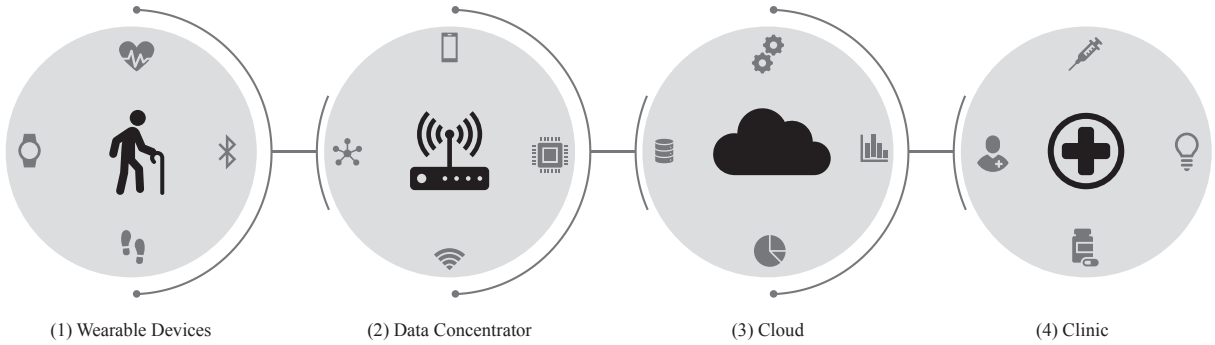

Figure 1.1: A typical home health monitoring infrastructure consists of four elements: wearable devices, data concentrator, cloud, and clinic.

assessing their patients' health when they are at home. [Amf18, Hir14, Kum17, Zha14]

The described ecosystem can add considerable value to a patient-centered healthcare system by enabling remote and individualized health interventions, such as diagnostic control, treatments, or interoperability between patients and physicians. Additionally, significant cost savings in nursing and care compared to traditional hospital-centered healthcare system are possible. The large amount of data generated by body-worn or near-body sensors can furthermore optimize future therapy planning by analyzing recovery cycles of existing patients. [Hir14, Kum17, Omr10]

The present work concentrates on the second element of this ecosystem, the data concentrator. The central goal is to simplify the communication between the edge computing device and the wearable sensors in such a home monitoring system. A number of short-range communication protocols are available for data transmission between these two tiers. The main requirement of this protocol is low energy consumption, which mainly includes ZigBee, *ANT*, *Near Field Communication (NFC)*, or *BLE* [Gup16]. In this work, *BLE* was used for the following reasons. *BLE* was introduced as a part of the Bluetooth 4.0 specification in the year 2010 [Blu10]. It was designed to decrease the power consumption of Bluetooth devices allowing charging cycles of several months or even years [Let17]. The great advantage of *BLE* over other low-power communication technologies is that it can benefit from the wide diffusion of Bluetooth, which is integrated into almost all smartphones, tablets, and laptops. As *BLE* can be easily incorporated into this infrastructure, the number of devices using *BLE* is expected to rise significantly [Gom12].

A number of different applications are running on the data concentrators in existing home monitoring systems. Most of them are limited to a specific *OS*, so they cannot simply be moved to other platforms. For example, Android offers its own official Bluetooth *Application Programming Interface (API)*, allowing easy access to the Bluetooth protocol stack. Access to the Bluetooth

interface on desktop *OSs*, such as Windows, Linux, or MacOS is often much more complex than with Android. Especially in the area of home monitoring, it would be advantageous to have a data concentrator application that runs smoothly on different *OSs*. Such a cross-platform application allows for a higher flexibility regarding the *OS* of the edge computing device.

Built on Google's JavaScript engine from the Chrome browser, V8, Node.js was developed for high-performance, low-memory, and long-running server processes [Her13, Til10]. This allows the development of server-side applications using Node.js. Sandeep Mistry furthermore provides a useful library called *noble*, an *API* that simplifies and standardizes the access to the Bluetooth stack of the device the application is running on. At the moment, it is supported by the main desktop *OSs* macOS, Linux, and Windows [San].

Within the scope of this work, a Node.js-based sensor library framework for a home monitoring system is developed. The underlying idea and structure are based on the Java-based Android sensor library, which was developed by the Machine Learning and Data Analytics Lab of the *Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)* [Ric]. Since this library is limited to Android devices only, a framework supported by multiple *OSs* is required. The aim of the sensor library developed within this work is to simplify the implementation of high-performance, cross-platform applications, which concentrate the biomedical data from wearable sensors. The setup of a frontend on top of this server furthermore enables users and researchers to monitor a patient's motion and biomedical signals via a *Graphical User Interface (GUI)*. Based on this ecosystem, physicians can efficiently build human activity analysis systems by leveraging the techniques of rapid prototyping. It is planned that the library will support other communication standards, such as Bluetooth Classic or Wi-Fi, in addition to *BLE*. Regardless of their used protocols, it should be possible to collect data from all types of sensors.

The work is organized as follows: Chapter 2 provides the technical background relevant for the further course of this work. In a first step, potential wireless communication technologies for home monitoring systems are discussed in more detail. It follows an summary of available *APIs* for the Bluetooth protocol stack as well as an introduction to design patterns in the software development in JavaScript. Chapter 3 describes the development of the Node.js-based sensor library framework where the focus is on the structure and possible applications of the package. A potential use case of the sensor library is given in Chapter 4: a Node.js-based server embedding the library and a *GUI* on top to manage sensor operations at the client-side. The basic functionalities as well as the technology stack of the *GUI* are explained in more depth. A conclusion of the thesis and potential further steps are discussed in Chapter 5.

Chapter 2

Technical Background

2.1 Wireless Communication Technologies for a Home Monitoring System

Wireless communication includes every form of contactless information transmission between two or more devices. Its scope is very broad and covers a wide range of distances: from a few centimeters for mobile payment via *NFC* up to several thousands kilometers for parcel tracking using *Global Positioning System (GPS)*. [Gup16]

This section focuses on the wireless communication in home health monitoring ecosystems. Worn in, on, or around the human body, wearable devices collect vital and physiological data and transfer it to an external data concentrator. For this purpose, they use wireless communication, which is more comfortable and user-friendly than wired communication. A number of different technologies are available for this contactless connection. Existing home monitoring systems use *BLE*, ZigBee, *NFC*, *Adaptive Network Technology (ANT)*, and Wi-Fi. Ali et al. present an implantable glucose monitoring system that utilizes a wireless *BLE* transmitter to send the measured data to a patient's mobile phone [Ali11]. Malhi et al. propose a wearable device which can be worn on the wrist or finger by people at risk [Mal12]. It tracks a patient's physiological parameters, such as heart rate and body temperature, and uses ZigBee modules to transmit these data to a central processing unit. An integrated insole biofeedback and gait analysis system for post stroke patients was developed by Johansson et al. combining several communication standards [Joh11]: The architecture consists of two foot pressure sensors that to send data via the *ANT* protocol to a personal server attached to a user's belt, for example. The personal server in turn is connected to a central home server via Wi-Fi or Bluetooth. Opperman et al. present a

generic *NFC*-enabled remote monitoring ecosystem [Opp11]. For data transfer, the measuring device must be held in the proximity of a mobile phone, both equipped with *NFC* tags.

The examples described show that no standardized wireless communication protocol for home monitoring systems has yet emerged. The reason for this is that each technology has its own advantages and disadvantages. In the following sections, the various communication standards are therefore examined and compared in more detail.

2.1.1 Overview of Short-Range Communication Protocols

The main criteria for communication technologies appropriate for a home monitoring system are energy consumption, range, data rate, and the existing infrastructure. Another important requirement is the interoperability between products from different manufacturers [Omr10]. This compatibility is made possible by various standards, especially the IEEE 802 family of norms. IEEE 802.11 defines *wireless local area network (WLAN)* standards such as Wi-Fi, IEEE 802.15 characterizes *wireless personal area network (WPAN)* protocols including Bluetooth and ZigBee [Zha14]. The *NFC* protocol is defined in the standards ECMA-340 [Eur13] and ISO/IEC 18092 [Int13], whereas *ANT* was developed using its proprietary protocol. [Suh12, Zha14]

Table 2.1 shows the advantages and disadvantages of the communication standards used in existing home monitoring systems. The assessment ranges from *very bad* (– –) to *very good* (+ +). *Bluetooth Classic* refers to the earlier versions of Bluetooth, which was mainly designed for file transmission and audio streaming. It is also known under the name *Bluetooth Basic Rate and Enhanced Data Rate (BR/EDR)*. *BLE 4* covers the versions 4.0, 4.1, and 4.2., which were introduced for low power applications in the years 2010, 2013, and, 2014, respectively. The comparison in Table 2.1 combines the three versions, as the updates did not lead to any relevant changes in the criteria presented. The latest version of *BLE*, Bluetooth 5, which was introduced at the end of 2016, enables a significantly higher range as well as an increased throughput rate.

The main applications of *NFC* include mobile payment, ticketing, and identification by smartphone, chip card, or simple tag [Lan10]. For security reasons, the maximum distance is just 10 cm. This means that data transmission via *NFC* in a home monitoring infrastructure requires a conscious placement of the wearable device near the data concentrator. With a range of at least 30 m, the other wireless communication protocols provide more flexibility and convenience. [Pat17]

The maximum data rate and energy consumption of wireless communication protocols usually correlate. For instance, Wi-Fi was primarily developed for large data transmissions using high-speed throughput. This results in high power consumption, making both Wi-Fi and Bluetooth

Table 2.1: The main criteria of wireless communication protocols for home monitoring systems are range, data rate, power consumption, and availability.

Protocol	Range	Max. Data Rate	Power Consumption	Availability
Bluetooth Classic	+	+ +	−	+ +
***BLE* 4**	+	−	+ +	+ +
***BLE* 5**	+ +	+	+ +	+ +
ZigBee	+	− −	+ +	−
NFC	− −	− −	+	+
ANT	+	− −	+ +	+
Wi-Fi	+	+ +	− −	+ +

Classic unsuitable for low-power operations [Pat17]. The remaining protocols, *BLE*, *ANT*, *NFC*, and ZigBee show significantly lower data rates. However, they still meet the needs of most wearable healthcare systems [Zha14]. At the same time, they can run on tiny coin-cell batteries for months or even years reducing maintenance and running costs [Omr10]. Further details were provided by a detailed study by Dementyev et al., who closely examined the energy consumption of the three technologies *BLE*, *ANT*, and ZigBee in a typical home monitoring infrastructure: *BLE* achieved the lowest power consumption, followed by ZigBee and *ANT* [Dem13].

The existing infrastructure of Wi-Fi, *BR/EDR*, and *BLE* is well developed. Almost every smartphone, laptop, or tablet offers built-in support for these technologies. *NFC* and *ANT* are at least compatible with most of the latest mobile phones. Other communication protocols and devices, require an external dongle. [Nik18]

Comparing different wireless communication protocols for a home monitoring systems shows that *BLE* offers the best compromise between range, data rate, power consumption, and availability. In particular, *BLE* 5 is very suitable for a health monitoring infrastructure. It enables distances of up to 400 m and throughputs of up to 2 Mbps with low energy consumption and high compatibility with most devices. In the year 2009, Continua Health Alliance, an industry coalition of leading healthcare and technology companies, chose *BLE* for wireless health monitoring systems and included it in their Design Guidelines [Blu09, Omr10].

2.1.2 Bluetooth Low Energy

Bluetooth® is a wireless communication standard for transmitting data over short distances managed by the Bluetooth *Special Interest Group (SIG)*. The technology can be divided into two main categories: *BR/EDR* for high data throughput and *BLE*, also called *Bluetooth Smart*, for low-energy applications. Both are operating in the unlicensed *Industrial, Scientific and Medical (ISM)* frequency band from 2.400 to 2.485 GHz. Depending on the supported functionality, there are three types of Bluetooth devices: *BR/EDR Devices* and *Single Mode LE Devices* only support either Bluetooth Classic or *BLE*. As these two technologies feature different channel architectures, they are not compatible with each other. Most smartphones, tablets, and laptops, on the other hand, are *Dual Mode Devices*. They provide both functionalities, which means they are able to communicate with *BR/EDR* and Single Mode LE Devices, even at the same time. [Gup16, Zha14]

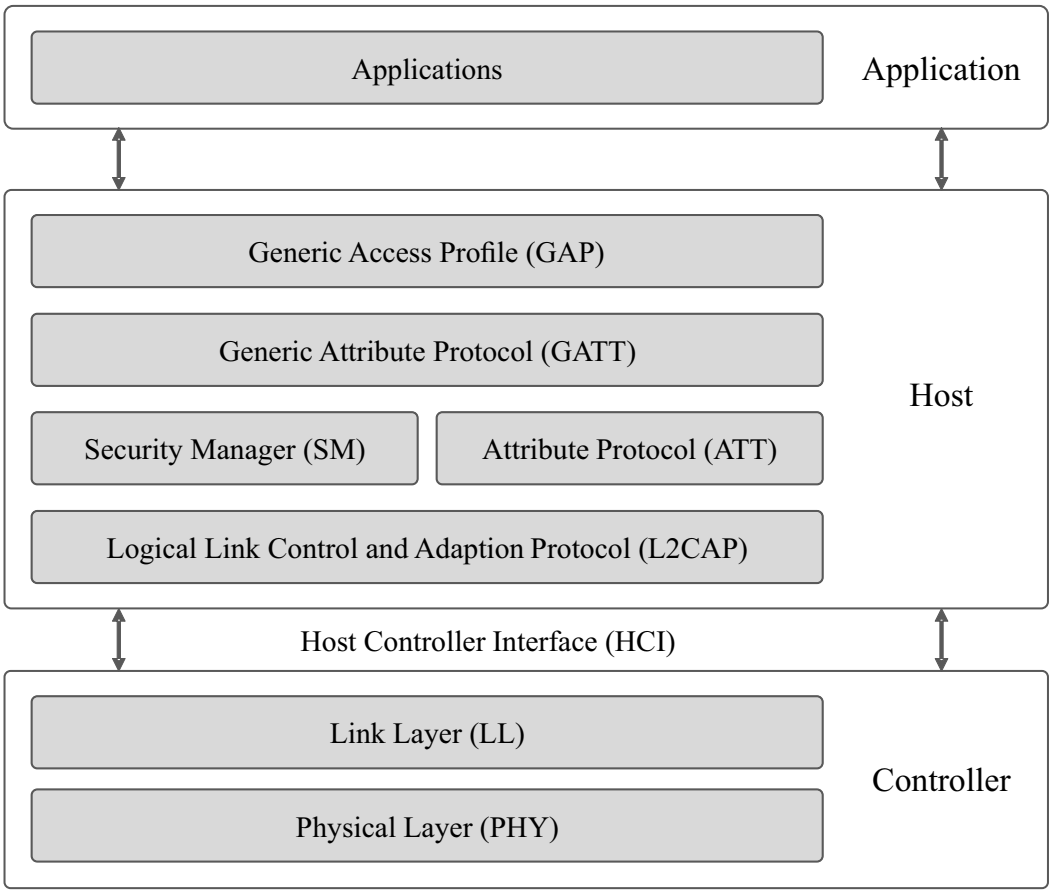

Figure 2.1: The *Bluetooth Low Energy* Protocol Stack consists of three main building blocks: Controller, Host, and Application.

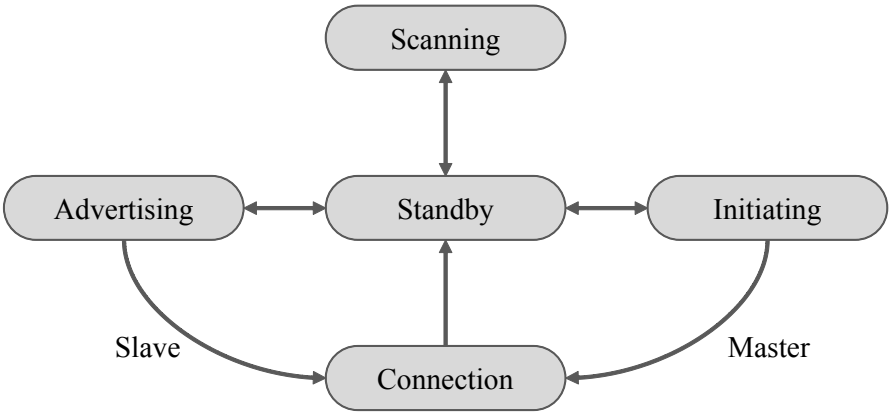

Figure 2.2: The State Machine of *Bluetooth Low Energy* has five states: Standby, Scanning, Advertising, Initiating, and Connection.

As illustrated Figure 2.1, the *BLE* protocol stack consists of a number of layers, which can be assigned to three basic building blocks: *Controller*, *Host*, and *Application*. The controller is usually implemented as a small system-on-chip with an integrated Bluetooth radio, thus acting as the interface to the physical environment. It comprises two layers, the *physical layer* as well as the *link layer*. As the lowest level of the *BLE* stack, the *physical layer* defines the 40 *Radio Frequency (RF)* channels, which are used for device discovery, connection establishment, and data transmission. To reduce interference, it uses an adaptive frequency hopping mechanism that changes the data channels used at a certain interval. [Gom12, Zha14]

The operations of the *link layer* can be explained by the *BLE* state machine. As shown in Figure 2.2, a *BLE* device always has one of the following five states: Standby, Scanning, Advertising, Initiating, or Connection. In order to exchange data between two devices, a connection must be established. For this purpose, a device, called the *master*, initiates the connection and manages it later, and the *slave* accepts the request. A slave can only be connected to one master, while a master can establish multiple connections with different slaves at the same time. This topology is called *star network*. As of Bluetooth version 4.1, it is possible that one *BLE* device can be part of several networks, being master and slave simultaneously. In a typical home monitoring infrastructure the data concentrator acts as the master, the wearable device as the slaves. [Gom12] [Gup16, Mik16, Nik18]

The *Host Controller Interface (HCI)* enables the connection between the hardware chip of the controller and the processor, on which the host and the application are running. For this purpose, it uses a serial interface such as *Universal Asynchronous Receiver/Transmitter (UART)* or USB. The main task of the *Logical Link Control and Adaption Protocol (L2CAP)* is to convert and transfer

received data to the upper layers, *Attribute Protocol (ATT)* and *Security Manager (SM)*. The *SM* is another protocol that generates security keys and distributes them among peers. The next layer of the host is the *ATT*, a simple client/server protocol for the communication between two devices. However, it is not the *ATT*, but the layer on top of it, the *Generic Attribute Profile (GATT)* that determines the client or server role. These roles are independent of the slave or master states of the link layer. A client is able to request read or write access to the data from a server, which then responds accordingly. [Gom12, Mik16, Zha14]

The *GATT* forms the core of the standardization of the *BLE* protocol. It defines a data model and hierarchy, and determines how data is organized and shared between different applications. A *GATT* profile can be described as a use case, a general behavior of a *BLE* device. As shown in Figure 2.3, a *GATT* profile contains multiple services, which cover the behavior of a specific part of a device. A service, in turn, is organized into various characteristics, an attribute type that encapsulates a single value as well as metadata, the descriptors. Services, characteristics, and descriptors can be distinguished by their *universally unique identifiers (UUIDs)*. The Bluetooth *SIG* has defined a number of standard profiles and services including their characteristics and descriptors. An example for a service that is supported by almost all *BLE* devices is the *Battery Service*. It includes the *Battery Level* characteristic which exposes the state of the battery of a device. A key feature of the *GATT* is that it provides a standard mechanism that can deploy new profiles, which allow a quick customization of new applications. [Blu, Let17, Mik16]

The highest level of the host layer, the *Generic Access Profile (GAP)*, determines the way devices discover each other and establish a connection. Therefore, it defines its own set of device roles, the most important of which are the *central* and the *peripheral*. A central device is able to initiate and manage several connections with other devices at the same time, a peripheral, on the other hand, can only connect to one central. Consequently, the central and peripheral roles require that the controller of the device supports the master and slave roles in the link layer, respectively. In a typical remote health monitoring infrastructure the data concentrator would therefore take the role of the central/master, whereas the wearable sensors act as peripherals/slaves. Usually, data is transmitted from the sensors to the data concentrator, which means that *GATT* servers are running on the peripherals and a *GATT* client is operating on the central. At the top of architecture, the actual application is running. It uses the functionalities of the *BLE* stack and defines the necessary services. The next section introduces various *APIs* that can be used to access the *BLE* protocol stack. [Gom12, Men18, Mik16]

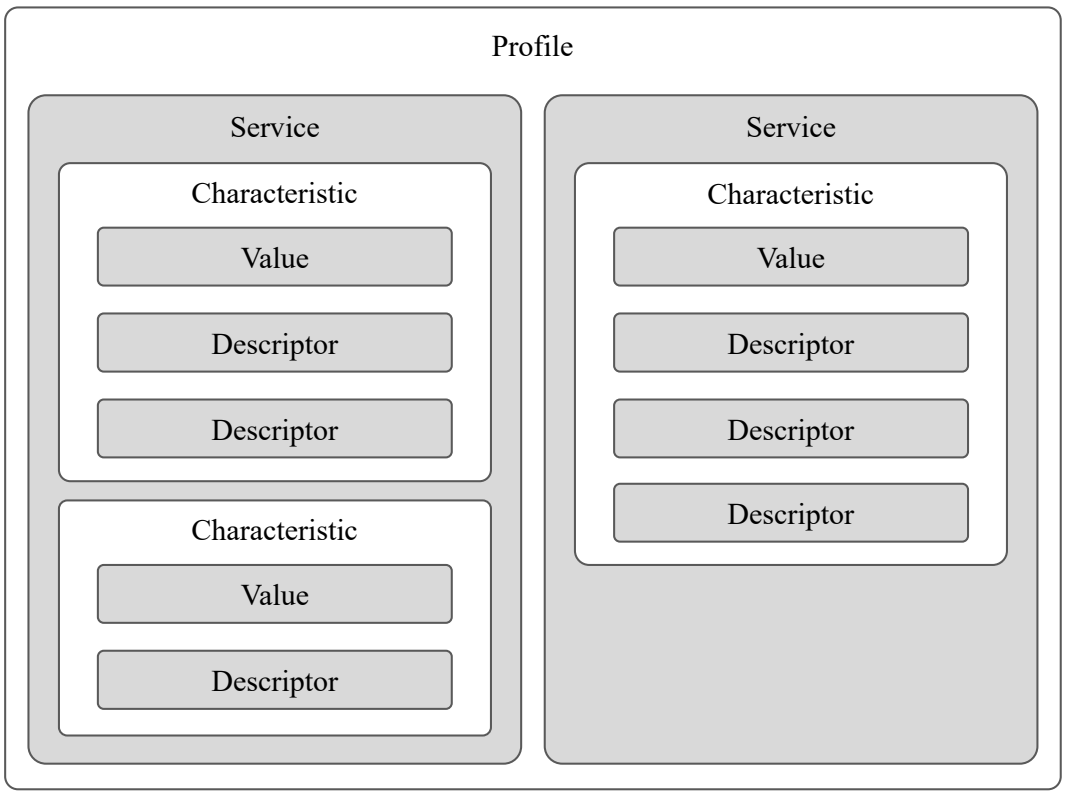

Figure 2.3: The *BLE Generic Attribute Profile* is divided into Services, Characteristics, and Descriptors.

2.2 Bluetooth Low Energy Stack APIs

This work focuses on the application that is running on the data concentrator in a home monitoring infrastructure. The communication between the wearable devices and the data concentrator is established via the *BLE* protocol. The sensor-side tasks of the data concentrator include establishing the connection to the wearable devices, passing commands, or exchanging data. Therefore, the application must be able to access the *BLE* interface of the device it is running on. In order to simplify the interaction of an application with the *BLE* protocol stack, a number of *APIs* are available. As shown in Table 2.2, some of them, such as *bluepy* or *BluetoothKit*, are only supported by one *OS*, others, like *noble* or *Qt*, are cross-platform software.

Qt is a widely used development framework for implementing *GUIs* and cross-platform applications in C++. It is currently supported on Linux and macOS as well as on the mobile platforms Android and iOS [The18]. The *bluepy* framework allows a Python application running on a Linux platform to connect to *BLE* devices [Ian]. For mobile applications on Android or

Table 2.2: *BLE* Stack *APIs* are supported on different *Operating Systems*.

API	Linux	Windows	macOS	iOS	Android
noble	yes	yes	yes	no	no
Qt	yes	no	yes	yes	yes
bluepy	yes	no	no	no	no
Android *API*	no	no	no	no	yes
BluetoothKit	no	no	no	yes	no

Apple's iOS, the official Android *BLE API* or the Swift library *BluetoothKit*, respectively, simplify the communication between a central device and peripherals [And, Ras].

Noble is a Node.js package developed by Sandeep Mistry that enables the communication between a central device and multiple peripherals via *BLE*. Based on Chrome's JavaScript V8 engine provided by Google, Node.js allows for cross-platform applications. This means they can run on different *OSs*. Just as Node.js itself, the noble library is supported on all common desktop *OSs*, such as Linux, macOS, and Microsoft Windows [San]. Node.js also enables the development of full-stack JavaScript applications, which means a single programming language, JavaScript, can be used to build front- and backends, respectively. For these reasons, Node.js and noble are used in this work to develop a sensor library framework for the communication via *BLE*. For this implementation, a number of programming paradigms for software development in JavaScript can be used.

2.3 Programming Paradigms in JavaScript

A programming paradigm can be defined as a methodology, a set of principles for software development. In other words, it is a way to approach a problem and design a solution. Each programming language supports one or more of these paradigms. Therefore, paradigms, such as imperative or object-oriented programming, can be used as classifiers for programming languages. [van09]

Since Node.js is a JavaScript runtime, they both follow the same programming paradigms. JavaScript is a multi-paradigm language, which means it supports several paradigms at the same

time. Two very important ones are also used in the sensor library package developed within this work: *Object-oriented Programming (OOP)* and asynchronous event-driven programming.

2.3.1 Object-Oriented Programming

Programming languages that support *OOP* are based on a collection of *objects* communicating with each other. An object forms the smallest entity in *OOP* and is represented by certain characteristics, called *properties*, and behaviors, called *methods*. An object is always created as an *instance* of a *class*. A class also contains certain attributes and serves as a template for a particular type of object. Similar objects are instantiated from the same class and automatically have its properties and methods.

In *OOP*, there are four key concepts that build on these fundamental ideas: (1) *Encapsulation* describes the idea of wrapping data (i.e. properties) and operations that manipulate that data (i.e. methods) within an object (i.e. a class instance). It is often used for *information hiding*, which means the internal state of an object is protected from direct external interference and is only accessible via the object's methods. In some languages it is possible to define the user access level for certain attributes. For example, a `private` method can only be used within an object, whereas a `public` property can also be accessed from outside. Encapsulation is closely related to (2) *abstraction*. According to this concept, an object reveals only relevant data and capabilities to the environment and hides background implementation and operations. (3) *Inheritance* enables a class to inherit properties and methods of another object, the *parent* class. In this way, certain attributes can be reused in similar classes. (4) The concept of *polymorphism* implies that the same data is processed differently by various objects. For example, a class can override a method that it inherits from its parent class. As a result, the method behaves differently, depending on whether it is called on an instance of the parent or child class. [Ste08, Zak09]

The *OOP* paradigm is mainly used for problems with a large number of related data abstractions, which are organized in a hierarchical structure [van09]. A common way to illustrate such a hierarchy is to use *Unified Modeling Language (UML)* class diagrams [Obj]. An example of a class diagram is shown in Figure 2.4.

In this case, `Sensor` is the parent class providing the properties `id`, `name`, and `initialized` as well as the method `connect()`. The two classes `PulseSensor` and `TemperatureSensor` inherit from the `Sensor` class, they *extend* the base class as well as its functionalities. This means an instance of the *PulseSensor* class can use its own method `scanPulse()` as well as the inherited method `connect()`. The characters in front of the attributes indicate their access level: `public` (+), `private` (–), and `protected` (#). The latter implies

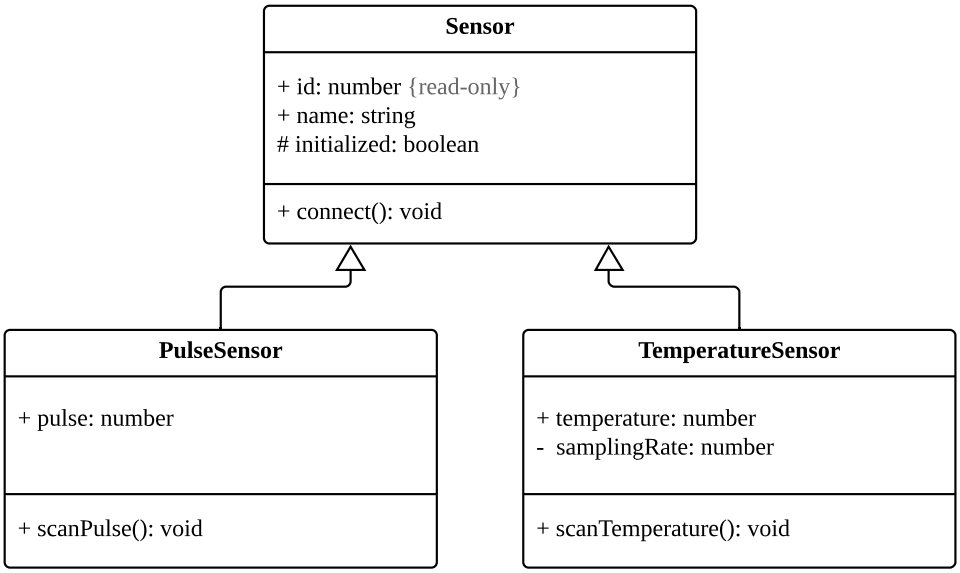

Figure 2.4: A *Unified Modeling Language* class diagram illustrates a hierarchical class structure in *Object-oriented Programming*.

that a property or method is only accessible within a class itself or its inherited classes.

Another data structure that is commonly used for abstraction and encapsulation in *OOP* is an `interface`. First of all, an interface is a collection of properties and methods that collectively describe a particular type or functionality of an object. Each class that implements a particular interface is required to also provide all its properties and methods. Thus, classes that implement the same interface can be handled in the same way. Therefore, interfaces are often used to define compatibilities between classes that do not inherit from each other. This facilitates the development of applications based on them.

JavaScript supports *prototype-based OOP*. Compared to *class-based* languages, such as Java or C++, JavaScript does not provide a real class implementation. However, the JavaScript version of 2015, *European Computer Manufacturers Association Script (ECMAScript)* 6, introduced a *class* keyword. But since objects are the only constructs that exist in JavaScript, classes in JavaScript are technically pure objects. The concept of inheritance, though, is still supported in JavaScript: Each object has a private property that contains a link to a parent object, called its *prototype*. [Guh10]

One drawback of using JavaScript for *OOP* is that it does not provide interface support. Additionally, it is not possible to define the access level of properties or methods using keywords such as public or private. An alternative to JavaScript is TypeScript, which is developed and

maintained by Microsoft. As it is a strict superset of *ECMAScript* 6, TypeScript is fully compatible with JavaScript and all its libraries. However, by adding interfaces, enumerations, generics, and a typing system, it can simplify the implementation of large-scale JavaScript applications [Bie14]. Furthermore, TypeScript allows using the access modifiers `public`, `protected`, and `private`. However, it is compiled to JavaScript, which does not support this way of information hiding[1]. Access modifiers in TypeScript are hence just a syntactic structure allowing programmers to develop cleaner and more transparent software according to the *OOP* paradigm. Since TypeScript nevertheless offers many advantages over JavaScript, it is used in this work to develop the sensor library framework.

As an example, the previous `Sensor`, `PulseSensor`, and `TemperatureSensor` classes are programmed in TypeScript in Listing 2.1. As the `Sensor` class implements the `BasicSensorInfo` interface, it is enforced to contain the attributes `id`, `text`, and `connect()`.

```
1  interface BasicSensorInfo {
2    id: number;
3    name: string;
4    connect(): void;
5  }
6  class Sensor implements SensorBasicInformation {
7    protected initialized: boolean;
8    public id: number;
9    public name: string;
10   public connect(): void {
11     // connects to sensor
12   }
13 }
14 class PulseSensor extends Sensor {
15   public pulse: number;
16   public scanPulse(): void {
17     // scans pulse
18   }
19 }
20
```

[1] During compilation, TypeScript is converted to JavaScript. However, since JavaScript and TypeScript have a very similar level of abstraction, TypeScript is actually not *compiled*, but *transpiled* to JavaScript. But for the sake of simplicity, the term *compile* is used in this paper.

```
21 class TemperatureSensor extends Sensor {
22   private samplingRate: number;
23   public temperature: number;
24   public scanTemperature(): void {
25     // scans temperature
26   }
27 }
```

Listing 2.1: TypeScript allows to build a hierarchical structure using classes and interfaces.

2.3.2 Asynchronous Event-Driven Programming

Another programming paradigm supported by JavaScript is *asynchronous event-driven programming*. In general, it is a concept in which the program flow is determined by events, such as messages from other programs, sensor inputs, or user actions. It is therefore often used in multi-threaded applications. Each incoming event is added to an event or task queue. As soon as one thread (e.g. the main thread or a thread responsible for data processing) is available, it removes the first element from the queue and calls the corresponding task, its event handler. In an asynchronous architecture, these calls are non-blocking, which means several operations can be started an the same time. Thereby, the order in which event handlers are executed is not set. [Jam13]

JavaScript is a single-threaded, event-driven programming language with an asynchronous control flow. This means, the JavaScript runtime itself can only execute a single task at a time. However, most JavaScript applications use various web or Node.js *APIs* to interact with programs outside of the own processor. Usually, these tasks include time-intensive operations, such as database requests or network communication with other devices. In order to optimize the control flow, both of the important JavaScript platforms, the browser and Node.js, make these kind of operations asynchronous. This allows multiple tasks to be executed concurrently without blocking the main thread. Each time an asynchronous function is called, its *callback function* is added to the task queue. In JavaScript, a callback function is an operation that is invoked as soon as the main function has completed. [Hav18]

As shown in Figure 2.5, the callback queue is managed by the *event loop*. It manages all incoming requests and takes care that the JavaScript thread never blocks. For example, when an application is waiting for an *API* to return a particular request, it can still handle other tasks, such as user inputs, during that time. [Hav18]

The main advantage of an asynchronous event-driven program flow is that the main thread is

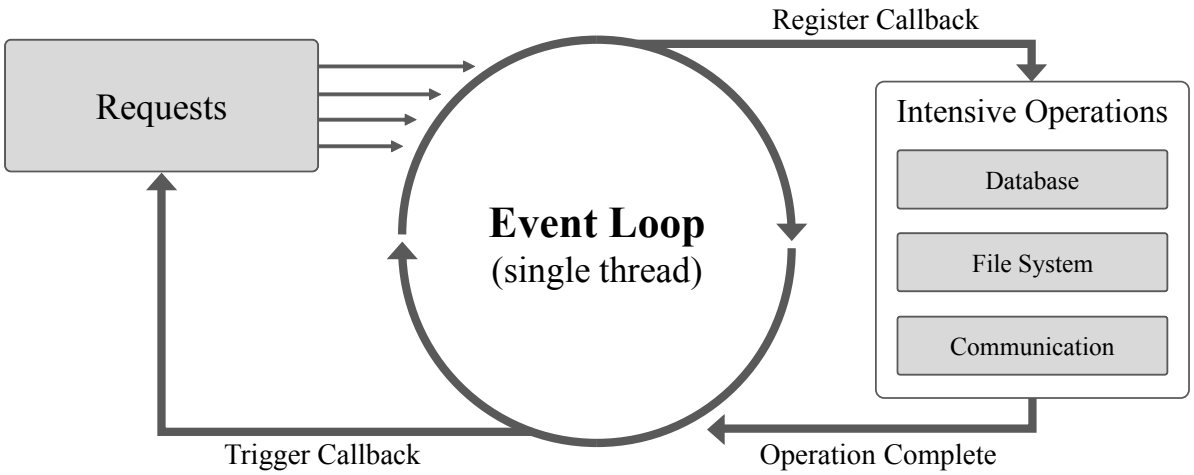

Figure 2.5: JavaScript's asynchronous control flow allows the concurrent execution of multiple operations.

never blocked by intensive operations. The drawback is that it is not possible to determine the exact start and end times of a task. However, in many applications it is necessary that certain operations are executed in a specific order. One way to do this is to pass the function that has to be executed second as the callback of the first function. In large applications with many functions based on one another, this can quickly become complex. The keywords `async/await`, which were introduced with *ECMAScript* 8 in the year 2017, represent a clearer alternative. An intensive function, such as a database query, can be marked with the keyword `async`. Thereby, it returns a `promise`, which is resolved as soon as the corresponding request is completed. In a JavaScript program, the call of such an `async` method can be marked with `await`. As a consequence, the control flow stops at this point and waits for the function to resolve its promise before continuing. This makes it possible to call several tasks in a specific order.

Chapter 3

System Description

This chapter focuses on the development of a sensor library framework for a home monitoring system. The core idea of the library is to simplify the implementation of a data concentrator application. In a typical remote health monitoring infrastructure (cf. Figure 1.1), such an application represents the link between wearable sensors and a cloud. On the sensor side, the main tasks of the data concentrator include connection establishment, passing commands to the wearables, as well as receiving measured values from the devices. On the other side, the data must be loaded into the cloud. The framework developed as the result of this work concentrates on the sensor side of this infrastructure. For the communication with wearable devices, the library currently only supports a connection via *BLE*. In this case, the data concentrator acts as the central device, the wearables sensors as peripherals.

The framework itself is developed in TypeScript, but is provided to the user in both, a TypeScript and a compiled JavaScript version. As a result, cross-platform, Node.js-based data concentrator applications that import the sensor library can be implemented in both languages. As explained in the next sections, it follows the *OOP* and asynchronous event-driven programming paradigms.

3.1 Package Structure

The sensor library framework is structured according to the concepts of *OOP*. It defines several classes, from which objects with the same attributes can be created as instances. Thereby, the library distinguishes between two main class types: (1) sensor classes for particular sensor groups as well as (2) *BLE* protocol specific classes.

(1) Classes for Sensor Types

The core idea of the library is that each wearable device is represented by a separate object, in which all its attributes are encapsulated. Since a home monitoring infrastructure typically contains multiple identical sensors with the same characteristics and behaviors, the library provides sensor classes. As a collection of properties and methods, a sensor class covers all features and capabilities of a specific sensor type. A separate instance of this sensor class is created for each device of this type.

The sensor library package is initially intended for home monitoring systems using a *BLE*-enabled sensor group called *NilsPod Sensors*. These are portable sensors, developed by Portabiles, a spin-off of the Machine Learning and Data Analytics Lab of the *FAU* [Por]. These sensor nodes are able to record acceleration and angular velocity using an *Inertial Measurement Unit (IMU)*. They record human movements for applications in the fields of healthcare, mobility, and sports. One particular use case of the NilsPod sensors is within a mobile system for sensor-based medical gait analysis. For different applications, various types of NilsPod sensors with special features and capabilities have been developed.

For each of these NilsPod sensor models, the library provides a separate sensor class. However, they are not completely different, but share most of their attributes. For this reason, the *OOP* paradigm of *inheritance* was leveraged by introducing an `AbstractNilsPodSensor` base class that implements shared features of all NilsPod derivatives. It acts as a generic sensor class, from which all other NilsPod sensor classes inherit. Even below this base class, there are sensor types with the same characteristics and capabilities, which requires further abstract classes. This leads to a tree-like, hierarchical class structure.

It is planned that the sensor library framework will be extended by further sensor groups or protocol standards. In order to facilitate the integration of new *BLE* sensor classes into the library, a generic `BleSensor` class, a superclass of `AbstractNilsPodSensor` is introduced. It encompasses properties and methods that are required by any *BLE* device: basic attributes, such as name or address, as well as all functions necessary for establishing a connection to the peripheral. In this way, it can be used as the parent class for all new *BLE* sensor classes.

The entire sensor hierarchy of the NilsPod sensors is shown in a reduced version of a *UML* class diagram in Figure 3.1. The different elements of the NilsPod class tree are examined in more detail in Section 3.1.1. Supporting the concepts of *OOP*, this architecture allows a high degree of modularity and expandability. New sensor classes can be added either directly in the library or in the application that is importing the sensor framework.

Each of the NilsPod-specific sensor classes overrides the static property `nameFilterSearch-`

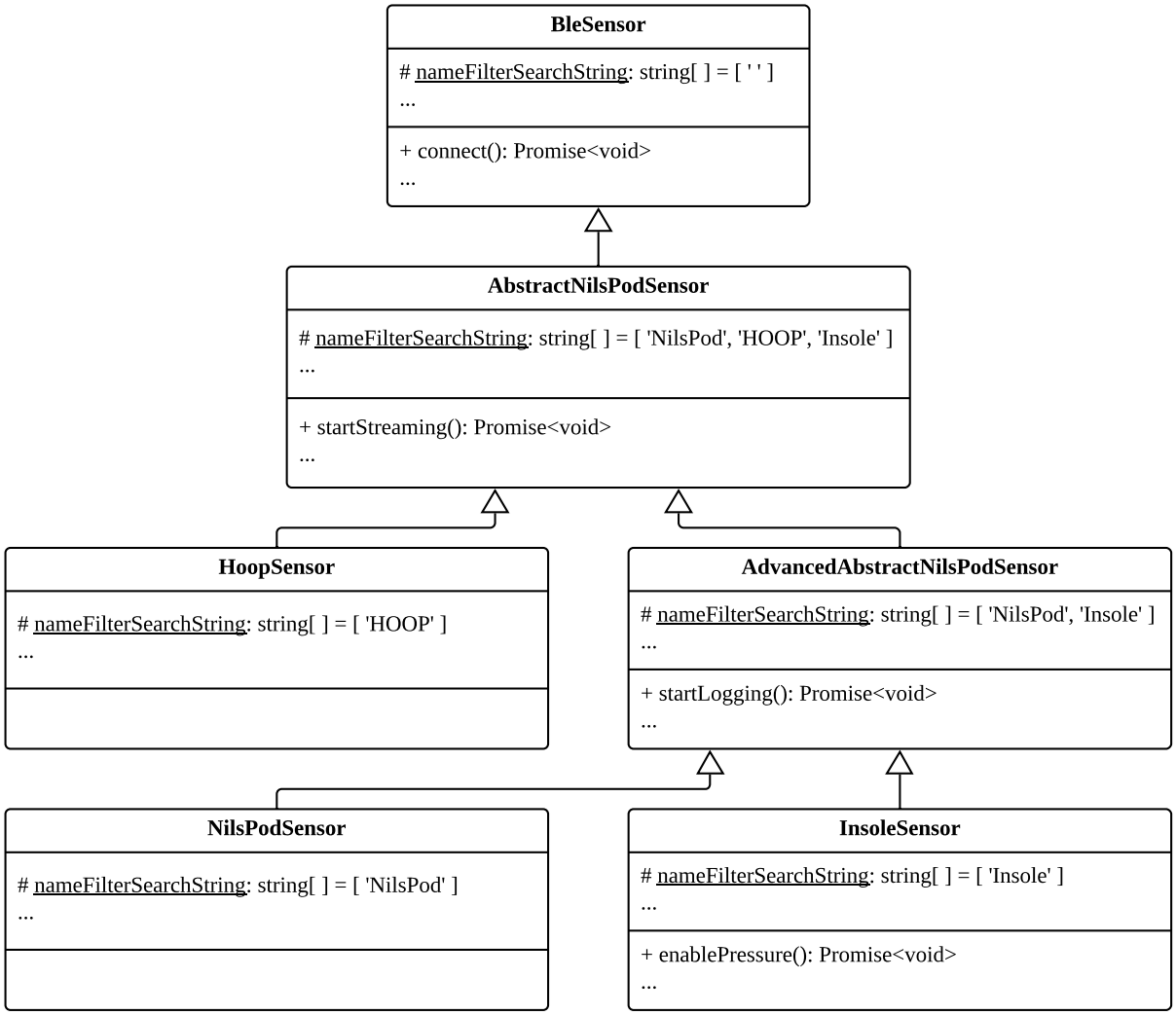

Figure 3.1: Supporting the concepts of *OOP*, the NilsPod sensor classes are structured in a hierarchical way.

`String` of the `BleSensor` class by its own list of names. In contrast to *instance* variables, `static` attributes are used to describe properties or methods of the class itself, not of the instances. The `nameFilterSearchString` attribute is used to assign a newly discovered device to one of the classes based on its name. The exact functioning of this identification process is further described in Section 3.1.2.

Depending on their features and capabilities, some of the sensor classes implement specific interfaces. Especially with regard to the further expansion of the library, this ensures that classes implementing the same interface can be treated equally. But in contrast to classes, TypeScript interfaces only exist at compile time and are not integrated into the resulting JavaScript. This

means, if the application using the sensor framework is developed in JavaScript, the interfaces can not be employed outside the library.

(2) BLE Protocol Specific Classes

As mentioned earlier, different NilsPod derivatives can contain specific sensor packages. Depending on this hardware configuration, they process and transmit different data packages. For this reason, application-specific *BLE GATTs* were designed. Depending on the *GATT* of their *BLE* stack, peripherals offer different services, characteristics, and descriptors. Since all services, regardless of their functions, have similar attributes and capabilities, the library provides a separate class for this *GATT*-specific data type. The same applies for characteristics and descriptors. The `BleService`, `BleCharacteristic`, and `BleDescriptor` classes encapsulate their respective properties and methods.

As soon as a peripheral is connected to the central device, its services can be discovered. For each service detected, an instance of the `BleService` class is created. In the same way, `BleCharacteristics` and `BleDescriptors` are instantiated.

The `BleService`, `BleCharacteristic`, and `BleDescriptor` classes are illustrated in Figure 3.2 as *UML* class diagrams. Each of them provides read-only accessors for its name, type, and *UUID*. Furthermore, the `BleService` and `BleCharacteristic` classes have a list of their characteristics and descriptors, respectively. During the discovering phase, these arrays are filled with instances of the corresponding lower-level data type. In addition, the `BleCharacteristic` class has a getter for `properties` containing a list of access rights for the value of the characteristic. Some characteristics are read-only, others are writable or permit subscription. A characteristic is subscribed in order to receive a notification each time its value changes.

In order to read or write a value of a characteristic, a `Buffer` is received or sent, respectively. A `Buffer` is a Node.js-specific data type to exchange data in serial connections as byte streams. In this sensor package, `Buffers` are used to transmit data packets between the central *BLE* device and the peripherals. The *BLE* protocol stack limits the maximum data size of a packet to 20 bytes. In the case of the NilsPod sensors, this size covers most of their data transmission operations, such as receiving control commands or streaming data samples. However, large data sets must be divided into blocks of 20 bytes and sent individually.

The *BLE* specific classes provide a variety of methods that take care of the communication between the data concentrator and the sensors. For example, the `BleService` class contains the method `discoverCharacteristics`, which detects the characteristics supported by a

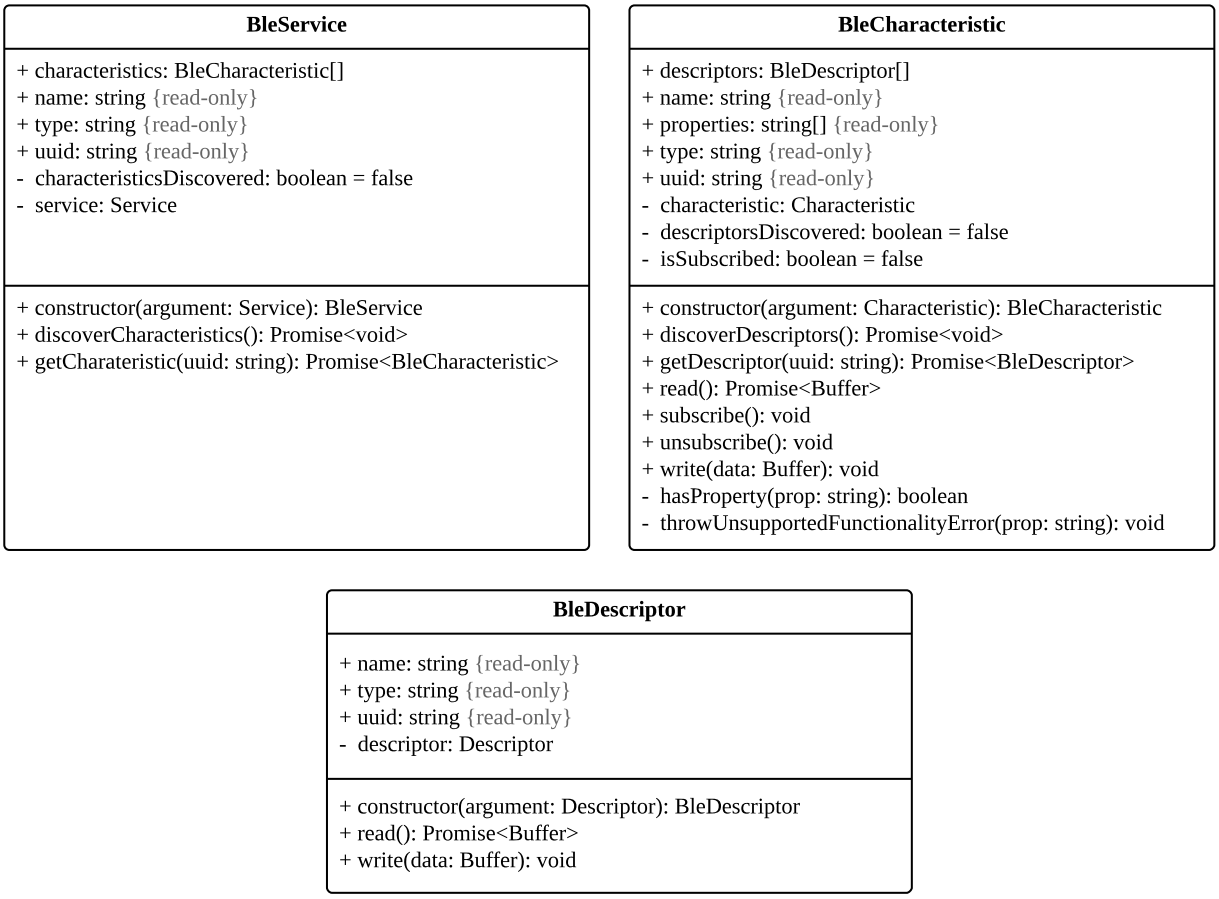

Figure 3.2: The `BleService`, `BleCharacteristic`, and `BleDescriptor` classes encapsulate properties and methods that are specific for their *GATT* data type.

service. The `write` method of the `BleCharacteristic` class allows to manipulate the value of a characteristic. Since these actions have to access the *BLE* protocol stack, they are usually time-intensive. More importantly, it is normally difficult to determine in advance when exactly these procedures will be completed. Therefore, these methods usually return a promise, which is resolved as soon as their corresponding task is completed. Such a promise can either contain a particular data type (e.g. `Promise⟨Buffer⟩`) or no value (`Promise⟨void⟩`).

Many procedures of the sensor library framework build on each other. For these operations, it must be ensured that the previous action has been successfully completed. An example of this is the `getCharacteristic` method of the `BleService` class, shown in Listing 3.1. It takes a *UUID* and returns a promise with the corresponding `BleCharacteristic` instance. The method uses the keyword `await` to ensure that the characteristics are first discovered before searching for the passed *UUID*.

```
1  class BleService {
2    ...
3    public async getCharacteristic(uuid: string): Promise<
         BleCharacteristic> {
4
5      await this.discoverCharacteristics().catch((error: Error) => {
6        throw error;
7      });
8
9      return this.characteristics.find(
10       (characteristic: BleCharacteristic): boolean => {
11         return characteristic.uuid === uuid;
12       }
13     );
14   }
15   ...
16 }
```

Listing 3.1: The `getCharacteristic` method uses the `await` keyword to ensure a synchronous control flow.

Another *BLE* specific class is the `BleScanner` class. As with the `BleService`, `BleCharacteristic`, and `BleDescriptor` classes, it includes the noble *API* in order to access the *BLE* protocol stack of the central device. Each application that employs the sensor library package uses internally an instance of the `BleScanner` class for initiating and managing the scanning process. The `BleScanner` class inherits from the `EventEmitter` class provided by Node.js' built-in *events* module. Node.js core *API* itself is based on an asynchronous event-driven architecture, in which objects, called *emitters*, throw events that cause functions, *listeners*, to be called. The `BleSensor` class extends the `EventEmitter` class to enable its instance to emit `discoverSensor`, `updateRSSI`, and `updateManufacturerData` events. Possible applications of these custom events are described in Section 3.2.

3.1.1 Sensor Classes and Interfaces

This section takes a closer look at the sensor classes. The focus is on the differences between the device types they represent. Their potential applications are further presented in Section 3.2.

BleSensor Class

`BleSensor` is a generic sensor class for all *BLE*-enabled devices. Its main purpose is to serve as a parent class for specific sensor classes. For this reason, it encompasses the basic properties and methods of a *BLE* device. These attributes are shown in the *UML* class diagram in Figure 3.3.

As long as a peripheral is not connected to a central *BLE* device, it continuously broadcasts `advertisement` packets, along with a set of metadata. These attributes, such as the `ID`, `address`, `name`, `state`, `addressType`, the *`Received Signal Strength Indicator (RSSI)`*, and the `isConnectable` flag are directly accessible through getters. Another important attribute is the `peripheral`, which represents an instance for a peripheral from the *noble API* and is mainly used for internal operations.

Following the *OOP* concept of polymorphism, the `BleSensor` class has some attributes that can or should be overridden by its derived classes. For example, the property `sensorType` contains the class name itself, in this case `BleSensor`. It is automatically described by the prototype name during the instantiation of a device object. However, other attributes, such as `hasBatteryMeasurement` or `sensorConfig` must be manually overridden in inherited classes. The sensor configuration property provides device settings, such as the sampling frequency or enabled sensor packages.

The protected properties `BleSensorServiceUuids` and `BleSensorCharacteristicUuids` are read-only string-dictionaries. In programming, a *dictionary* is a data structure for storing a group of objects. As shown in Listing 3.2, each object consists of a *key*, for example `batteryService`, and a *value*, in this case a *string* providing its *UUID*, *180f*. These two dictionaries contain the *UUIDs* of services and characteristics, which are standardized by the Bluetooth *SIG* and are supported by most of the available *BLE* peripherals. They are internally used to interact with the corresponding services and characteristics, respectively.

```
1 protected BleSensorServiceUuids: ReadonlyDictionary<string> = {
2   genericAccess: '1800',
3   genericAttribute: '1801',
4   batteryService: '180f',
5   deviceInformation: '180a'
6 };
```

Listing 3.2: The `BleSensorServiceUuids` property is a read-only dictionary providing the *UUIDs* of standardized *GATT* services

Figure 3.3: The `BleSensor` is a generic sensor class for all *BLE*-enabled devices.

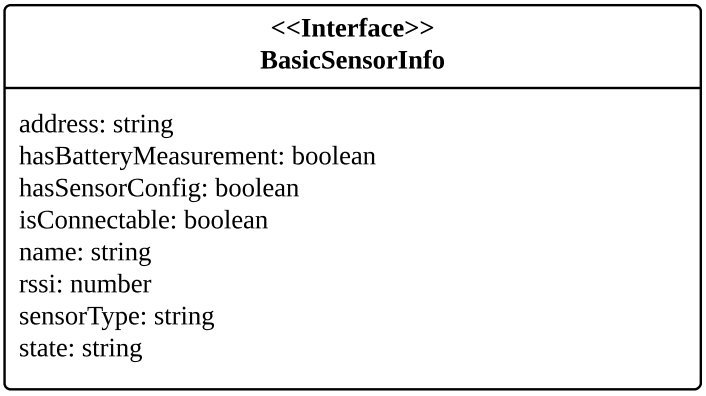

Figure 3.4: The `BasicSensorInfo` covers basic attributes that should be provided by any sensor device, regardless of its used protocol standard.

Some methods of the `BleSensor` class include preparatory or supportive procedures for more complex actions of the derived sensor classes. For example, the protected method `discoverServicesCharacteristicsAndDescriptors` must be successfully completed before any operation with one of these data types can be used.

As with the `BleScanner` class, the `BleSensor` also extends the `EventEmitter`. This allows instances of the `BleSensor` class or its derived classes to emit custom events. As explained in Section 3.2, the NilsPod-specific sensor classes use this capability to to throw `data`, `warning`, and `error` events.

The `BleSensor` class implements the interface `BasicSensorInfo`, whose *UML* diagram is shown in Figure 3.4. It contains basic properties all wearable devices, regardless of their supported protocol standard, should provide. The underlying idea of the `BasicSensorInfo` interface is that it should be implemented by all generic sensor classes of this library. For example, if the framework is extended by an abstract `ZigBeeSensor` class, this parent class should also implement this interface. Thus, all sensor instances can be handled in the same way.

NilsPod Sensor Classes

The `BleSensor` class serves as parent class for NilsPod-specific sensor classes. Within the scope of this work, sensor classes for the three most important device types are developed. This includes the Hoop, classic NilsPod, and Insole sensor, which are depicted in Figure 3.5.

The three sensor models can be divided into two groups. On the one hand, there is the Hoop sensor, a basic sensor node that can be worn around the wrist or ankle. It has a built-in *IMU*, consisting of an accelerometer and a gyroscope. Since Hoop sensors do not have a flash memory,

Figure 3.5: The three NilsPod sensor types, Insole (back), Hoop, and classic NilsPod sensor (front) record human movements for healthcare, mobility, and sports applications.

they are not able to log data, but need to directly stream it to a data concentrator.

On the other hand, the classic NilsPod and the Insole sensor form a group that can be called *advanced* NilsPod sensors. The classic NilsPod sensor is equipped with a barometer and an *IMU*. The Insole sensor has an additional three pressure sensors distributed in the sole. In contrast to the Hoop sensor, both have an internal flash memory of 2 or 4 Gbit, in which they can temporarily store the captured data. In this way, they are able to to record data without being constantly connected to a data concentrator. The 2 Gbit memory allows an Insole sensor with all its internal sensors enabled and a sampling frequency of 204.8 Hz to log data for approximately 16 hours. In contrast, a classic NilsPod sensor that only collects *IMU* and barometer data can record up to 19 hours. Each time a sensor starts logging, it stores the data in a separate session in their memory. If they are within the Bluetooth range of a data concentrator, they can transmit the recorded sessions to the computing device via *BLE*. The specifications of the three NilsPod sensor models are summarized in Table 3.1.

The second technical advantage of the classic NilsPod and Insole sensors compared to the Hoop sensor is related to their different *GATTs*. In addition to the standard *BLE* services *Generic Access*, *Generic Attribute*, *Battery Service*, and *Device Information* (cf. Listing 3.2), all three

Table 3.1: The NilsPod sensor models differ in their specifications, capabilities, and implemented interfaces.

	Hoop Sensor	**NilsPod Sensor**	**Insole Sensor**
Parent Class	AbstractNilsPodSensor	AdvancedAbstractNilsPodSensor	
Streamable	yes	yes	yes
Loggable	no	yes	yes
Modifiable	no	yes	yes
Available Sensors	*IMU*	*IMU*, Barometer	*IMU*, Barometer, Pressure
Implemented Interfaces	BasicSensorInfo Streaming Resettable	BasicSensorInfo Streaming Resettable Logging SensorModification	

sensor types provide the *UART service*. It is used to exchange data between the peripheral and the central device. Furthermore, the *GATT* of an advanced NilsPod sensor contains the *configuration service*. It contains several characteristics, each representing different internal configurations of a device, such as the sampling frequency or enabled sensors packages. By changing the values of these characteristics it is possible to adjust the corresponding internal settings. All classic NilsPod and Insole sensors have a default configuration, which is stored internally in the devices. Table 3.2 gives an overview of the standard and alternative values. For example, by default, only the *IMU* sensor is switched on. However, by changing the value of the characteristic responsible for the internal sensor composition, each sensor can be enabled and disabled individually.

The sensor library package provides separate classes for Hoop, classic NilsPod, and Insole sensors. However, as explained above, all three sensors share many features and capabilities. For this reason, the advantages of *OOP* were leveraged to extract these shared properties and methods in common parent classes (`AbstractNilsPodSensor` and `AdvancedAbstractNilsPodSensor`, respectively). As shown in the class tree in Figure 3.1, the `HoopSensor` and the `AdvancedAbstractNilsPodSensor` classes extend the `AbstractNilsPodSensor`, whereas `NilsPodSensor` and `InsoleSensor` classes inherit from the `AdvancedAbstractNilsPodSensor`. Both of the parent classes are pictured in Figure 3.6 and Fig-

Table 3.2: The configuration of classic NilsPod and Insole sensors consists of different settings, which have default values, but are modifiable.

Property	Default Value	Additional/Alternative Values
Enabled Sensors	*IMU*	Barometer, Pressure
Sampling Frequency	102.4 Hz	204.8, 256, 512, 1024
Sensor Position	not defined	Left Foot, Right Foot, Hip, Wrist, Chest
Automatic Logging	disabled	enabled
Motion Interrupt	disabled	enabled

ure A.1/Figure A.2, respectively.

During the initialization process of a NilsPod sensor, all its services, characteristics, and descriptors are discovered. Some of them are repeatedly used by different methods. In order to simplify their access, they are stored in separate properties. For the Hoop sensors, this only applies for the *UART* service, classic NilsPod and Insole sensors additionally save the configuration service. Their necessary *UUIDs* are provided by read-only dictionaries from the parent classes. The initialization of the advanced NilsPod sensors cover another step: the verification of the current sensor configuration and its assignment to the `sensorConfig` variable. For this purpose, the values of the characteristics of the configuration service are read.

By default, an advanced NilsPod sensor streams data samples of 14 bytes, which bundle the measured values of all sensors together with a 2-byte sample counter that increments with each transmitted sample. The main purpose of the counter is to check for missed data packets. As shown in Table 3.3, each axis of the gyroscope and accelerometer occupies 2 bytes of the data packet. In case of the classic NilsPod and Insole sensor, switching the other sensors, barometer or pressure sensors, on or off results in bigger or smaller data packets. The data sample can be extended by the barometer (2 bytes) or the pressure sensors (1 byte for each measurement point, 3 bytes in total) to a maximum of 16 or 19 bytes, respectively.

If the advanced NilsPod sensors are logging, they use almost the same data string, but instead of transmitting them directly to the central device, they store them in their internal flash memory. In contrast to streaming, however, a logging data sample contains a 4-byte counter representing the number of stored data packets since midnight of the start date of the recording. In combination

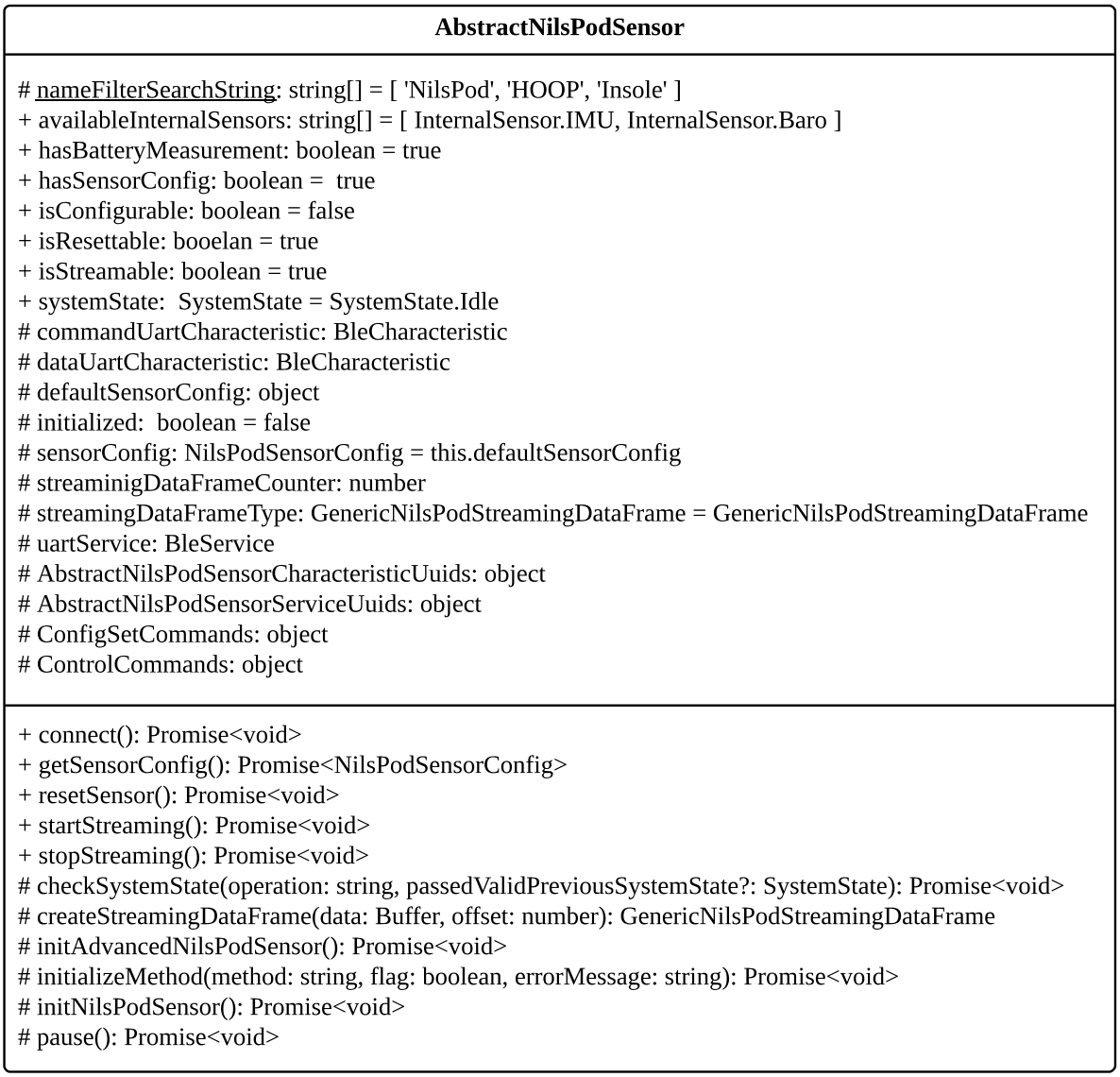

Figure 3.6: The `AbstractNilsPodSensor` class provides shared properties and methods of all NilsPod sensors.

Table 3.3: When streaming, a NilsPod sensor transmits data samples of 14 byte consisting of the gyroscope and accelerometer data as well as a 2-byte counter, by default.

bytes	**0-1**	**2-3**	**4-5**	**6-7**	**8-9**	**10-11**	**12-13**
data	GyroX	GyroY	GyroZ	AccelX	AccelY	AccelZ	Counter

with the start time of the logging session obtained from the session header and the sampling frequency, the exact timestamp of each sample can be determined.

By default, the sampling frequency of the internal sensor packages of a classic NilsPod or Insole sensor is 102.4 Hz, and that of the Hoop sensors 200 Hz. For each sampling interval, the sensor data is bundled and transmitted to the central device (streaming) or stored in the flash memory (logging). For the advanced NilsPod sensors, it is possible to change this frequency to one of the following values: 204.8, 256, 512, or 1024 Hz.

In streaming mode, each data sample is transmitted to the data concentrator as a `Buffer` consisting of consecutive hexadecimal numbers. This data format is very impractical for further processing. Therefore, the library converts it into an easy-to-handle data frame, which is then made available to the user. Since the three NilsPod sensor models can have different sensor configurations, their sensor classes use specific data frames. These are stored in a separate property, the `streamingDataFrameType`. The different data frames are structured similar to the sensor classes. Since they share certain data (e.g. sample counter or *IMU* readings), they also form a hierarchical class structure. The hierarchy of the streaming data frames is illustrated in *UML* diagrams in Figure A.3. Each time a data sample is received by the central device, an instance of the corresponding data frame class is generated. Listing 3.3 shows an example of an `InsoleStreamingDataFrame` instance.

```
InsoleStreamingDataFrame {
  localCounter: 1,
  gyro: { x: -1, y: 0, z: 11 },
  accel: { x: -102, y: 71, z: 2003 },
  barometer: 980.45,
  fsr: { '0': 0, '1': 0, '2': 0 }
}
```

Listing 3.3: The `InsoleStreamingDataFrame` provides the sensor readings in form of an easy-to-handle object.

All three NilsPod sensor models are equipped with a *BMI160*, an *IMU* produced by Bosch Sensortec GmbH [Bos18]. It consists of a 3D accelerometer and gyroscope. The orientation of the axes in a classic NilsPod sensor as well as the positioning of the *IMU* in the Insole sensor is shown in Figure 3.7 (left, center). For each axis, the data frames provide the raw gyroscope and accelerometer data. Using the calibration factors from the data sheet they can be converted to physical units, i.e. g for the accelerometer and $°/s$ (degrees per second) for the gyroscope.

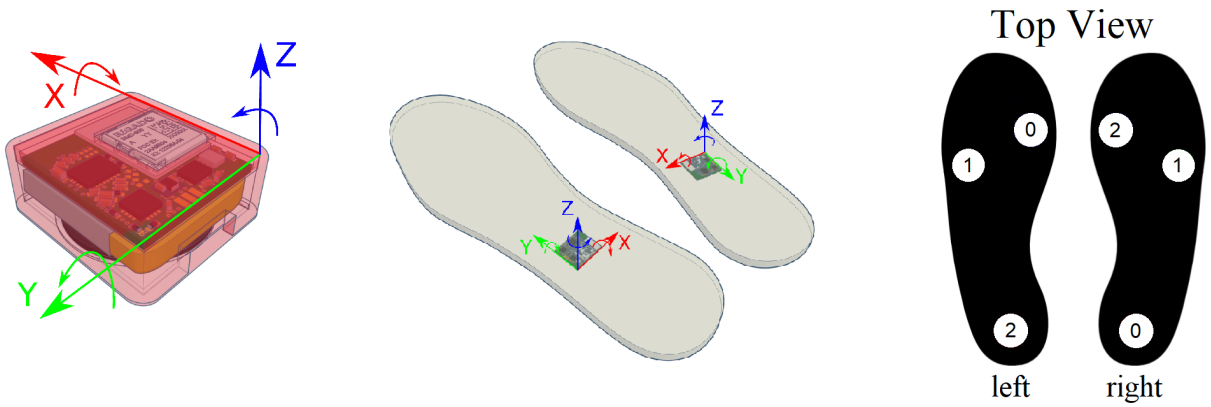

Figure 3.7: The NilsPod sensors are equipped with *IMUs*, which measure angular velocity and acceleration in X, Y, Z direction (left, center). In addition, the Insole sensors have three pressure sensors (right).

The barometric pressure readings are expressed in *Hectopascal (hPa)*. As depicted in Figure 3.7 (right), the Insole sensors additionally have three distributed *Force Sensitive Resistors (FSRs)*, which measure the pressure as relative values without a unit.

Both parent classes, `AbstractNilsPodSensor` and `AdvancedAbstractNilsPodSensor`, contain read-only dictionaries with specific commands. They are internally used to control the behavior of the peripheral, modify the sensor configuration, or execute flash memory operations. For example, the `AbstractNilsPodSensor` class provides necessary control commands, which are shown in Listing 3.4. In order to start the data streaming of a sensor, the hexadecimal value `0xC2` must be written to the command characteristic of the *UART* service. In a similar way, further actions such as *Start Logging* or *Sensor Reset* are also triggered. The library wraps the respective necessary steps in separate methods, such as `startStreaming` or `stopLogging`, and provides them to the user.

Another essential property of all NilsPod sensors is the `systemState`. It can have the values `Idle`, `Streaming`, `Logging`, `TxSessionData`, `TxSessionList`, `FlashErase`, or `SaveSettings` and is mainly used to define the way incoming data is handled. In general, the following steps are taken to send data from the peripherals to the data concentrator. During the initialization process of a NilsPod sensor, all services, characteristics, and descriptors are discovered and instantiated. Since the data characteristic of the *UART* service is internally accessed several times, it is directly stored in a separate property. As with the `BleSensor` class, the `BleCharacteristic` class extends the `EventEmitter` class. This enables all its instances, including the `dataUartCharacteristic`, to create and emit custom events. Another step in

```
protected ControlCommands: ReadonlyDictionary<Buffer> = {
  STOP_STREAMING: Buffer.from('C1', 'hex'),
  START_STREAMING: Buffer.from('C2', 'hex'),
  STOP_LOGGING: Buffer.from('C3', 'hex'),
  START_LOGGING: Buffer.from('C4', 'hex'),
  RESET: Buffer.from('CFFF', 'hex')
};
```

Listing 3.4: The `ControlCommands` dictionary is a collection of commands that are used to control the behavior of a NilsPod sensor.

the initialization process is the subscription of the `dataUartCharacteristic`. After that, it throws a `data` event every time it receives a message. The `AbstractNilsPodSensor` class provides an `EventListener` that handles these `data` events. Up to this point, all incoming data is handled equally. In addition, the data has still the form of a `Buffer` consisting of successive hexadecimal numbers. However, depending on the current data transfer operation, the `Buffer` positions contain completely different data and must therefore be processed differently. It is the system state, which is defined at the beginning of an operation, that determines how the data is handled.

For example, if the method `startStreaming` is called on an Insole sensor, its system state is set to `Streaming`. In this case, the Insole sensor instance uses the `InsoleStreaming-DataFrame` to transform each data string into an easy-to-handle data object (cf. Listing 3.3). However, streaming is not the only operation, where data is transferred from peripherals to the central device. Other actions, such as the transmission of the sensor's session list or entire logged sessions, use exactly the same data channels: the data characteristic or the *UART* service and `data` events. If `readSessionList` is called, the library uses another type of data frame to convert the incoming `Buffer` into an object that gives an overview of the various sessions stored in the flash memory. When one of these recorded sessions is transferred, the incoming data packets are concatenated and stored in a binary file on the data concentrator.

As shown in Figure 3.8, the specific sensor classes for the Hoop, classic NilsPod, and Insole sensor type hardly contain attributes. Most of their necessary properties and methods are provided by their superordinate classes `AbstractNilsPodSensor` and `AdvancedAbstractNil-sPodSensor`, respectively. Only the class for the Insole sensors, which have additional pressure sensors, is extended by corresponding features. The other two classes simply override the static property `nameFilterSearchString`.

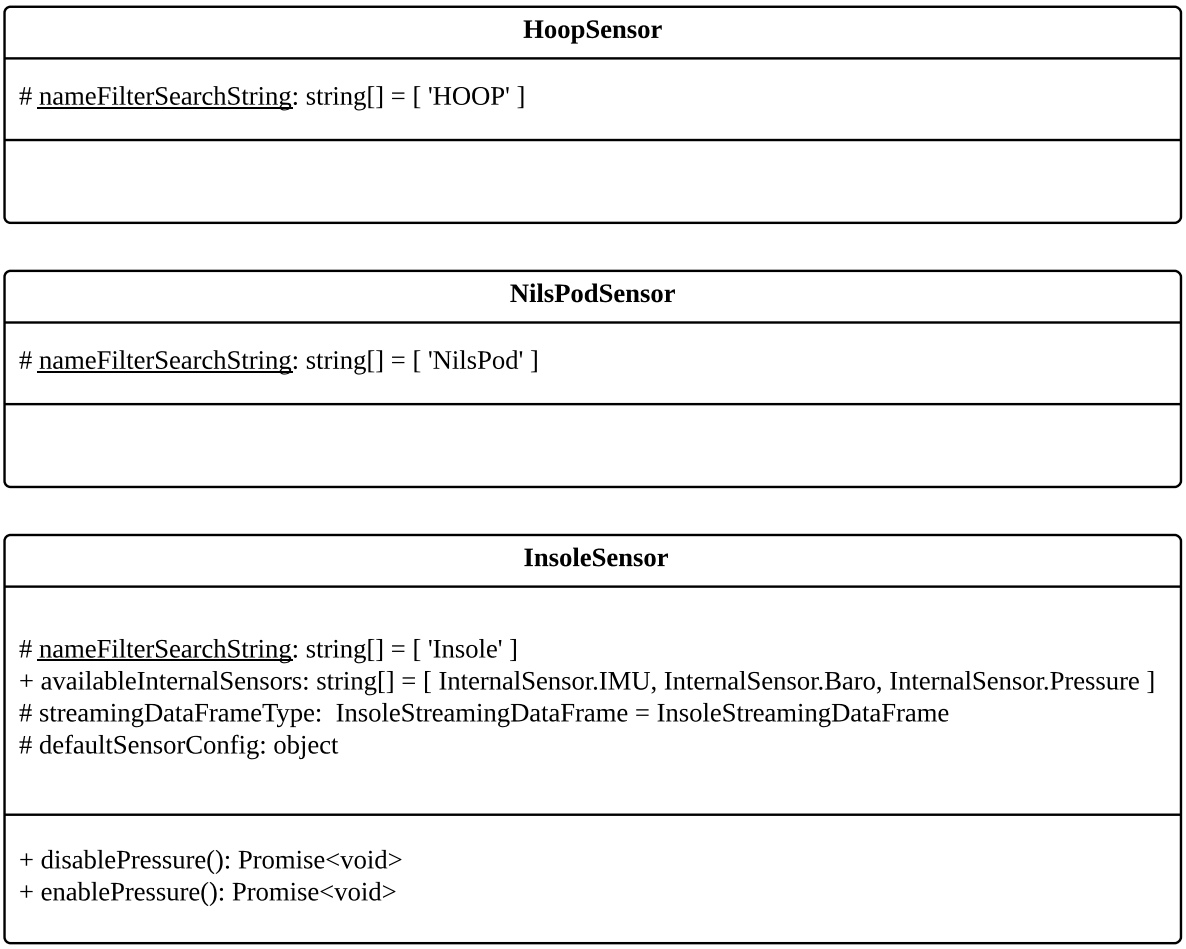

Figure 3.8: Most of the attributes of Hoop, classic NilsPod, and Insole sensors are already provided by their parent sensor classes.

Depending on their capabilities and features, the NilsPod specific classes also implement certain interfaces (cf. Table 3.1). In addition to the `BasicSensorInfo` interface, which is inherited from the `BleSensor` class, all three NilsPod sensor classes implement the `Streaming` and `Resettable` interfaces. The classic NilsPod and Insole sensor classes further integrate the `Logging` as well as the `SensorModification` interfaces. As shown in Figure 3.9, these two interfaces cover several sensor functionalities and flags. In general, these `boolean` variables can be used to check whether a sensor has a certain capability, such as scanning or logging. This is particularly helpful for JavaScript applications that do not support interfaces. In TypeScript, it is possible to apply the `is` operator to check whether an object implements a particular interface. Other possibilities to inspect the functionalities of sensor instances are the `sensorInstance.hasOwnProperty(prop)` or `sensorIn-`

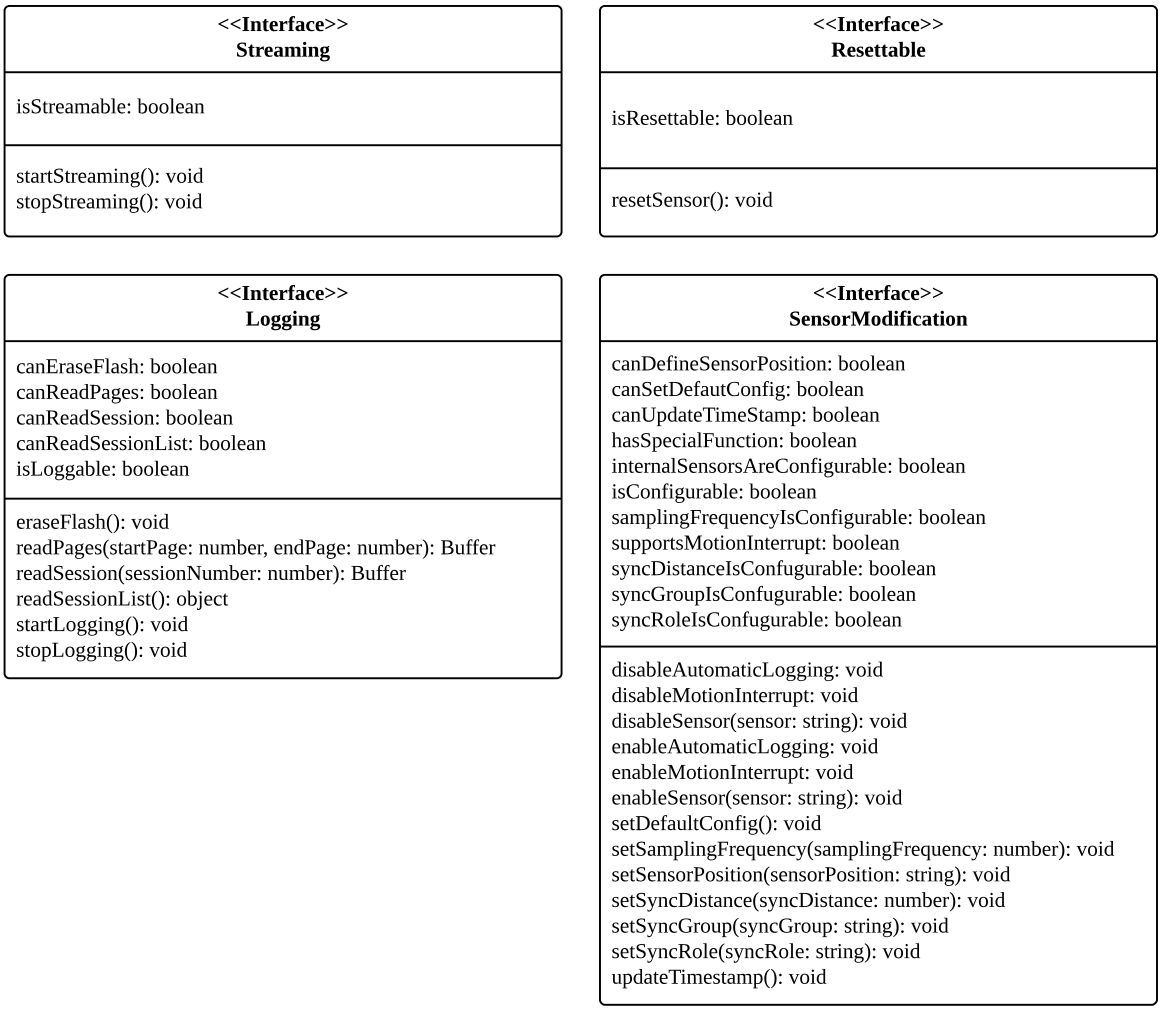

Figure 3.9: In addition to `BasicSensorInfo`, the sensor library framework provides further interfaces that can be implemented by sensor classes.

`stance.startStreaming ? trueOperation : falseOperation` syntaxes. As an alternative to examining the features of a sensor object in advance, it is also an option to directly call a method and catch and handle possible errors accordingly. These can be either an `UnsupportedFunctionalityError`, in case the boolean flags are set to `false`, or a `TypeError`, if an instance does not have this attribute. In addition, the interfaces require the implementing classes to have the methods that perform the respective operations. When developed in TypeScript, new sensor classes can, regardless of their supported communication protocol, apply the same interfaces. This facilitates the implementation of data concentrator applications based on this sensor library framework.

3.1.2 Sensor Registry

The basic idea of a sensor registry is to automatically assign a newly discovered peripheral device to the appropriate sensor class. For example, in a home monitoring system a number of different NilsPod sensor models are used. The data concentrator discovers several peripherals, each emitting its specific advertisement package, along with some metadata. Based on these first information, the sensor library now has to select the correct sensor classes for the devices. The sensor library then creates instances of the respective class as representatives of the physical peripherals. The correct assignment of the corresponding sensor classes is very important, because only then do the sensor classes cover all their features and capabilities.

The approach of the sensor library for this automatic classification in sensor classes is divided into two steps. First of all, a registry of all known and available sensor classes is necessary. For this purpose, the `BleSensor` class provides the static property `registeredSensorClasses`. It is initialized as an empty object, but can be filled with all known sensor classes at the beginning of an application. This requires to call the `BleSensor` class method `registerAll`, which is shown in Listing 3.5.

```
1  class BleSensor extends EventEmitter implements BasicSensorInfo {
2    ...
3    public static registerAll(): void {
4      this.register(AbstractNilsPodSensor);
5      this.register(HoopSensor);
6      this.register(AdvancedAbstractNilsPodSensor);
7      this.register(NilsPodSensor);
8      this.register(InsoleSensor);
9    }
10   ...
11 }
```

Listing 3.5: The static method `registerAll` enters all known sensor classes in the sensor registry.

The *registerAll* function calls another user-defined static method, `register`, for each known sensor class. The `register` method, in turn, enters the respective sensor class at the correct position in the sensor registry, the property `registeredSensorClasses`: As illustrated in Listing 3.6, parent classes, such as `BleSensor` or `AbstractNilsPodSensor`, are registered as keys, a list of their child classes as their values. This procedure enables dynamic registration of

the available sensor classes during runtime.

```
{ BleSensor:
  [
    { [Function: AbstractNilsPodSensor] }
  ],
  AbstractNilsPodSensor:
  [
    { [Function: HoopSensor] },
    { [Function: AdvancedAbstractNilsPodSensor] }
  ],
  AdvancedAbstractNilsPodSensor:
  [
    { [Function: InsoleSensor] },
    { [Function: NilsPodSensor] }
  ]
}
```

Listing 3.6: The static property `registeredSensorClasses`, defined in the `BleSensor` class, serves as the sensor registry for known sensor classes.

In a second step, a newly detected peripheral is classified in the corresponding class from the sensor registry. The core of this categorization process is the static method `findSensorClass`, which is also defined in the `BleSensor` class. It returns the appropriate sensor class for a given peripheral object. For this, it starts searching at the parent class, `BleSensor`, and then follows the class tree to more specific sensor classes. Thereby, it calls the helper method `filterScan` on each registered sensor class. This method returns `true` in case the discovered peripheral belongs to the respective sensor class. The algorithm used for this decision compares the name advertised by the peripheral with a list of predefined names. As shown in Figure 3.1, each sensor class provides its own list of allowed names in the already mentioned static property `nameFilterSearchString`. Strictly speaking, this list contains words that only need to be part of the peripheral's name. The `nameFilterSearchString` property of the `BleSensor` class consists of an empty string (`''`). This means that *BLE*-enabled devices whose name is `undefined` or `null` are not assigned to the `BleSensor` class.

For example, after all known sensor classes have been registered using the `registerAll` method, a *BLE* device named *HOOP-2EDE* is discovered. In the sensor classification procedure it

is first categorized as `BleSensor`, then `AbstractNilsPodSensor`, and finally `HoopSensor`. Consequently, it is instantiated as a `HoopSensor`.

In case the sensor library is expanded by further *BLE* sensor classes, they can either override only the `nameFilterSearchString` property with its own array of names or the `filterScan` method as a whole. In order to include them to the sensor registry, there are again two possibilities: They can be directly added to the `registerAll` method of the `BleSensor` or registered during runtime using the `register` function. The second option is explained in more detail in Subsection 3.2.7.

3.2 Application

The aim of the Node.js-based sensor library framework developed in this work is to facilitate the implementation of a data concentrator application for a home monitoring infrastructure. For this purpose, the library provides various functionalities that cover the most important use cases of such an application.

3.2.1 Integrating the Sensor Library Framework

The sensor library is developed in TypeScript, but exported both in a TypeScript and in a compiled JavaScript version. Thus, any Node.js-based application that integrates the framework can be programmed in both languages. However, since Node.js is an engine that runs JavaScript, software written in TypeScript must always be compiled first. Nevertheless, by providing interfaces, enumerations, generics, and a typing system, the use of TypeScript in Node.js-based applications is recommended for large projects.

For the installation of the sensor framework, the *Node.js Package Manager (NPM)* can be used. The library and all its dependencies, such as noble, are installed with the following *NPM* command: `npm install [path-to-local-library]/nodejs_sensorlib`. As shown in Listing 3.7, the required modules can then be imported in two different ways, depending on the JavaScript version of the application. Node.js supports most of the new features of *ECMAScript* 6, such as classes, arrow functions, or the `let` and `const` keyword [Kap]. However, as of now (March 2019), *ECMAScript* 6 modules cannot be used with Node.js. The current module system of Node.js is *CommonJS*, which utilizes the `require` keyword for imports. A common way to develop Node.js-based applications with all *ECMAScript* 6 features including the `import` keyword is using a compiler library, such as the open-source *Babel* package [BT]. Its main

functionality is to convert *ECMAScript* 6 into older JavaScript versions, which are compatible with Node.js and all browsers. In contrast, the TypeScript comes with a built-in compiler, which can convert TypeScript into a JavaScript version that is supported by Node.js. Therefore, all new JavaScript features can be used in a TypeScript program. The following application examples are programmed in JavaScript with CommonJS modules, which allows them to be directly executed.

```
1 // CommonJS Modules:
2 const { BleSensor } = require('nodejs_sensorlib');
3
4 // ES6 Modules:
5 import { BleSensor } from 'nodejs_sensorlib';
```

Listing 3.7: Depending on the JavaScript version of the application, the sensor library modules must be imported in two different ways.

3.2.2 Scanning

The sensor library framework offers the ability to scan for new sensor devices. Additionally, it is possible to check the current *RSSI* value and the manufacturer data of devices without connecting to them. In general, the library provides two basic scanning modes: event-driven and time-controlled scanning. In the first one, presented in Listing 3.8, the `EventEmitter` extension of the `BleScanner` class is used. As shown in Figure 3.3, each *BLE* sensor class has the static property `scanner`, a link to the global `BleScanner` instance of the application. The `scanner` emits three different types of custom events. If a new peripheral is detected, a `discoverSensor` event is triggered, which passes the instance of the appropriate sensor class. In case the *RSSI* value of an already discovered devices changes, an `updateRSSI` event is thrown that transmits the sensor ID along with the new *RSSI* value. Another event, `updateManufacturerData`, is emitted, if the manufacturer data of an already detected devices is updated. The manufacturer data is part of the advertisement packet, which is regularly broadcast by a *BLE* sensor. In this application example, three `EventListeners` are defined that respond to these events. Their callback functions take the passed arguments and handle them accordingly. In this case, they only log the content of the events to the console, but in other applications, for example, these information could be send to a front-end. After the definition of the `EventListeners`, the static `registerAll` method is called. As a result, all known and available sensor classes are entered into the sensor registry. Finally, the actual scanning process is started by invoking `findAll`,

another static method of the `BleSensor` class. Consequently, the `scanner` searches for all *BLE* sensors within the range of the central device without time limit. In the case of an error when starting the the scanner, it is caught and logged to the console. Depending on the cause, the sensor library framework defines a number of custom errors. For example, if the `scanner` is busy with another scanning operation, an `InvalidOperationError` is thrown. With the event-driven scanning mode, a list of available *BLE* devices can be generated that updates when a new peripheral is detected or one of the *RSSI* values or manufacturer data changes.

```
1  const { BleSensor } = require('nodejs_sensorlib');
2
3  BleSensor.scanner.on('discoverSensor', sensor => {
4    console.log('New Sensor discovered:
5      Name: ${sensor.name}
6      Sensor Type: ${sensor.sensorType}
7      Address: ${sensor.address}'
8    );
9  });
10
11 BleSensor.scanner.on('updateRSSI', (id, rssi) => {
12   console.log('New RSSI value of ${id}: ${rssi}');
13 });
14
15 BleSensor.scanner.on('updateManufacturerData', (id, manufacturerData)
       => {
16   console.log(id, manufacturerData);
17 });
18
19 BleSensor.registerAll();
20
21 BleSensor.findAll().catch(error => console.error(error));
```

Listing 3.8: In the event-driven scanning process, the emitted events of the `scanner` property of the `BleSensor` class are used to handle new sensors, updated *RSSI* values, or manufacturer data changes.

In contrast to the event-driven scanning, time-controlled scanning is limited to a certain period of time. In an application, this is realized by passing a number as an argument to the

`findAll` method specifying the scan time in seconds. In this case, the method returns a promise containing an array of appropriate sensor instances. The method `find`, another static method of the `BleSensor` class, works similarly, but unlike `findAll`, it only returns a single instance. For example, `BleSensor.find(5)` gives back an instance of the first *BLE* peripheral that can be discovered within five seconds. If no device is found within the given time period, a corresponding error is thrown.

For both scanning modes, event-driven as well as time-controlled, it is possible to filter the scan. One way to do this is to first scan for all types of sensors and afterwards treat the devices differently depending on properties such as sensor type, name, *RSSI*, or isConnectable. The other possibility provided by the framework is to directly constrain the scan to certain sensor types. Instead of calling `findAll` or `find` on the `BleSensor` class, they can be invoked on any derived sensor class, such as `AbstractNilsPodSensor` or `HoopSensor`. As a result, the user is only provided with peripherals of the respective sensor type including its inherited types.

Listing 3.9 presents a time-controlled scanning process, which is limited to NilsPod sensors only. The application searches for NilsPod sensors for three seconds and stores the result, a list of detected instances of the appropriate sensor class, in a variable. As the `await` keyword is used to wait for the promise to be resolved, the operation has to be within an `async` function closure.

```
1  const { AbstractNilsPodSensor } = require('nodejs_sensorlib');
2
3  AbstractNilsPodSensor.registerAll();
4
5  const logListOfNilsPodSensors = async () => {
6    const listOfNilsPodSensors = await AbstractNilsPodSensor.findAll(3);
7
8    listOfNilsPodSensors.forEach(sensor => {
9      console.log('
10       Name: ${sensor.name}
11       Sensor Type: ${sensor.sensorType}
12     ');
13   });
14 };
15 logListOfNilsPodSensors().catch(error => console.error(error));
```

Listing 3.9: The time-controlled, filtered scanning process can be used to search for a particular type of sensors for a certain time period.

Another use case could require the possibility of directly finding an already known sensor and then connecting to it. Listing 3.10 shows the option to discover a *BLE* devices using its address. In contrast to the other examples, the sensor is not instantiated by internal procedures of the library framework, but in the application itself. For this purpose, the `constructor` function of the `BleSensor` class is used, which takes the address as a string to create an instance. Subsequently, the method `discoverSensor` has to be called, in order to have the internal `scanner` search for the corresponding device. The timeout for this scan is defined in the private property `scanForKnownSensorTimeout` and is three seconds, by default. As a next step, the sensor instance can be connected.

```
1  const { BleSensor } = require('nodejs_sensorlib');
2
3  const exampleAddress = 'a1:b2:c3:d4:e5:f6';
4
5  const scanForKnownSensor = async () => {
6    const sensor = new BleSensor(exampleAddress);
7    // sensor is instantiated, but not yet discovered
8
9    await sensor.discoverDevice(); // sensor is discovered
10 };
11
12 scanForKnownSensor().catch(error => console.error(error));
```

Listing 3.10: An already known sensor can be directly found using the constructor function and its *BLE* address.

3.2.3 Connecting and Getting Information

In a typical home monitoring system, data is exchanged between the peripherals and the central unit. This requires these two devices to be connected successfully. As shown in Listing 3.11, the library provides a `connect` method, which can be called on discovered sensors. It returns an empty promise that is resolved as soon as the device is connected. Furthermore, the `BleSensor` class has the static property `autoConnect`, which is set to `false`, by default. By changing its value to `true`, a discovered device is automatically connected when a method is invoked on it.

As soon as a *BLE* device is connected, several methods can be called on it, for example `getBatteryLevel` or `getSensorConfig`. As both operations require to read certain char-

acteristics from the sensor, they do not return their values immediately, but pack them into promises. Thus, the following actions, in this case the `console.log`, are not called until the promises are resolved and can directly access their contents. If the sensor does not provide battery measurement or has no sensor configuration property, an appropriate custom error, in this case an `UnsupportedFunctionalityError`, is thrown. They can be intercepted and handled accordingly. Alternatively, the user could use the boolean flags, such as `hasBatteryMeasurement`, to check beforehand whether a sensor instance has a certain functionality.

```
1  const { BleSensor } = require('nodejs_sensorlib');
2  BleSensor.registerAll();
3  BleSensor.autoConnect = true; // 'false' by default
4
5  const ConnectToSensor = async () => {
6    const sensor = await BleSensor.find(3); // scans for 3 seconds
7
8    await sensor.connect();
9    // actually redundant, because BleSensor.autoConnect = true
10
11   const batteryLevel = await sensor.getBatteryLevel();
12   console.log('The battery level is ${batteryLevel} %.');
13
14   const sensorConfig = await sensor.getSensorConfig();
15   console.log('The sensor configuration is ${sensorConfig}.');
16
17   await sensor.disconnect();
18 };
19
20 ConnectToSensor().catch(error => console.error(error));
```

Listing 3.11: After the sensor is connected important information (e.g. the battery level) can be obtained using the approprate methods.

The sensor library framework allow multiple peripherals to be simultaneously connected to a central device. The Bluetooth *SIG* does not limit the number of *BLE* connections, in most cases, however, it is restricted by the available memory resources of the master device.

3.2.4 Streaming

One functionality supported by all NilsPod sensors is *streaming*. A device records measured values through its internal sensor packages, such as *IMU* or barometer, and sends them directly to the central unit. Listing 3.12 demonstrates that this data transfer can be easily implemented in an application with the help of the sensor library.

```
1  const { AbstractNilsPodSensor } = require('nodejs_sensorlib');
2  AbstractNilsPodSensor.registerAll();
3
4  const streaming = async () => {
5    const sensor = await AbstractNilsPodSensor.find(3);
6
7    sensor.on('warning', warning => {
8      // In case a data frame gets lost, a warning is emitted and can be
           handled here.
9      console.warn(warning);
10   });
11
12   sensor.on('data', dataFrame => {
13     console.log(dataFrame); // the data frames can be handled here
14   });
15
16   await sensor.connect();
17   await sensor.startStreaming();
18
19   setTimeout(async () => {
20     await sensor.stopStreaming().catch(error => console.error(error));
21     await sensor.disconnect().catch(error => console.error(error));
22     // errors in a 'setTimeout' construct must be directly caught
23   }, 5000); // stops streaming after 5 seconds
24 };
25
26 streaming().catch(error => console.error(error));
```

Listing 3.12: When a device is streaming, its sensor instance emits a `data` event for each incoming data packet.

The streaming functionality of the sensor framework leverages the `EventEmitter` extension of the `BleSensor` class. As soon as the `startStreaming` method is called, the sensor emits `data` events containing the respective data frames. For example, an Insole device sends its data packets as `InsoleStreamingDataFrames` (cf. Listing 3.3). In the application, an `EventListener` can be defined, which handles these `data` events. In this case, it logs the data frames to the console and causes the sensor to stop streaming after 5 seconds.

It can happen that a NilsPod sensor loses a data frame during streaming. In this case, a `DataFrameLostWarning` is thrown. However, in event-driven operations, such as streaming, the `catch` keyword cannot be used to handle errors or warnings. The `.catch` or `try...catch` syntaxes only work in a synchronous program flow. For example, in Listing 3.12 in line 16, the program stops and awaits the promise to be resolved, which happens when the sensor is successfully connected. If the sensor is not connectable, the promise is rejected and a corresponding error is triggered, which is intercepted in line 26 by the `catch` keyword. A `DataFrameLostWarning` would not be thrown immediately, but at any point during streaming. At this time, the engine has already left the `streaming().catch(...)` construct in line 26, which means errors are not caught anymore. For these reasons, sensor instances emit `warning` events, if they lose a data frame during streaming. As with the `data` events, the applications defines another `EventListener` to handle this type of events.

The example application of Listing 3.12 can be extended by a `CsvWriter`, a tool provided to the user in form of a separate class. It can be used to directly store received data during streaming in a *Comma-separated Values (CSV)* file. As shown in Listing A.1, a `CsvWriter` is instantiated with the storage path of the file as argument. For each measured value of the incoming data frames, the `CsvWriter` creates a separate field. For example, if a Hoop sensor is streaming, the header of the file contains the following comma-separated fields: `localCounter`, `gyroX`, `gyroY`, `gyroZ`, `accelX`, `accelY`, and `accelZ`. The readings of each received data frame are saved in the *CSV* file as a new line. The `CsvWriter` instance emits a `FileSystemError`, if the given storage path is invalid or the corresponding file is read-only, which means it is locked by another user or process. These errors can be intercepted and handled by an `EventListener`. Once the streaming process is complete, it is important to call `end` on the `CsvWriter` instance, in order to unlock the *CSV* file.

3.2.5 Logging

The sensor library framework provides methods to start or stop the logging mode of a sensor. In contrast to streaming, a peripheral stores the measured values in its internal flash memory

instead of transferring them directly to a central device. This functionality is supported by the advanced NilsPod devices, the classic NilsPod and Insole sensors. In the example application, shown in Listing 3.13, a sensor is logging for five seconds. If the `stopLogging` method is directly followed by another command, such as `disconnect`, the `await` keyword must be prefixed. The reason for this is that the sensor needs a short moment to exit logging mode.

```
1  const { AdvancedAbstractNilsPodSensor } = require('nodejs_sensorlib');
2  AdvancedAbstractNilsPodSensor.registerAll();
3
4  const logging = async () => {
5    const sensor = await AdvancedAbstractNilsPodSensor.find(3);
6
7    await sensor.connect();
8
9    await sensor.startLogging();
10
11   setTimeout(async () => {
12     await sensor.stopLogging().catch(error => console.error(error));
13     await sensor.disconnect().catch(error => console.error(error));
14     // errors in a 'setTimeout' construct must be directly caught
15   }, 5000); // stops logging after 5 seconds
16 };
17
18 logging().catch(error => console.error(error));
```

Listing 3.13: The logging mode can be easily turned on and off using appropriate methods of the `AdvancedAbstractNilsPodSensor` class.

Each time a sensor starts logging, the data is stored in a separate session in the flash memory. A classic NilsPod or Insole sensor can record a maximum of 25 sessions, regardless of their size. Their flash memories have a capacity of 2 or 4 Gbit and consist of multiple pages of 2048 bytes each. As shown in Listing 3.14, the sensor library framework provides appropriate methods to handle the stored sessions of the advanced NilsPod sensors.

```
const { AdvancedAbstractNilsPodSensor } = require('nodejs_sensorlib');
AdvancedAbstractNilsPodSensor.registerAll();

const handleFlashSessions = async () => {
  const sensor = await AdvancedAbstractNilsPodSensor.find(3);

  sensor.on('error', error => {
    // in case the given path is invalid or the file is read-only
    console.error(error);
  });

  await sensor.connect();

  // await sensor.eraseFlash(); // partial erase
  // await sensor.eraseFlash(true); // full erase
  // Use these methods to erase all sessions from flash memory.

  const sessionList = await sensor.readSessionList();
  console.log(sessionList);

  const sessionOne = await sensor.readSession(1);

  await sensor.saveBinaryFile(sessionOne, 'example');

  await sensor.disconnect();
};

handleFlashSessions().catch(error => console.error(error));
```

Listing 3.14: The sensor library framework provides a number of methods for transferring, saving, and deleting the logged sessions from the flash memory.

First of all, the `readSessionList` method can be called on a sensor instance to get an overview of the recorded sessions. Therefore, the peripheral sends an encoded data string containing information about each session. As with the different streaming data frames, a separate class, the `FlashSessionDataFrame`, is used to convert this byte string into an easy-to-handle

object. In the end, the `readSessionList` method returns a promise with a list containing an instance of this class for each session. An example of such an object is shown in Listing 3.15.

```
FlashSessionDataFrame {
  index: 1,
  startPage: 0,
  lastPage: 3,
  size: 8192,
  samplingFrequency: 102.4,
  sampleSize: 16,
  terminationFlag: 'SESSION_TERMINATION_BLE',
  timestamp: 2019-03-16T11:20:56.000Z
}
```

Listing 3.15: The `FlashSessionDataFrame` provides information about a logged session from the flash memory.

The example shows the first recorded session of an classic NilsPod and Insole sensor. It has the `index` 1 and covers pages 0 to 3 inclusive and thus has a total size of 8192 bytes. The measured values were recorded with a sampling frequency of 102.4 Hz, with each data sample having a size of 16 bytes. The `terminationFlag` property specifies how the session was terminated. In this case, it was ended with the `stopLogging` command. The `timestamp` key gives the exact starting time of the session.

As a next step, the `readSession` method can be used to transmit a specific session from the peripheral to the central device. It returns a promise containing the measured values from the session with the passed index in the form of a `Buffer`. Afterwards, `saveBinaryFile`, another method defined in the `AdvancedAbstractNilsPodSensor` class, saves the `Buffer` as a binary file in the given storage path. As with the save operation of the `CsvWriter`, an `error` event is emitted, if the path is invalid or the file is read-only. Another method provided by the sensor library framework is `eraseFlash`, which clears either only the pages containing data (partial erase) or the entire flash of a device (full erase).

3.2.6 Modifying the Sensor Configuration

Another feature supported by the advanced NilsPod sensors is the modification of the device configuration. Their corresponding sensor class, the `AdvancedAbstractNilsPodSensor`,

provides public methods that take care of the internal operations required for these actions. Some of these methods are used in the example application in Listing 3.16. For example, `setSamplingFrequency` takes a certain sampling frequency as a number and writes the value of one of the characteristics of the configuration service accordingly. The method only allows specific values as arguments. A user can get an overview of the possible sampling frequencies by calling `Object.values(sensor.SamplingFrequency)`. The `enableSensor` and `disableSensor` methods are used to turn one of internal sensors on and off, respectively. As argument, one of the following values has to be passed as strings: 'IMU', 'Barometer', or 'Pressure'. Alternatively, methods such as `disableIMU` or `enableBarometer` can be used to modify the sensor composition. The `updateTimestamp` function updates the internal clock of the peripheral.

A method, which is supported by all NilsPod sensor instances, is `getSensorConfig`, thus it is already defined in the `AbstractNilsPodSensor` class. It returns an object with information about the current sensor configuration. By calling the method `setDefaultConfig`, classic NilsPod and Insole sensors can be reset to default settings (cf. Table 3.2).

```
1  const { AdvancedAbstractNilsPodSensor } = require('nodejs_sensorlib');
2  AdvancedAbstractNilsPodSensor.registerAll();
3
4  const sensorModification = async () => {
5    const sensor = await AdvancedAbstractNilsPodSensor.find(3);
6
7    console.log('Sampling Frequency:', Object.values(sensor.
       SamplingFrequency));
8    // logs the possible values for the following sensor settings
9
10   await sensor.connect();
11
12   await sensor.setSamplingFrequency(204.8);
13   // sets the sampling frequency to 204.8 Hz
14
15   await sensor.disableSensor('IMU'); // alternative: disableIMU()
16   // disables the internal sensor 'IMU'
17
18
```

```
19    await sensor.enableSensor('Barometer');
20    // alternative: enableBarometer()
21    // enables the internal sensor 'Barometer'
22
23    await sensor.updateTimestamp();
24    // updates the timestamp of the sensor
25
26    const sensorConfig = await sensor.getSensorConfig();
27    console.log(sensorConfig);
28
29    // await sensor.setDefaultConfig();
30    // use this method to reset a sensor to its default configuration
31    // disconnects automatically
32
33    await sensor.disconnect();
34  };
35
36  sensorModification().catch(error => console.error(error));
```

Listing 3.16: The sensor library facilitates the modification of the sensor configuration by providing methods such as `setSamplingFrequency` or `enableSensor`.

3.2.7 Create and Register new Sensor Classes

The sensor library is designed according to the *OOP* paradigm. As a result, it can easily be extended by further sensor classes, either directly in the framework or later in the application importing the library. New *BLE* sensor classes can inherit from the `BleSensor` class, which already provides their basic properties and methods. If TypeScript is used, additional classes can further implement one or multiple of the predefined interfaces. As a result, it is ensured that instances of sensor devices with identical functionalities provide the same properties and methods. Moreover, new sensor classes should be entered into the sensor registry. This can be done either directly in the library, by appending the class in the `registerAll` method or by using the `register` method in the application itself.

The possibility to add a new sensor class only in the actual application is presented in Listing 3.17. In this example, a new *BLE* sensor class is created extending the `BleSensor` class. It overrides the `nameFilterSearchString` by its own array of names. Furthermore, it is

entered into the sensor registry by applying the static method `register`, which is provided by the `BleSensor` class. Consequently, a peripheral that has either *ExampleSubstring1* or *ExampleSubstring2* in its name, will be instantiated from this class.

```
1  const { BleSensor } = require('nodejs_sensorlib');
2
3  class NewExampleSensorClass extends BleSensor {
4    constructor(args) {
5      super(args);
6      this.hasBatteryMeasurement = true;
7    }
8
9    exampleMethod() {
10     // some example operation
11   }
12 }
13
14 NewExampleSensorClass.nameFilterSearchString = ['ExampleSubstring1', '
     ExampleSubstring2'];
15
16 NewExampleSensorClass.register();
```

Listing 3.17: The `NewExampleSensorClass` class inherits from the `BleSensor` class, overrides the `nameFilterSearchString` and is entered into the sensor registry. Since JavaScript does not support `static` attributes within a class closure, they have to be defined outside a class.

Chapter 4

GUI Development

A possible use case of the sensor framework developed within this work is the implementation of a sensor control application. In order to test and validate the sensor library, such an application was implemented as part of this work. It consists of two building blocks: On the one hand, a Node.js-based backend embedding the sensor library, on the other hand, a *GUI* as a frontend. The main purpose of such an application is to allow a user to control the communication with *BLE* sensors. Its *GUI* provides operating elements to scan for available *BLE* devices, modify their configuration, let them stream or log sensor data, plot the measured values, and display the session list from the flash memory. For this, the sensor library framework is used by the Node.js-based backend to handle all communication tasks with the *BLE* devices. Thus, it simplifies the development of such *BLE* applications considerably.

A sensor control application can be used by researchers and physicians to rapidly build up test and prototype setups for motion analysis systems. Based on this, a more complex data concentrator application for home monitoring systems can be developed.

4.1 Overview of the Technology Stack

The sensor control applications is implemented in *Electron*, an open source JavaScript library developed and maintained by GitHub [Git]. It is a framework for building cross-platform desktop applications combining Node.js (backend) and Google's Chromium (frontend) into a single runtime. This allows the realization of *GUI* applications for Windows, Linux, and macOS using HTML, CSS, and JavaScript, technologies originally intended for web development. Another advantage of Electron is that communication between the backend and frontend is very simple. It provides a module called *Inter-Process Communication (IPC)*, which allows sending and receiving

serialized *JavaScript Object Notation (JSON)* messages between the *main* and *renderer* processes. The *main* process is the entry point of every Electron application and can be described as the backend. With the fully integrated Node.js *API*, it supports any Node.js-based library. Therefore, it is the *main* process that imports the sensor library framework developed within this work to communicate with potential sensor devices. Any desktop application developed with Electron can have one or more windows, each running in its own *renderer* process. They are created and managed by the *main* process and can be interpreted as the frontend of the application. Another feature of the Electron framework are *global variables*. Since they can be accessed from both, the *main* and the *renderer* processes, they provide an efficient way to exchange information without serialization between backend and frontend.

In the sensor control application developed within this work, both ways of communication, *IPC* and global variables, are applied. The *IPC* is used to build an event-driven program flow by passing user inputs from the frontend, the *GUI*, to the backend. Based on these inputs, the *main* process performs different operations and responds to the *renderer* process as needed. For example, a user clicks a *Start Scanning* button in an application window. Consequently, the corresponding *renderer* process sends a *startScanning* command via the *IPC* module to the *main* process. Leveraging the sensor library, the *main* process, in turn, calls the `BleSensor.findAll()` method. The *IPC* can also be used to transmit simple objects containing only primitives data types, such as `number` or `boolean`. For example, this applies to the data packets storing the sensor readings. However, the *IPC* module is not able to exchange more complex data structures, such as sensor instances, between the backend and frontend. In this case, the global variables are utilized as they can store objects along with their methods.

The *renderer* processes of the sensor control application use *React* to build their windows [Fac]. Developed and maintained by Facebook, React is a frontend framework that enables the creation of scalable user interfaces. A React page is divided into multiple *components*, each representing a specific part of the *GUI*. The different components can change their displayed contents, without reloading the page or re-render the entire interface. An advantage of this structure is that these components can be reused several times on a page or even in multiple applications.

4.2 Basic Functionalities

The following sections give a brief overview of the most important functionalities of the sensor control application developed within this work. Figure 4.1 shows the basic component of the *GUI*, the *sensor list*. When the scanning process is started, it presents the *sensor type*, *name*,

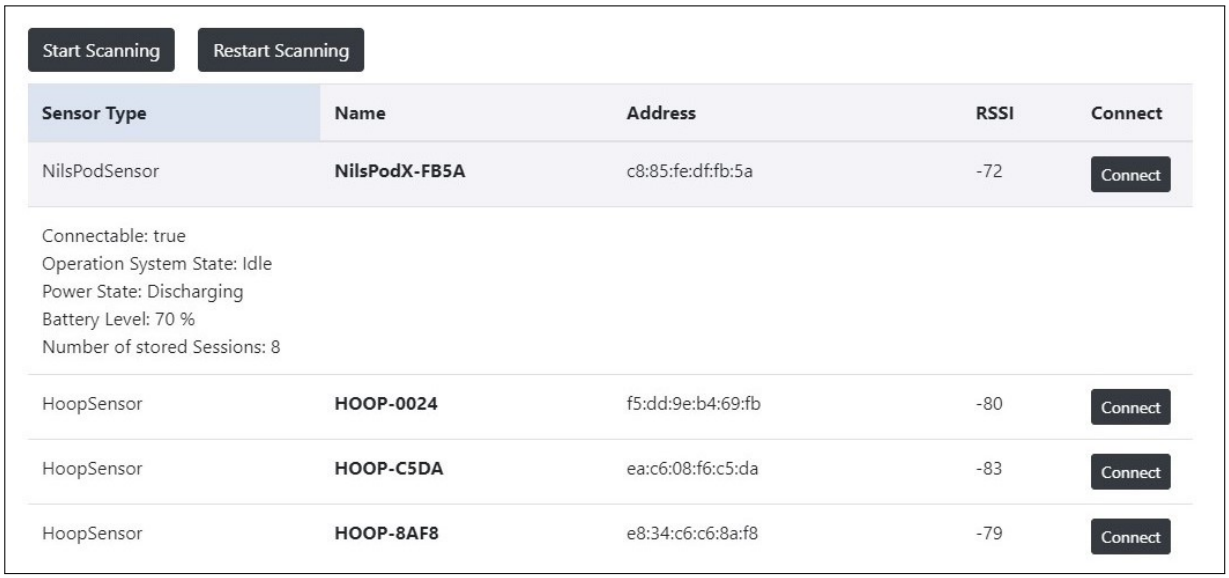

Figure 4.1: The base component of the *GUI* is the *sensor list*. It shows basic information of all available *BLE* devices.

address, and the *RSSI* of all available *BLE* devices. For this functionality, the backend uses the `findAll()` method and the `scanner` property of the `BleSensor` class provided by the sensor library framework. New sensor instances are stored in the global variable `sensors`. Once their *RSSI* value changes, the corresponding property is overridden. In the frontend, the *renderer* process accesses the global `sensors` variable and fills the *sensor list* accordingly. The resulting table can be sorted by any of the displayed characteristics. Selecting one of the sensors shows additional information by the device, such as the operation system state or the battery level advertised. The *renderer* process updates the list frequently, whereby the corresponding interval size can be varied. It is advised to set it to at least 1 s, in order not to overload the *renderer* process.

If the user clicks the *Connect* button of a sensor, a connection to the corresponding device is initiated. Once the connection has been successfully established, another component, the *sensor panel*, is displayed next to the sensor list. As shown in Figure 4.2, the content of this component depends on the capabilities of the sensor. For example, if the connected device is a classic NilsPod or Insole sensor, it contains a form to modify the sensor configuration. In addition, buttons to control the sensor, such as *Start Logging* or *Erase Flash*, are displayed. In contrast, the panel for a Hoop sensor only consists of the *Disconnect*, *Start Streaming*, *Stop Streaming*, and *Reset Sensor* buttons. For this differentiation, the application uses the boolean flags, such as `isLoggable` or `isConfigurable`, provided by each sensor instance: The *Sensor Configuration* form is

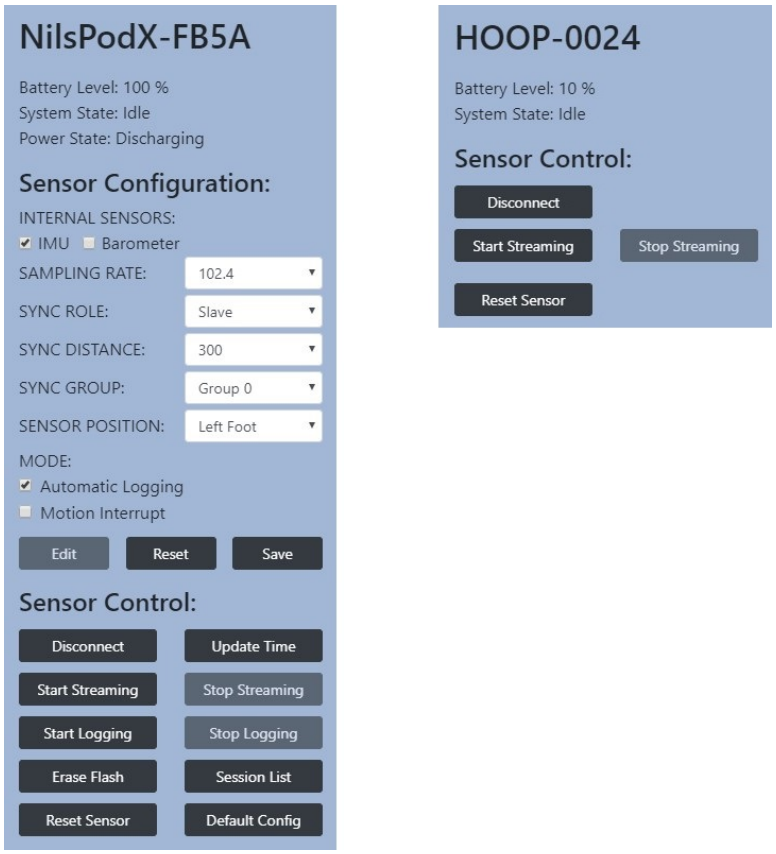

Figure 4.2: Depending on the capabilities and features of the connected sensor device, different fields are displayed in the *sensor panel* component.

only displayed, if the `isConfigurable` property of the sensor is set to `true`. Similarly, the checkboxes for toggling the internal sensors depend on the available sensor composition of the device. For Insole devices, another selection field for the *Pressure* sensor is shown.

The instances of the connected sensors are stored in separate global variables. While their *sensor panel* components are mounting, the `getBatteryLevel` and `getSensorConfig` methods are called on the instances. Since global variables support complex data structures, including functions, these methods can be directly called in the *renderer* process. The same applies in the event that one of the control elements, such as the *Disconnect* or *StartLogging* buttons are clicked: The corresponding methods are directly invoked in the frontend.

In case the *Start Streaming* button is clicked, the *main* process calls the `startStreaming` method on the respective sensor instance. As a result, the device records measured values and transmits them to the computer running the application in `data` events. The *main* process receives these events and creates new windows to display the readings as graphs. In the current state,

Figure 4.3: During a streaming process, the sensor control application can continuously display the measured values of the sensors. In this case, the gyroscope readings in $°/s$ of an Insole sensor during walking are shown. More precisely, these are three steps forward.

the application is only able to display the values of the *IMU*, but it can be extended to show the barometer or pressure sensor readings as well. If the *IMU* is enabled, two new windows open as soon as streaming starts: one for the accelerometer and one for the gyroscope. Their *renderer* processes first calibrate the incoming raw data and then continuously plot the data in line charts. For this, the open-source plotting library *Plotly.js* is used [Plo].

Figure 4.3 shows a snapshot of the graph of the gyroscope readings. The angular velocity in X-, Y-, and Z-direction is given in $°/s$. The figure illustrates the measurements of a right Insole sensor worn while walking. The sampling frequency of the *IMU* is set to 102.4 Hz, but for performance reasons only every second data sample is shown in the graph.

The corresponding data of the accelerometer are shown in Figure 4.4. The graph shows the acceleration of the Insole sensor in all three coordinate axes. The readings are given as a multitude of the gravitational acceleration constant of $g = 9.81\ m/s^2$. Based on these measured values, motion analysis systems, such as gait evaluations, can be developed.

For the classic NilsPod and Insole sensors, a list of the recorded sessions stored in their flash memory can be displayed. By clicking the corresponding button of the sensor panel the *main* process creates a separate window for this overview. As shown in Figure 4.5, the properties from

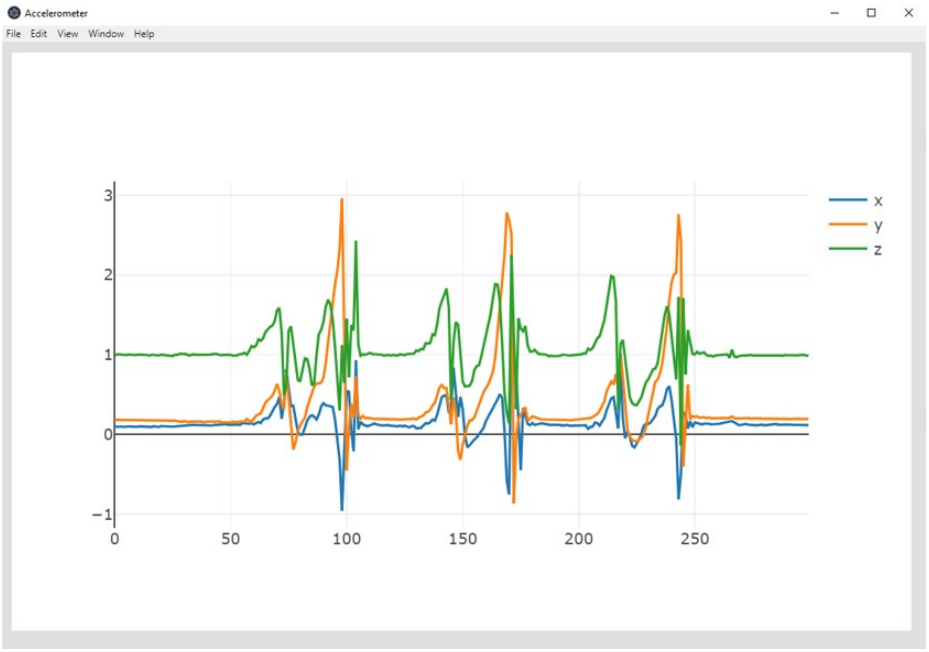

Figure 4.4: The sensor control application opens a new window for plotting the different sensors. Here, the accelerometer data in *g* of an Insole sensor is displayed during walking.

the different `FlashSessionDataFrame` instances (cf. Listing 3.15) are rendered in form of a table. Clicking the *Save* button of one of the sessions would invoke the corresponding data transmission and saving process. However, this functionality is not yet supported by the sensor control application. In addition, the application could be extended by further features, such as the simultaneous connection to multiple sensor nodes, or the creation of a *CSV* file during the streaming process.

Session List

Index	Start Page	Last Page	Size	Sample Size	Sampling Frequency	Termination Flag	Timestamp	Save
1	0	5	12288	16	102.4	SESSION_TERMINATION_BLE	30.3.2019, 18:46:47	Save
2	6	11	12288	18	102.4	SESSION_TERMINATION_BLE	30.3.2019, 18:47:12	Save
3	12	22	22528	18	256	SESSION_TERMINATION_BLE	30.3.2019, 18:48:06	Save
4	23	46	49152	12	256	SESSION_TERMINATION_DOCK	30.3.2019, 18:49:00	Save

Figure 4.5: For the classic NilsPod and Insole sensors a *session list* component can be displayed. It shows important information about the different recorded sessions stored on the flash memory.

Chapter 5

Conclusion and Outlook

The combination of wearable technologies, edge computing, and smart algorithms is expected to play an important role in the future of healthcare: Together, they have the potential to form what is called a *home health monitoring system*. Wearable devices leverage advanced sensor technologies to seamlessly track a person's motion and biomedical parameters. An edge computing device concentrates the measured values from multiple sensor nodes and stores the data in a network of interconnected remote servers, a *cloud*. Computer algorithms use the advances of artificial intelligence and machine learning to generate knowledge from the data. These insights support physicians in important medical decision-making processes. Such an ecosystem can play an essential part in the transition from a hospital-centered to a more patient-centered healthcare system.

Within the scope of this work, a Node.js sensor library framework was developed. It can be utilized in the core of the technical infrastructure described above: It facilitates the development of a data concentrator application, which is based on the cross-platform Node.js runtime. For this, the framework takes care of the communication and data exchange between the edge computing device and the wearable sensors. Since the library is strongly aligned to the programming paradigm of *OOP*, its basic concept is that each sensor node is represented by a separate `object`. Sensors with similar specifications and capabilities are instantiated from the same `class`, a collection of properties and methods. Within this work, sensor classes for Portabiles' *BLE*-enabled motion tracking devices, called NilsPod sensors, are implemented. The sensor classes cover their most important capabilities, which are needed to operate the NilsPod sensors in a home monitoring infrastructure. Using the library framework, the data concentrator application can connect to the NilsPod sensors, modify their sensor configuration, let them stream data packets, or temporarily store the readings in their internal flash memory.

The sensor library framework developed within this work is designed in such a way that it can be easily extended with further sensor classes. An important building block in this modular structure is the `BleSensor` class. It is a generic class that can be used as a parent class for any *BLE*-enabled sensor type. In addition, the library provides different interfaces, each representing a specific sensor functionality. They contain certain properties and methods that are required for this feature. Sensor classes that support one of these capabilities can implement the corresponding interface, thus forcing their instances to have the attributes defined therein.

In the moment, *BLE* is the only wireless communication standard supported by the sensor library framework. The reason for this is that *BLE* offers the best compromise between range, data throughput rate, battery live, and availability. As a result, it best meets the requirements of most wearable computing applications in the healthcare sector. However, the sensor library framework can still be extended by other communication protocols, such as Wi-Fi or ZigBee, by using the introduced class structures.

The sensor library is build on the server programming language *Node.js*. In contrast to other frameworks, such as the Android sensor library, it thus allows for cross-platform applications. Leveraging the *API* of the *noble* package, Node.js provides easy access to the Bluetooth protocol stack of a data concentrator device.

The programming language *TypeScript* is used to develop the sensor library package. As a superset of JavaScript, it is fully compatible with its libraries and *APIs*, such as Node.js. But in contrast to JavaScript, it provides a strong typing system, support for interfaces, access modifiers for attributes, generics, and enumerations. Thus, it simplifies the development of complex, modularly built JavaScript applications, which follow the *OOP* paradigm. The sensor library package is exported in both, a Typescript and a compiled JavaScript version. However, it is recommended that large-scale data concentrator applications integrating the sensor library, are also developed in TypeScript.

In order to test the sensor library framework, an example application for *BLE* sensor control was developed within this work. An appropriate *GUI* allows a user to scan for *BLE* devices, establish a connection, modify the sensor configuration and let the sensor stream data. The incoming measured values can be visualized in line charts. The entire communication with the *BLE* devices is handled via the sensor package. The implementation of this example has demonstrated that the development of a Node.js-based application that communicates to wearable sensor nodes can be considerably simplified by using the sensor library framework.

Appendix A

Additional Figures

Figure A.1: The *AdvancedAbstractNilsPodSensor* class provides shared properties for NilsPod sensors that have an internal flash memory and are able to modify their configuration.

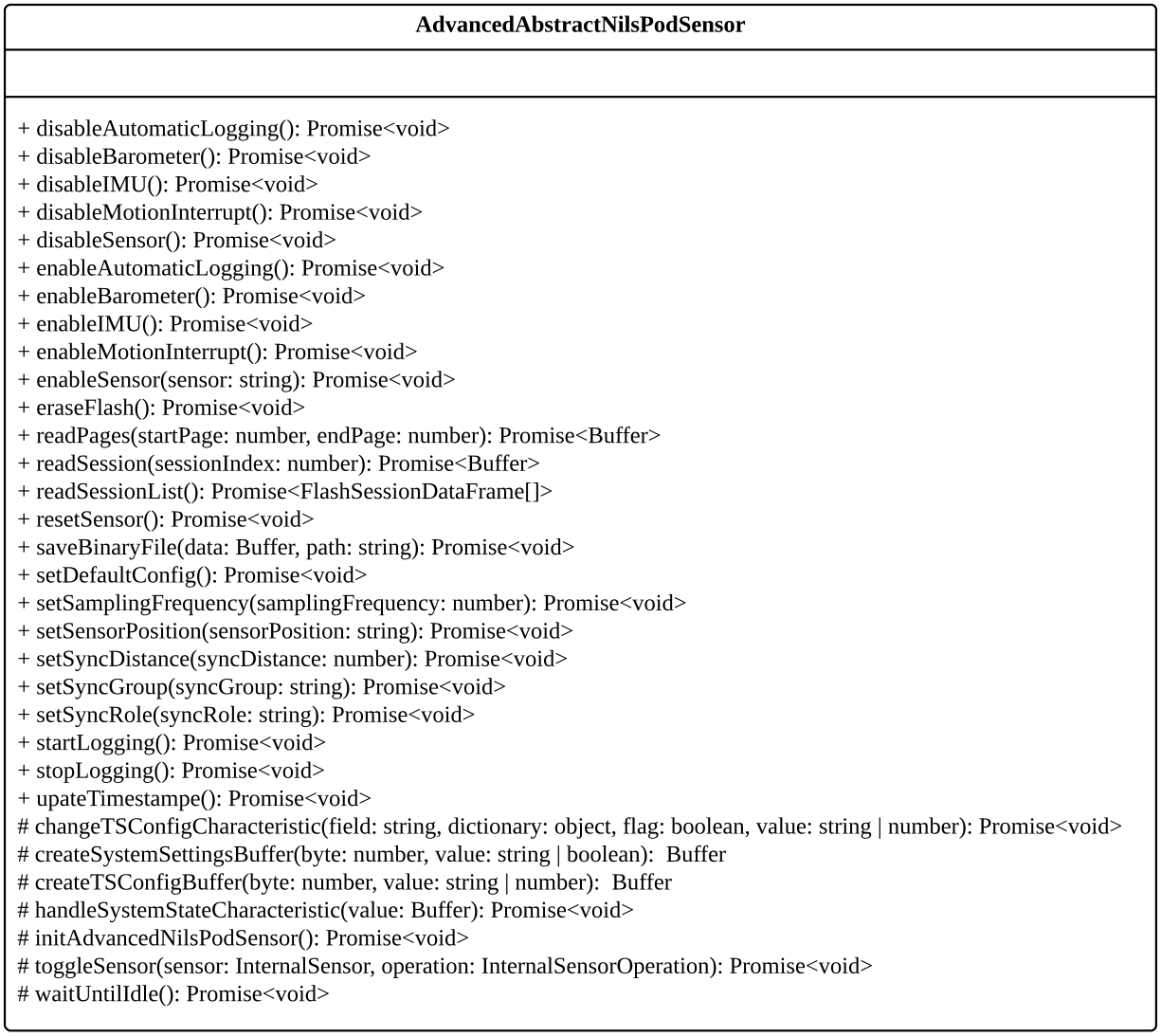

Figure A.2: The *AdvancedAbstractNilsPodSensor* class provides shared methods for NilsPod sensors that have an internal flash memory and are able to modify their configuration.

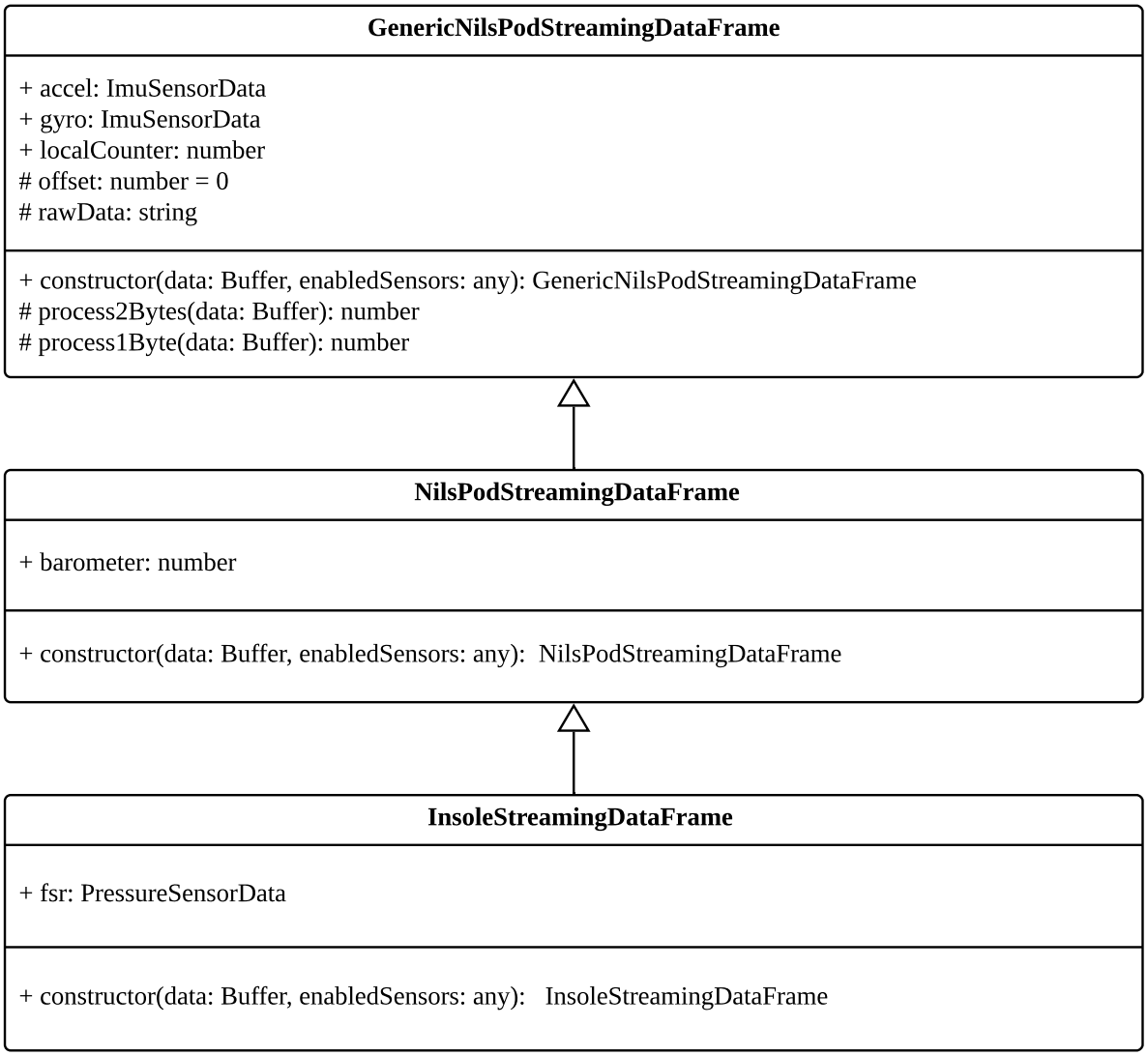

Figure A.3: During streaming, a NilsPod sensor class uses a corresponding data frame class to convert the incoming data samples.

```
1  const { AbstractNilsPodSensor, CsvWriter } = require('nodejs_sensorlib
       ');
2  AbstractNilsPodSensor.registerAll();
3
4  const streaming = async () => {
5    const csvWriter = new CsvWriter('example.csv');
6
7    csvWriter.on('error', error => {
8      sensor.stopStreaming();
9      console.error(error);
10   });
11
12   const sensor = await AbstractNilsPodSensor.find(3);
13
14   sensor.on('warning', warning => {
15     console.warn(warning);
16   });
17
18   sensor.on('data', async dataFrame => {
19     csvWriter.writeData(dataFrame); // enters a new row into the CSV
         file
20   });
21
22   await sensor.connect();
23   await sensor.startStreaming();
24
25   setTimeout(async () => {
26     await sensor.stopStreaming().catch(error => console.error(error));
27     csvWriter.end();
28     await sensor.disconnect().catch(error => console.error(error));
29   }, 5000); // stops streaming after 5 seconds
30 };
31 streaming().catch(error => console.error(error));
```

Listing A.1: The *CsvWriter* class can be used to store the received sensor data in a *CSV* file. A separate row is created for each data packet.

Glossary

ANT Adaptive Network Technology

API Application Programming Interface

ATT Attribute Protocol

BLE Bluetooth Low Energy

BR/EDR Bluetooth Basic Rate and Enhanced Data Rate

CSV Comma-separated Values

ECMAScript European Computer Manufacturers Association Script

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

FSR Force Sensitive Resistor

GAP Generic Access Profile

GATT Generic Attribute Profile

GPS Global Positioning System

GUI Graphical User Interface

HCI Host Controller Interface

hPa Hectopascal

IMU Inertial Measurement Unit

IoT Internet of Things

IPC Inter-Process Communication

ISM Industrial, Scientific and Medical

JSON JavaScript Object Notation

L2CAP Logical Link Control and Adaption Protocol

NFC Near Field Communication

OOP Object-oriented Programming

OS Operating System

RF Radio Frequency

RSSI Received Signal Strength Indicator

SIG Special Interest Group

SM Security Manager

UART Universal Asynchronous Receiver/Transmitter

UML Unified Modeling Language

UUID universally unique identifier

WIoT Wearable Internet of Things

WLAN wireless local area network

WPAN wireless personal area network

List of Figures

List of Tables

Bibliography

[Ali11] M. Ali, L. Albasha, H. Al-Nashash: *A Bluetooth low energy implantable glucose monitoring system*, in *2011 8th European Radar Conference*, 2011, pp. 377–380.

[Amf18] O. Amft: *How Wearable Computing Is Shaping Digital Health*, *IEEE Pervasive Computing*, , Nr. 1, 2018, pp. 92–98.

[And] Android Developers: *Android Bluetooth LE Documentation*, `https://developer.android.com/reference/android/bluetooth/le/package-summary`, accessed 20-February-2019.

[Bie14] G. Bierman, M. Abadi, M. Torgersen: *Understanding typescript*, in *European Conference on Object-Oriented Programming*, 2014, pp. 257–281.

[Blu] Bluetooth SIG Inc.: *GATT Specifications*, `https://www.bluetooth.com/specifications/gatt`, accessed 15-February-2019.

[Blu09] Bluetooth SIG Inc.: *Bluetooth Technology chosen as Health Device Standard: Continua Health Alliance Selects Bluetooth Low Energy Technology for Design Guidelines*, 2009.

[Blu10] Bluetooth SIG Inc.: *Specification of the Bluetooth System-Covered Core Package version: 4.0*, 2010.

[Bos18] Bosch Sensortec GmbH: *Data sheet: BMI160: Small, low power inertial measurement unit*, 2018.

[BT] Babel-Team: *Babel - JavaScript Compiler*, `https://babeljs.io/`, accessed 29-March-2019.

[Chu10] M. Chui, M. Löffler, R. Roberts: *The internet of things*, *McKinsey Quarterly*, Vol. 2, Nr. 2010, 2010, pp. 1–9.

[Dav15] T. Davenport, J. Lucker: *Running on data: Activity trackers and the Internet of Things*, *Deloitte Review*, Vol. 16, 2015.

[Dem13] A. Dementyev, S. Hodges, S. Taylor, J. Smith: *Power consumption analysis of Bluetooth Low Energy, ZigBee and ANT sensor nodes in a cyclic sleep scenario*, in *2013 IEEE International Wireless Symposium (IWS)*, IEEE, 2013, pp. 1–4.

[Eur13] European Computer Manufacturers Association and others: *ECMA-340: Near Field Communication - Interface and Protocol (NFCIP-1)*, 2013.

[Fac] Facebook Inc.: *React.js*, `https://reactjs.org/`, accessed 15-March-2019.

[Gar19] *Internet of Things*, `https://www.gartner.com/it-glossary/internet-of-things/`, 2019, accessed 10-February-2019.

[Gha15] M. Ghamari, H. Arora, R. S. Sherratt, W. Harwin: *Comparison of low-power wireless communication technologies for wearable health-monitoring applications*, in *2015 International Conference on Computer, Communications, and Control Technology (I4CT)*, 2015, pp. 1–6.

[Git] GitHub Inc.: *Electron.js*, `https://electronjs.org/`, accessed 7-March-2019.

[Gom12] C. Gomez, J. Oller, J. Paradells: *Overview and Evaluation of Bluetooth Low Energy: An Emerging Low-Power Wireless Technology*, *Sensors*, Vol. 12, Nr. 9, 2012, pp. 11734–11753.

[Guh10] A. Guha, C. Saftoiu, S. Krishnamurthi: *The essence of JavaScript*, in *European conference on Object-oriented programming*, 2010, pp. 126–150.

[Gup16] N. C. Gupta: *Inside Bluetooth Low Energy, Second Edition*, Artech House mobile communications series, Artech House, Norwood, 2. Issue, 2016.

[Hav18] M. Haverbeke: *Eloquent Javascript: A Modern Introduction to Programming*, No Starch Press, San Francisco, USA, 3. Issue, 2018.

[Her13] D. Herron: *Node Web Development*, Packt Publishing, Birmingham, 2nd ed.. Issue, 2013.

[Hir14] S. Hiremath, G. Yang, K. Mankodiya: *Wearable Internet of Things: Concept, architectural components and promises for person-centered healthcare*, in *Wireless Mobile*

Communication and Healthcare (Mobihealth), 2014 EAI 4th International Conference on, 2014, pp. 304–307.

[Ian] Ian Harvey: *bluepy Documentation*, `https://github.com/IanHarvey/bluepy`, accessed 12-February-2019.

[IHS19] IHS Markit: *The top trends of 2019: powered by transformative technologies*, `https://ihsmarkit.com/Info/0119/top-tech-trends-2019.html`, 2019, accessed 5-February-2019.

[Int13] International Organization for Standardization: *ISO/IEC 18092:2013: Information technology – Telecommunications and information exchange between systems – Near Field Communication – Interface and Protocol (NFCIP-1)*, 2013.

[Jam13] M. James: *What Is Asynchronous Programming?*, `https://www.i-programmer.info/programming/theory/6040-what-is-asynchronous-programming.html`, 2013, accessed 21-February-2019.

[Joh11] A. Johansson, W. Shen, Y. Xu: *An ANT based wireless body sensor biofeedback network for medical e-health care*, in *2011 7th International Conference on Wireless Communications, Networking and Mobile Computing*, 2011, pp. 1–5.

[Kap] W. Kapke: *Node.js ES2015 Support*, `https://node.green/`, accessed 29-March-2019.

[Kum17] P. Kumari, M. López-Benitez, G. M. Lee, T.-S. Kim, A. S. Minhas: *Wearable Internet of Things-from human activity tracking to clinical integration*, in *Engineering in Medicine and Biology Society (EMBC), 2017 39th Annual International Conference of the IEEE*, 2017, pp. 2361–2364.

[Lan10] J. Langer, M. Roland: *Anwendungen und Technik von Near Field Communication (NFC)*, Springer-Verlag, 2010.

[Let17] N. Lethaby: *Wireless connectivity for the Internet of Things: One size does not fit all*, *Texas Instruments*, 2017, pp. 2–10.

[Luc16] S. Lucero, others: *IoT platforms: enabling the Internet of Things: IHS TECHNOLOGY*, *White paper*, 2016.

[Mal12] K. Malhi, S. C. Mukhopadhyay, J. Schnepper, M. Haefke, H. Ewald: *A zigbee-based wearable physiological parameters monitoring system*, *IEEE sensors journal*, Vol. 12, Nr. 3, 2012, pp. 423–430.

[Men18] F. Mendoza, L. Alonso, A. López, D. and Patricia Arias Cabarcos: *Assessment of Fitness Tracker Security: A Case of Study*, *Proceedings*, Vol. 2, Nr. 19, 2018, pp. 1235.

[Mik16] MikroElektronika D.O.O.: *Bluetooth Low Energy - Part 1: Introduction To BLE*, `https://www.mikroe.com/blog/bluetooth-low-energy-part-1-introduction-ble`, 2016, accessed 20-February-2019.

[Min15] R. Minerva, A. Biru, D. Rotondi: *Towards a definition of the Internet of Things (IoT)*, *IEEE Internet Initiative*, Vol. 1, 2015, pp. 1–86.

[Mur12] C. Murphy: *Union Pacific Delivers Internet Of Things Reality Check: U.S.'s largest railroad uses sensors and analytics to prevent derailments, but it also shows where the next wave of innovation is needed.*, `https://www.informationweek.com/it-leadership/union-pacific-delivers-internet-of-things-reality-check/d/d-id/1105644`, 2012, accessed 06-February-2019.

[Nik18] A. Nikoukar, S. Raza, A. Poole, M. Gunes, B. Dezfouli: *Low-Power Wireless for the Internet of Things: Standards and Applications*, *IEEE Access*, Vol. 6, 2018, pp. 67893–67926.

[Obj] Object Management Group: *Unified Modeling Language*, `https://www.omg.org/spec/UML/`, accessed 06-February-2019.

[Omr10] A. H. Omre, S. Keeping: *Bluetooth low energy: wireless connectivity for medical monitoring*, *Journal of diabetes science and technology*, Vol. 4, Nr. 2, 2010, pp. 457–463.

[Opp11] C. A. Opperman, G. P. Hancke: *A generic NFC-enabled measurement system for remote monitoring and control of client-side equipment*, in *2011 Third International Workshop on Near Field Communication*, 2011, pp. 44–49.

[Pat17] Patrick Mannion: *Comparing Low-Power Wireless Technologies*, `https://www.digikey.de/en/articles/techzone/2017/oct/`

comparing-low-power-wireless-technologies, 2017, accessed 12-February-2019.

[Plo] Plotly: *Plotly.js*, https://plot.ly/, accessed 19-January-2019.

[Por] Portabiles GmbH: *Portabiles - Healthcare, Mobility, Sports*, https://portabiles.de/, accessed 13-February-2019.

[Ras] Rasmus Høhndorf Hummelmose: *BluetoothKit Documentation*, https://github.com/rhummelmose/BluetoothKit, accessed 19-January-2019.

[Ric] R. Richer, S. Gradl: *SensorLib: Java-based Android library*, https://github.com/mad-lab-fau/SensorLib, accessed 19-January-2019.

[San] Sandeep Mistry: *noble Documentation*, https://github.com/noble/noble, accessed 20-January-2019.

[Ste08] S. Stefanov: *Object-Oriented JavaScript*, Packt Publishing, Birmingham, 2008.

[Suh12] J. Suhonen, M. Kohvakka, V. Kaseva, T. D. Hämäläinen, M. Hännikäinen: *Low-Power Wireless Sensor Networks: Protocols, Services and Applications*, SpringerBriefs in Electrical and Computer Engineering, Springer US, Boston, MA, 2012. Issue, 2012.

[Swa12] M. Swan: *Sensor Mania! The Internet of Things, Wearable Computing, Objective Metrics, and the Quantified Self 2.0*, *Journal of Sensor and Actuator Networks*, Vol. 1, Nr. 3, 2012, pp. 217–253.

[The18] The Qt Company Ltd.: *Qt Bluetooth*, 2018.

[Til10] S. Tilkov, S. Vinoski: *Node. js: Using JavaScript to build high-performance network programs*, *IEEE Internet Computing*, Vol. 14, Nr. 6, 2010, pp. 80–83.

[van09] P. van Roy, others: *Programming paradigms for dummies: What every programmer should know*, *New computational paradigms for computer music*, Vol. 104, 2009, pp. 616–621.

[Zak09] N. C. Zakas: *Professional javascript for web developers*, Wrox professional guides, Wiley Pub, Indianapolis, Ind, 2nd ed.. Issue, 2009.

[Zha14] T. Zhang, J. Lu, F. Hu, Q. Hao: *Bluetooth low energy for wearable sensor-based healthcare systems*, in *Healthcare Innovation Conference (HIC), 2014 IEEE*, 2014, pp. 251–254.