

The present work was submitted to Lehr- und Forschungsgebiet Educational Technologies at DIPF

# Real-time Detection of Student Engagement Using Wearable Computers in a Learning Analytics System

Master-Thesis

Presented by

**Jian, Felix Darius**

6205885

First examiner: Prof. Dr. Hendrik Drachsler

Second examiner: M.Sc. George-Petru Ciordas-Hertel

Frankfurt, Friday 30<sup>th</sup> April, 2021



## **Erklärung**

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

I hereby declare that I have created this work completely on my own and used no other sources or tools than the ones listed.

---

Frankfurt, Friday 30<sup>th</sup> April, 2021



# Abstract

Big Data and learning analytics increase their influence in modern education with the aim to create ubiquitous learning environments. With the rise of technology in our lives and, therefore, in the learning space of students, engagement in learning activities becomes an increasingly important task. Detecting levels of engagement to be able to perform targeted interventions could be a promising approach to increase productivity. But the collection and evaluation of multimodal data from learning environments poses many infrastructural, ethical and legal challenges.

This thesis covers design and implementation of a learning analytics infrastructure on which real-time engagement detection and targeted intervention can be performed in the future. A suitable architecture was designed, relevant technologies and protocols evaluated in terms of the requirements and the core of the system was implemented, leaving analytical processing and engagement detection open for future projects. Security and privacy implications of such an infrastructure were evaluated to properly address them during implementation. It was concluded that not only privacy and data security is a challenging task, considering the sensitive data that is managed, but that specifically the involvement of smartphones and smartwatches may pose a privacy threat due to the proprietary nature of the operating systems and software. Finally, a performance evaluation was conducted for the developed system, resulting that the system meets the performance requirements of the use case.



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	Problem Statement . . . . .	2
1.3	Research Questions . . . . .	2
<b>2</b>	<b>Related Work</b>	<b>3</b>
<b>3</b>	<b>System Development</b>	<b>5</b>
3.1	Requirements Analysis . . . . .	5
3.1.1	Functional Requirements . . . . .	5
3.1.2	Non-functional Requirements . . . . .	6
3.1.3	Data Security and Privacy . . . . .	7
3.2	System Design . . . . .	9
3.3	Evaluation of Related Technologies . . . . .	10
3.3.1	Protocols and Data Formats . . . . .	11
3.3.1.1	Messaging Protocols: MQTT and CoAP . . . . .	11
3.3.1.2	What about WebSocket and HTTP? . . . . .	13
3.3.1.3	Data Formats: Avro, JSON and XML . . . . .	14
3.3.1.4	Specialized Formats: The Extensible Data Format . . . . .	15
3.3.2	Third-party Systems and Components . . . . .	15
3.3.2.1	Eclipse Mosquitto . . . . .	16
3.3.2.2	Lab Streaming Layer . . . . .	16
3.3.3	Infrastructure and Virtualization . . . . .	17
3.3.3.1	Kafka . . . . .	17
3.3.3.2	Docker . . . . .	18
3.4	Implementation . . . . .	18
3.4.1	System Overview . . . . .	18
3.4.2	Technical Component Description . . . . .	20
3.4.2.1	Wearable Companion . . . . .	20
3.4.2.2	Phone Companion . . . . .	22
3.4.2.3	Mosquitto MQTT Broker . . . . .	24
3.4.2.4	Kafka MQTT Connector . . . . .	25
3.4.2.5	Activity Detector . . . . .	27
3.4.2.6	Indicator Service . . . . .	28
3.4.2.7	Indicator UI . . . . .	28

3.4.3	Component Interaction . . . . .	30
3.4.4	Functional Description of User Applications . . . . .	32
<b>4</b>	<b>Performance Evaluation</b>	<b>37</b>
<b>5</b>	<b>Conclusion</b>	<b>41</b>
5.1	Key Findings . . . . .	41
5.2	Outlook . . . . .	42
<b>6</b>	<b>Source Code</b>	<b>45</b>
	<b>List of Tables</b>	<b>47</b>
	<b>List of Figures</b>	<b>49</b>
	<b>Bibliography</b>	<b>51</b>



# 1 Introduction

## 1.1 Motivation

In recent years, Big Data and machine learning technologies have found their way into a vast number of research fields. Learning and education is one of the fields that provide numerous challenges for which the use of Big Data technologies may prove beneficial. Lecturers are often times faced with large classrooms that need to be supervised, whilst simultaneously expected to impart complicated subjects. In the United States, particularly the fields of science, technology, engineering, and mathematics (STEM) have suffered from ineffective teaching for decades [Sey97, p. 34–35]. The majority of these issues in STEM education have not been resolved in recent years and still persist [SH19, p. 109–114, 440].

Students' attention is crucial for effective learning. If students' loss of attention occurs undetected, by both the students themselves and the lecturer, precious time and energy will go to waste in the classroom. Considering the vast amount of digital learning capabilities online, especially free or affordable online video courses, it becomes more and more difficult for university lectures to compete. After all, an online video can be started, paused and ended on demand. Students are able to schedule their sessions to achieve an optimal learning outcome for their own attention spans and habits. Moving towards ubiquitous learning environments may be necessary for institutions to offer a similar degree of flexibility.

One may argue that lectures are merely too long and could simply be shortened. However, research shows that common knowledge about fixed time intervals in which students can continuously maintain their attention may be flawed. Attention lapses may occur within the first minute of the lecture and are highly dependent on the level of interactiveness of the lecture [BFN10]. Therefore, detecting students' engagement is important to gain insights on how well lectures assess in terms of maximizing attention. Adding more break times is not enough to cope with the dynamic nature of attention cycles. While attention can not reliably be detected, detecting engagement is possible and may serve as an indicator of attention.

On the other hand, in a self-organized learning environment outside of lectures, mental breaks are also an important factor to increase productivity. More frequent, shorter breaks can be associated with higher academic success [SB00]. This could also explain the popularity of the pomodoro technique among students, which also proposes frequent, short breaks to increase productivity [Dio+16]. Moreover, most research and work outside of lectures will nowadays be conducted using a computer. Frequent short breaks from computer work have shown to have a positive effect on productivity [Hen+97].

To cope with dynamic attention cycles, real-time detection of engagement loss and feedback through machine learning technologies could provide a way for students and lecturers to perform context-based interventions when engagement plummets. Considering the self-independent nature of learning in higher education compared to first and second level education, self intervention may be the most suitable method for higher level education.

Wearable computers with integrated sensors have increased in popularity greatly in recent years. Considering the success of smartphones, it is not unreasonable to assume that wearable smart devices such as smartwatches will be established in the general population in near future. This development may present a great opportunity to leverage engagement detection in education to address students' attention cycles in education properly.

Many smartwatches ship with an integrated accelerometer and heart rate sensor. Hand movement and heart rate data could be monitored and analysed by a Learning Analytics System (LAS) to detect students' engagement in real-time and suggest context-based interventions to create a ubiquitous learning environment.

## 1.2 Problem Statement

The goal of this thesis is to design and develop an infrastructure for a Learning Analytics System that will receive sensor data over a network connection, preprocess it and stream it into the LAS using a suitable data format. The developed system shall be tailored to be easily extensible with components that use machine learning algorithms that detect students' engagement, and therefore meet appropriate throughput and latency requirements for this use case.

Furthermore, mobile applications for smartwatch and smartphone shall be developed to send sensor data via the smartphone to the web interface. Additionally, an appropriate visualization of collected sensor data that updates in real-time will be implemented.

## 1.3 Research Questions

- **RQ:** How may heterogeneous sensor data be streamed and analyzed in real-time through a learning analytics infrastructure?
- **RQ1:** What technologies and protocols are appropriate for a system capable of real-time analytical processing of sensor data?
- **RQ2:** What are the use cases for actors such as students and staff that shall be accommodated by the system?
- **RQ3:** What security mechanisms shall be implemented to assure data security and privacy for students?

## 2 Related Work

Some projects in the field of Predictive Learning Analytics have already made use of wearable computers and machine learning to monitor students engagement, performance and stress.

In an experimental research paper, smartwatches were attached to students' writing hand, using the integrated photoplethysmographic (PPG) and accelerometer sensors to monitor heart rate and hand movement respectively [ZOJ17]. However, the focus of this work was the data model and machine learning algorithm to predict engagement, not the integration in a LAS. Architectural considerations were not in the scope. Furthermore, real-time detection and feedback were not intended, as opposed to this thesis.

Another paper monitored heart rates of students with activity wristbands to predict performance and stress for self-regulated learning [Mit+17]. Similar to this thesis, a system that processed data in real-time was designed and implemented, but the focus was only on performance prediction in the context of self-regulated learning. In contrast, this thesis assumes various learning contexts and also discusses context-based interventions. Additionally, machine learning was a major part of the work, whilst this thesis rather focuses on data collection, preprocessing and integration into a multi-purpose LAS.

Other authors propose a generic internet of things (IoT) architecture for ubiquitous and context-aware learning [SZ20]. Multiple variants with different protocols and technologies were acquired and their advantages and disadvantages discussed. The four variants were designed with Constrained Application Protocol (CoAP), Message Queuing Telemetry Transport (MQTT), Advanced Message Queuing Protocol (AMQP) and Extensible Messaging And Presence Protocol (XMPP). The paper concludes that all architecture variants meet the requirements for most ubiquitous learning systems, with slight advantages and disadvantages depending on the requirements. In contrast, the scope of this thesis is to develop one infrastructure for later use in production, while the insights of the four designed variants will be considered.

In another paper, state of the art of wearables in education are in the focus [ACS17]. Current commercial wrist wearables are evaluated in terms of suitability for educational purposes. Among others, sensor quality and privacy concerns are discussed. This thesis likewise discusses privacy concerns, but different commercial wrist wearables are not evaluated. Furthermore, the core of this thesis is system development, which the paper does not address.

Hand activity recognition is another important aspect of research for this thesis. Experimental research was conducted where a smartwatch accelerometer was overclocked to allow recognition of hand *gestures* [LXH16]. The authors built upon the findings in their future work, where hand *activities* could be recognized by a machine learning algorithm [LH19]. The findings of these two projects are of great significance for the implementation in this thesis, especially

since the kernel module to overclock the accelerometer was made available publicly online. However, the work only includes an experimental setup, as opposed to this thesis, where the infrastructure shall meet the requirements of later use in production.

Another paper details the implementation of a smart classroom system that aimed at detecting interest in students during a lecture [Gli+15]. An accelerometer was used, but only to monitor the lecturers' motions. For the students, motion extraction was conducted with a camera and image processing techniques, sound extraction with a microphone with a feature extraction algorithm. Predictive analysis as well as presentation was implemented in Matlab. The difference of the work to this thesis is that the former used accelerometer data to analyse the motions of lecturers, whilst the latter aims to provide an infrastructure to analyse those of the students.

Predictive learning analytics requires multimodal data to be considered. Authors formulated the Multimodal Learning Analytics Model, which explores how machine learning can be combined with multimodal data and aligns the terminology of machine learning and learning science [Mit+18]. Considering wearable trackers, it is discussed briefly how they fit into the bigger picture of multimodal learning analytics. A shortage of studies that focus on feedback and interventions for the learner and their learning process was identified. This thesis has a more practical focus in designing and implementing a system as opposed to the formulation of a model.

Transmission security with wearables is also a highly relevant topic when considering a world of interconnected devices and IoT. Secure group key generation was used in a paper to securely transmit accelerometer data [Rev+17]. Generation of a private group key for all participating devices as well as private key distribution was implemented. The paper describes the method to be found as appropriate for practical application. This thesis is likewise concerned with transmission security of wearables, but relies on certificate authorities to solve the issue of key distribution.

Security and privacy for e-learning systems are discussed in other work by examining security vulnerabilities of e-learning systems [CN12]. While conceptually an e-learning environment is not the same as a ubiquitous learning environment, the same vulnerabilities in the technology used should apply. This thesis focuses rather of vulnerabilities of a ubiquitous learning environment.

## 3 System Development

This chapter covers all aspects of the system development process for the LAS, from requirements analysis to implementation. First, the requirements are outlined in the requirements analysis, then the resulting design is presented. Before the technical implementation is described in more detail, a separate section evaluates related technologies to choose the most appropriate candidate for the implementation.

### 3.1 Requirements Analysis

This section is split into subsections for functional and non-functional requirements as well as an own subsection for data security and privacy requirements. Although data security and privacy may be categorized as non-functional requirements, the importance in a LAS — where trust is imperative to convince users to allow sensor data being acquired from them — warrants its own subsection.

#### 3.1.1 Functional Requirements

Gathering the functional requirements for students and staff addresses *RQ2: What are the use cases for actors such as students and staff that shall be accommodated by the system?* The objective is to learn how actors need to interact with the system to collect the required data for analytical processing, and how the data may be presented.

(a) **Engagement detection**

Detect students' engagement by analysing multiple parameters from heterogeneous data sources: e-learning activity to detect student inactivity, smartphone usage to detect procrastination and hand gestures to detect engagement. Implementation of the engagement detection logic is not intended for this project, but the system shall facilitate sensor data recording, transmission as well as some basic processing to show that it meets necessary performance requirements for engagement detection.

(b) **Trigger learning sessions**

Learning sessions must be triggered to initiate detection. Various triggers may be desirable, such as e-learning login, calendar events or student hand gestures.

(c) **Context recognition**

The context of learning sessions should be detected to be able to suggest an appropriate

method of intervention. For instance, a home learning session can always be stopped by the student, but suggestion a break in the middle of a university lecture may be inappropriate. Light of environment, noise and session duration could be used. Proper implementation of context recognition goes beyond the scope of this project, but it shall be considered when designing the system.

(d) **Presentation**

Results of the analysis must be presented in a way that is easy and fast to understand, most likely using graphical charts. A graphical web user interface shall be used to visualize real-time results.

The functional requirements for the system address the first process of the Multimodal Learning Analytics Model (MLeAM), sensor capturing, which includes the design and implementation of a hardware and software infrastructure for data collection [Mit+18].

### 3.1.2 Non-functional Requirements

#### Design

- (a) *Interoperability*: The system must interoperate with other systems, such as the Moodle e-learning. Even more systems may be used in the future.

#### Adaptation

- (a) *Scalability*: The system must be able to scale to an increasing number of students and learning sessions that run in parallel.
- (b) *Modifiability*: Components of the system may be modified in the future, such as analytical components; more specifically, engagement detection components may be altered or replaced.
- (c) *Extensibility*: The system must be extensible so that more parameters can be analysed for predictive analysis in the future. More sensors with their respective analytical components may be added to the system.

#### Viability

- (a) *Performance*: The system must maintain fast response times to be able to perform real-time analysis of streamed data. Further, a high throughput must be achieved.
- (b) *Robustness*: The system must be able to remain functional even if errors occur in single components. For instance, if corrupted data is transmitted from a sensor and the responsible component for the analysis fails, other parameters must continue to be analysed and predictions must be performed without the missing parameter.
- (c) *Security*: The system must transmit sensor data securely to ensure privacy of students. Additionally, access to the system must be secured by suitable authentication and authorization mechanisms.

#### Sustainment

- (a) *Availability*: The system must be available during all learning sessions a very low tolerance for down-times. Real-time analysis would be affected severely by down-times.

- (b) *Reliability*: The system must recover from failures quickly. If a component fails, it must be restored or replaced swiftly.

### 3.1.3 Data Security and Privacy

Data security and privacy are essential aspects to consider for a system that is intended to be used by numerous students. Analyzing data security and privacy requirements addresses *RQ3: What security mechanisms shall be implemented to assure data security and privacy for students?*

Considering the fact that sensor data is gathered using wearable devices, which are usually attached to students throughout the day, even more security and privacy challenges arise. Not only is sensible data provided by students explicitly through user input but also implicitly through the sensors. If the sensors are activated outside of learning sessions — either accidentally by the students or because of a software or hardware defect — behavioural patterns outside of learning sessions may be evaluated without students' intent.

Users' trust — in the context of this thesis usually students' trust — in the system is essential if sensor data is acquired continuously from a physically attached device. Losing the trust of the students may lead to a rejection of the wearable software by the students, which would hinder the LAS to establish itself at an academic organization. Unfortunately, most technologies in the field of ubiquitous learning lack consideration to privacy and security concerns and rather emphasize course development and delivery [CN12].

An end-user licence agreement (EULA) should inform the student what data is gathered by the application and for what purposes. However, users may still experience a sense of “creepiness” when using the application. A solid base of trust must be established between the user and the application provider, which is not likely to happen through a detailed EULA alone. Users have shown to feel a sense of “creepiness” if applications do more than they seem at first glance, regardless of the EULA content [Shk+14]. In the case of the LAS frontend android applications, an example for application behaviour that triggers a sense of “creepiness” could be recording sensor data out of learning sessions, which is not expected by users.

It is therefore crucial to ensure secure transmission and storage of data throughout the system. All publicly exposed endpoints must implement authentication and encryption. Stored data must not be exposed publicly by other means than the mentioned secured endpoints. Within the backend system, data may be transferred unencrypted for performance reasons.

Furthermore, Third-party libraries and web application programming interfaces (APIs) must be examined thoroughly before being utilized. Any software that is not open source comes with a privacy threat that must be evaluated accordingly to decide if the trade-off is acceptable. Especially mobile phones and wearable devices are prone to privacy issues, since software is usually closely entangled with the vendor of the operating system such as Apple or Google.<sup>1</sup> Sensor and other System APIs are usually provided by the vendors themselves, potentially posing a conflict of interest in terms of privacy.

Considering the privacy scandals of Apple and Google in recent years, users may already be

---

<sup>1</sup> Often times the vendor of the operating system is even the vendor of the device itself, as with the Google Pixel phone or most Apple products.



reluctant to use devices and operating systems to track learning behaviour. Two examples of these scandals include the 2018 Google Data Breach<sup>2</sup> or the Apple Siri Scandal<sup>3</sup>. As a result, not harming the students' trust is a crucial consideration for the development of android client applications.

Although open source software is not inherently more secure than proprietary software, open source software has the potential for a large increase in trustworthiness [HKP02]. Privacy is not as simple as comparing the amount software vulnerabilities or response time to said vulnerabilities in open source and proprietary software. The software itself must be transparent so that a formal proof of adherence to privacy regulations is feasible [HKP02]. Using best-in-class security measures within a software is meaningless if the software is a black box; what really happens with the private data is intangible. The danger of illegitimate data collection is twofold: not only may the data be misused by the vendor; even if the vendor acts faithfully, unknown vulnerabilities may expose data to attacks when the data was never to be collected in the first place.

Nevertheless, software vulnerabilities alone are a security threat and therefore also a privacy threat for users, regardless of the software not mishandling private data. Even if there is more to privacy than software security — as established above — software must be secure to support privacy. From this perspective, both proprietary and open source software have their strengths and weaknesses. Open source software allows attackers to closely examine the source code for weaknesses; on the other hand, open source software has an advantage over proprietary software with shorter response times to those weaknesses [Bou05].

Due to the transparency and privacy advantages as well as the short response times to vulnerabilities, open source software should be favoured when private user data is involved. However, open source is merely a solid basis and does not make the software more secure. It is also crucial that the source is evaluated by a sufficient amount of experts and a large Internet community [HKP02].

Unintentional leaks of personal data by companies are a common threat scenario, as proven by data leaks from various companies [Ari+15]. These incidents show how vulnerable private user data actually is against malicious attackers. Especially IoT and wearable devices are vulnerable, not only on a software but also on a hardware level, as security is often neglected or not prioritized enough by manufacturers [Ari+15].

However, apart from the threat of malicious attackers exploiting hardware and software security vulnerabilities, the intentional collection of said data by corporations with the intent of monetization may also increase with the rise of IoT and wearable devices. An example is the apparent plan of Nest to share data from the Nest Thermostat with energy providers [Ari+15]. The magnitude of data collection by corporations is difficult to estimate, but it clearly is not to be taken lightly in terms of user privacy.

<sup>2</sup> <https://theguardian.com/technology/2018/oct/08/google-plus-security-breach-wall-street-journal> (accessed on: Thursday 30<sup>th</sup> April, 2020)

<sup>3</sup> <https://theguardian.com/technology/2019/jul/26/apple-contractors-regularly-hear-confidential-details-on-siri-recordings> (accessed on: Thursday 30<sup>th</sup> April, 2020)



## 3.2 System Design

The system's design builds upon the insights of the requirements analysis in Section 3.1. In the following, it is first shown how the proposed system will work in terms of the functional requirements; then, design choices caused by the non-functional requirements – including data security and privacy – are highlighted. Design choices regarding data security and privacy address *RQ3: What security mechanisms shall be implemented to assure data security and privacy for students?* All further details on the design should be seen in relation to the component diagram of the system design, as seen in Figure 3.2.

For engagement detection and context recognition through wearable sensor data, the sensor data is first transmitted from the student's wearable device to the phone wirelessly. The phone, in turn, transmits the data to the sensor broker via a publish endpoint on the network. The sensor broker exposes a subscribe endpoint for other components to read sensor data. The messaging connector connects to that endpoint to feed the data into the messaging system.

The messaging system is the core of the data processing infrastructure. It acts as a message queue with a publish-subscribe interface; analytical processors read from it, perform the processing and write back to it asynchronously. After the messaging connector fed new data into the messaging system, all processing happens isolated through the messaging system and the processors.

The activity detector component subscribes to raw sensor data topics, such as accelerometer data for hand gestures, performs various processing steps and publishes the data to the statistics topic. The activity detector calculates statistics such as average messages per second to prove that the throughput necessary to detect hand gestures can be reached by the system.

To present processing results, the components indicator service and indicator UI are used. The indicator service subscribes to the statistics topic, maps the data to a suitable format for the presentation layer and exposes it through a subscribe endpoint on the network. The indicator UI encapsulates the logic to render the results in a graph.

Triggering learning sessions happens on the wearable device only and is not yet persistent.

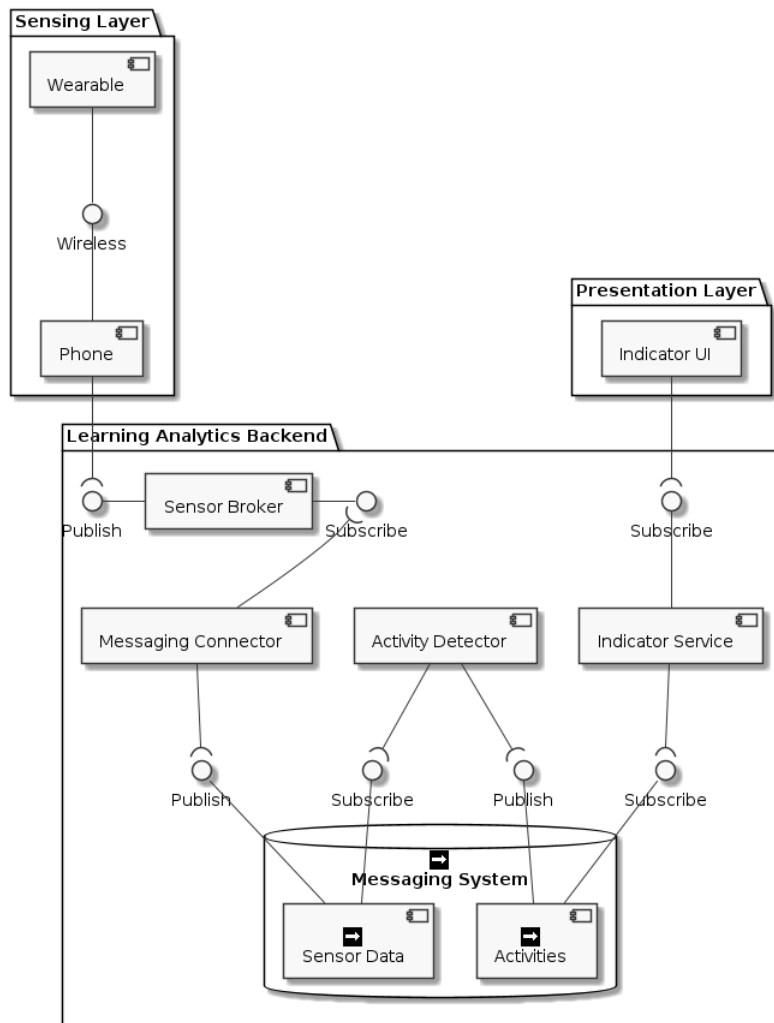
Multiple compact, loosely-coupled components ensure modifiability and extensibility. Message processors such as the activity detector only connect to the messaging system and perform analytical processing in the domain they are responsible for. They can easily be replaced or modified without touching other components. Furthermore, inter-component communication through the publish-subscribe based messaging system eases the integration of new analytical processors that subscribe to existing data streams. Additionally, adding new types of data streams does not influence existing components.

The design choice of many small components may also improve scalability; if a bottleneck is identified, multiple instances of the respective component may be installed to cope with the load. This, however, depends on the messaging system that is used and if it supports such distribution of work.

From a security perspective, the sensor broker and indicator service are the only backend components that are exposed to the outer world. If the sensor broker is compromised, it is only possible to listen to incoming data from the sensing layer; there is no persistent data that

can be accessed. The indicator service only exposes data from preconfigured topics, hence it is not possible to access other data from the messaging system if the indicator service is compromised. Similar to the sensor broker, the realistic threat is that illegitimate actors listen to live data.

Transmission security is only critical for the connection between phone and sensor broker, wearable and phone as well as indicator UI and indicator service. Other transmissions are only accessible from within the backend infrastructure.



**Figure 3.1:** Component diagram of the system design.

### 3.3 Evaluation of Related Technologies

This section assembles an appropriate technology stack for the implementation of the system. Candidates for protocols and data formats, third party system and components, software libraries and frameworks as well as infrastructure technologies are evaluated for the use case at

hand. The most suitable candidates are added to the technology stack. The resulting technology stack is the answer to *RQ1: What technologies and protocols are appropriate for a system capable of real-time analytical processing of sensor data?*

### 3.3.1 Protocols and Data Formats

Various standardized protocols and data formats may be useful to simplify the systems' development process and ensure maintainability by future developers. As the establishment of custom protocols and data formats is an expensive task, the utilization of established industry standards and protocols is preferred for the development of the LAS.

In this section, candidate protocols and data formats suitable for high throughput, low latency data transmission are gathered and, if multiple candidates are found, compared to one another. Comparing suitable protocols is not necessary for all interactions in the system. Some Frameworks that are used already implement a certain protocol, thus taking away the need to make a decision in this regard. On the other hand, choosing a data format is often independent of the underlying framework or technology and affects how data is represented throughout the system.

As discussed below in Section 3.3.3.1, Kafka will be integrated as a streaming platform for inter-component communication on the server side. The protocol used is an internal detail of the Kafka platform.<sup>4</sup> Therefore, protocols will not be discussed for communication within the server infrastructure. The main concern is high throughput, low latency data transmission from the mobile clients to the server. The protocols discussed will be concerned with specifically this client-server communication use case in mind.

Two suitable protocols were identified for the implementation of client-server communication between the Sensing Layer and the Learning Analytics Backend: MQTT and CoAP. Both are lightweight application protocols, appropriate for data transmission from the sensing layer to the analytical backend [Car+13].

Changing data formats among subsystems, on the other hand, leads to increasing complexity, since a mapping between the various formats must be implemented. Henceforth, for the sake of simplicity, data formats are evaluated for system-wide use. An important consideration for choosing the data format is the ease of integration with Kafka.

#### 3.3.1.1 Messaging Protocols: MQTT and CoAP

The application layer protocols MQTT and CoAP are built upon different transport layer protocols, MQTT on Transmission Control Protocol (TCP) and CoAP on User Datagram Protocol (UDP). To compare MQTT with CoAP, it is important to be aware of the differences in the underlying protocols first.

TCP is reliable and connection oriented; UDP, on the other hand, is unreliable and connectionless. A TCP connection must be setup by performing a three-way handshake between sender and receiver. A UDP connection, on the other hand, can be established without setup,

<sup>4</sup> <https://kafka.apache.org/protocol> (accessed on: Thursday 30<sup>th</sup> April, 2020)

merely by starting to transmit data to the receiver [LRS18]. The unreliable nature of UDP allows for faster performance compared to the more reliable TCP.

TCP and UDP by themselves both provide no protocol-level communications security such as encryption. Communications security may be achieved by using protocols that are built on top of them, specifically Transport Layer Security (TLS) for TCP and Datagram Transport Layer Security (DTLS) for UDP.

MQTT is a machine-to-machine IoT connectivity protocol that was designed as a lightweight publish-subscribe messaging transport [CC14]. MQTT covers features such as quality of service, retained messages, as well as “last will and testament”, which means the broker will message other clients if another client disconnects.<sup>5</sup> The implementation of the TCP protocol makes MQTT the more reliable application level protocol, but in turn the less performant.

As for privacy and authenticity, MQTT allows TLS encrypted TCP connections on the protocol level and supports username and password authentication by providing fields for username and password through the CONNECT packet. The authentication logic itself is the responsibility of the respective implementation of MQTT [CC14].

CoAP is also a machine-to-machine connectivity protocol, but is specialized for constrained nodes and networks. CoAP offers low overhead and simplicity for devices with low-power, lossy networks and low battery capacities. As opposed to MQTT — which follows a publish-subscribe model — CoAP is a request-response protocol, similar to Hypertext Transfer Protocol (HTTP) [SHB14].

Protocol-level security is provided by CoAP through its optional DTLS support. However, CoAP does not provide any means for authentication or authorization — other than the capabilities provided by DTLS, such as client certificates. Username and password authentication or token based authentication must be implemented around the CoAP implementation [SHB14].

Both MQTT and CoAP were designed for constrained sensor devices; however, they have been shown to be appropriate for smartphone applications as well, providing a specialized protocol for sensor data transmission while reducing network overhead and battery consumption [Car+13].

The low overhead of the UDP protocol leads to a performance advantage of CoAP over MQTT. CoAP has been shown to outperform MQTT in terms of bandwidth usage and round trip time [Car+13]. In the context of activity recognition, this performance benefit may be significant to reach a threshold of data frames per second to reliably recognize motions.

Both MQTT and CoAP have shown to fulfil the requirements for ubiquitous learning environments, in terms of performance, security, resource consumption and reliability. The decision between the protocol depends on the use case and will be made by examining closely the requirements of this project. It is not possible to rule out one or the other just in terms of general suitability for the implementation of an LAS [SZ20].

In terms of security, the capabilities of MQTT seem more appropriate for a LAS that must authenticate and authorize large numbers of users. MQTT implementations typically leverage the username and password field to authenticate clients based on either a username and password combination or a provided token through the username field. As mentioned above, CoAP

<sup>5</sup> <https://hivemq.com/blog/mqtt-essentials-part-9-last-will-and-testament> (accessed on: Thursday 30<sup>th</sup> April, 2020)

provides not built-in solution for password based authentication.

Another important consideration is the existence of ready-to-use implementations of the protocols to reduce the development effort. Whilst a number of CoAP implementations could be found on the Internet, MQTT implementations seem to have a significantly larger user base and online documentation available. For instance, the Mosquitto MQTT broker has ~4100 stars on GitHub, while the most prominent CoAP project, node-coap, has ~400 stars.<sup>67</sup>

Furthermore, MQTT server implementations have the notion of being shipped as “brokers”, which are ready-to-use components that provide features such as authentication out of the box. As for documentation, HiveMQ offers a lot of online support that is in many cases applicable for MQTT brokers in general. CoAP implementations seem to be more low-level, usually require implementing a server in a certain programming language, provide less documentation and either do not address application-level authentication at all or as a related project that is not straightforward to integrate.

Henceforth, specifically because of the built-in advanced security features as well as the publish-subscribe architecture, MQTT was chosen over CoAP as a sensor transmission protocol for this system. MQTTs reliability is not primarily a factor for this decision in the context of activity recognition, but could be of later use for other use cases in the larger scope of engagement detection. Table 3.1 summarizes the results of the comparison.

**Table 3.1:** Summary of MQTT and CoAP comparison.

Feature	MQTT	CoAP
Key feature	Reliability	Performance
Security	Authentication and authorization	Not built-in
Deployment	Pre-built broker	Libraries to implement
Specialization	Publish-subscribe	Low power, lossy network
Constrained devices	x	x
TLS	x	x
Resources	Numerous reference projects	Scarce
Largest community	Mosquitto	node-coap
GitHub stars	~4100	~400

### 3.3.1.2 What about WebSocket and HTTP?

Apart from protocols specialized for IoT device communication, such as CoAP or MQTT, the naive approach should still be considered; namely using well established application layer protocols for data transmission, such as WebSocket or HTTP.

HTTP follows — just like CoAP — a request-response model. However, it has a significantly larger overhead, which makes it not feasible to use for high-throughput data transmission, especially on constrained devices or mobile, battery powered devices in general [Kar+15]. Thus, if a request-response model is desired, CoAP beats HTTP easily for the relevant use case.

<sup>6</sup> Star count from <https://github.com/eclipse/mosquitto> (accessed on: Sunday 18<sup>th</sup> October, 2020)

<sup>7</sup> Star count from <https://github.com/mcollina/node-coap> (accessed on: Sunday 18<sup>th</sup> October, 2020)

Thus, HTTP is not an appropriate protocol for the use case at hand and will not be used for sensor data transmission.

WebSocket does not fall into the category of publish-subscribe protocols, as opposed to MQTT. Neither is it a request-response protocol, as compared to CoAP. The WebSocket protocol is rather used for data exchange with asynchronous full-duplex communication, meaning all participants can send or receive data within a single connection. The connection is initiated by a client by sending a handshake to the WebSocket server. Websocket Application Messaging Protocol (WAMP) may be used to provide publish-subscribe capabilities for WebSocket [Kar+15].

Since WebSocket is not applicable for resource constrained devices [Kar+15], it provides less flexibility in terms of what devices may be used to record sensor data. With a CoAP backend or an MQTT broker, also resource constrained devices may be used to transmit sensor data. That is why WebSocket was not chosen as a data transmission protocol for sensor data.

### 3.3.1.3 Data Formats: Avro, JSON and XML

Common interface definitions are crucial for the development of a messaging system, providing a common, agreed-upon data format that applications within the messaging system use for communication. The uniform format allows for using compact binary data formats such as Avro. [Kum17, p. 8].

It is good practice to limit the communication to only one protocol that is used consistently across applications. This creates a single integration point, separating data format conversion from processing logic. This prevents applications from becoming a maintenance burden, not needing to adjust to formats of each newly integrated data source [Kum17, p. 245].

Kafka is agnostic of any data format, which allows for a wide range of data formats to be used. Serializers and deserializers can be provided to producers, consumers and connectors at will [Nar17]. This flexibility warrants a closer examination of the requirements to make the appropriate choice for a data format with regard to the use case.

The most commonly used formats for data exchange over a network are Extensible Markup Language (XML) and Javascript Object Notation (JSON). Both are well established data formats with their own advantages and shortcomings. However, a binary format is better suited for a high-throughput, low-latency messaging system, due to the reduced message size. A common choice in the Kafka world is Apache Avro, for which Confluent Kafka provides serializers and deserializers that are ready to use.

Another benefit of Avro is schema evolution: If the schema of incoming data changes, the application can adapt to the changes without the need to be changed [Nar17, p. 55]. Furthermore, when using Kafka and Avro with a schema registry, the message size can be further reduced [Nar17, p. 56].

Although Avro is sufficient as an internal format for the Kafka messaging system, data must still be transmitted to the server before it is injected into the messaging system. Hence, JSON and XML are compared to decide which format will be used for data exchange between clients and server.

XML is a markup language that — due to its universal nature — has a wide range of ap-



plications. It is, thus, sometimes called the “holy grail” of computing. XML is designed to be extensible, human- as well as machine-readable and to be used across the internet. Object serialization for data transfer is just one use case for XML [Nur+09].

JSON, on the other hand, is specifically designed as a data exchange language. Just as XML, it is human- and machine-readable. Due to its compact nature and specialized design, it outperforms XML in terms of data exchange greatly [Nur+09].

JSON excels at compactness and performance, whereas XML provides namespace support and extensibility, features that JSON lacks. JSON is flexible and uses scoping through objects instead of namespacing, which is supposed to mitigate the missing features to some extent [Nur+09].

Due to the high-throughput requirements that the system faces, JSON seems to be the appropriate choice for a data exchange format. Moreover, Avro provides a *JsonDecoder*, which may be used to read a JSON record into an Avro record if the schema is known. There is no XML support implemented yet.<sup>8</sup> On a final note, Avro is configured through schema files written in JSON syntax, which means using XML as a message exchange format would introduce yet another format to work with.

Since JSON is the more compact, performant solution and — as opposed to XML — can be used in conjunction with Avro, we chose it as the primary data exchange format.

#### 3.3.1.4 Specialized Formats: The Extensible Data Format

Extensible Data Format (XDF) is a general-purpose format for time series data from multiple channels where extensive meta information is associated with the data.<sup>9</sup> It can handle data with a high sampling rate and is used within the Lab Streaming Layer (LSL) system.<sup>10</sup>

Unfortunately, not many reference projects using XDF were found on the Internet, suggesting a rather small community. This also makes troubleshooting more difficult when using this format. Furthermore, it is most likely not possible to use the format as a data format with the Kafka data pipeline, as XML metadata is mixed with unstructured record data that may even be in binary format. This means the only reasonable use case would be the usage in conjunction with the LSL to transmit sensor data to the server.

### 3.3.2 Third-party Systems and Components

Third-party components have the potential to help to save development effort and resources. That is, if the respective third-party solutions accommodate the defined requirements. As a consequence of the data security and privacy requirements, open source solutions are generally preferred. Solutions are furthermore evaluated with respect to the results of the evaluation of protocols and data formats, the requirements analysis and design choices.

<sup>8</sup> <https://issues.apache.org/jira/browse/AVRO-1294> (accessed on: Thursday 30<sup>th</sup> April, 2020)

<sup>9</sup> <https://github.com/sccn/xdx> (accessed on: Thursday 30<sup>th</sup> April, 2020)

<sup>10</sup> <https://github.com/sccn/labstreaminglayer> (accessed on: Thursday 30<sup>th</sup> April, 2020)

### 3.3.2.1 Eclipse Mosquitto

Eclipse Mosquitto is a lightweight, open source MQTT message broker.<sup>11</sup> Since it can run on a wide range of devices due its lightweight approach and because of its open source licence, it is widely used and has a large user base, as shown in Section 3.3.1.1. Although it may lack some more sophisticated features, it provides a simple way to get a project started through a small proof of concept.

Features that Mosquitto lacks are — most notably — scalability through distribution and identity management. In contrast to the Kafka messaging system, Mosquitto does not follow a distributed approach. Whilst it is possible to deploy multiple, load balanced Mosquitto brokers in a distributed system, it does not offer a built-in solution. Load balancing and distributed deployment have to be implemented around the Mosquitto broker manually.

In terms of identity management, Mosquitto only offers username and password authentication through the username and password fields required by the MQTT specification. User credentials are managed through plain text files, not providing any integration with the identity management of a system. However, open source plugins for Mosquitto exist that may be leveraged to integrate the brokers authentication and authorization mechanisms with most popular authentication and authorization frameworks.

Although some convenience features are not provided by Mosquitto, it is still a proven and widely used solution that seems to be an appropriate choice for the first stage of the system. Since the MQTT broker is only a small part of the system, it can be replaced as the system grows and requirements change without much effort. Depending on future requirements, building more features around Mosquitto might as well be a viable option.

### 3.3.2.2 Lab Streaming Layer

The LSL is a system for the collection of time series data in research experiments. Networking, time synchronization, collection and disk recording is handled by the system.<sup>12</sup> The core transport library *liblsl* is cross-platform and may be included into an Android application through the *liblsl* Java interface.<sup>13</sup>

An advantage of this system is that it provides an out-of-the box solution for complex issues such as time synchronization that have otherwise to be addressed separately. On the other hand, the LSL seems to be geared towards research experiments rather than data transmission in a production environment of an application. This does not mean that it is not suited for that purpose; however, it means that no reference products and applications exist that prove the suitability in terms of aspects such as reliability and security.

Furthermore, online research did not yield any results on the integration with a Kafka messaging infrastructure. The associated Lab Recorder merely writes results to the file system. This seems far less flexible compared to a publish-subscribe solution such as an MQTT broker. Since the XDF data format is not compatible to Kafka, as outlined above, sensor records would have to be translated to regular XML or JSON records.

<sup>11</sup> <https://mosquitto.org> (accessed on: Thursday 30<sup>th</sup> April, 2020)

<sup>12</sup> <https://github.com/scn/labstreaminglayer> (accessed on: Thursday 30<sup>th</sup> April, 2020)

<sup>13</sup> <https://github.com/labstreaminglayer/liblsl-Java> (accessed on: Thursday 30<sup>th</sup> April, 2020)



Because of these drawbacks for the particular use case, the LSL and XDF will not be used for the implementation of the system in this project.

### 3.3.3 Infrastructure and Virtualization

We built the system on two core infrastructure technologies: *Apache Kafka* and *Docker*. Since the core infrastructure of the system will impact greatly if the non-functional requirements of the system may be met, both technologies are inspected for their various traits that address the non-functional requirements defined in Section 3.1.1.

#### 3.3.3.1 Kafka

Apache Kafka is a distributed streaming platform typically used for building real-time data pipelines and streaming apps.<sup>14</sup>

Kafka meets the following non-functional requirements:

- (a) **Scalability:** Kafka is horizontally scalable; it is possible to add more Kafka brokers or even entire clusters if load increases.<sup>15</sup>
- (b) **Modifiability:** Components can easily be modified or replaced as long as they adhere to the Kafka protocol. No interdependent components.
- (c) **Extensibility:** To extend functional capacity of the system, new components can be added and connected to the Kafka pipeline without impacting other components.
- (d) **Performance:** Suitable for real-time data pipelines. Transforming and reacting to data streams is intended. Offers high throughput and low latency.<sup>16 17 18</sup>
- (e) **Robustness:** Fault-tolerant storage by replicating log partitions across a configurable number of nodes in the Kafka cluster.<sup>19</sup>
- (f) **Security:** Possibility to SSL encrypt messages between brokers and clients, between brokers or between brokers and tools. Note: performance trade-off with SSL enabled.<sup>20</sup>
- (g) **Availability:** Multiple Kafka brokers or clusters may be deployed to different servers to ensure availability, for example during maintenance.
- (h) **Reliability:** Multiple Kafka brokers or clusters may be deployed to decrease mean time between failure. If a broker or a cluster fails, load can be transferred to another until it is repaired.

<sup>14</sup> <https://kafka.apache.org> (accessed on: Thursday 30<sup>th</sup> April, 2020)

<sup>15</sup> <https://kafka.apache.org> (accessed on: Thursday 30<sup>th</sup> April, 2020)

<sup>16</sup> <https://kafka.apache.org/intro> (accessed on: Thursday 30<sup>th</sup> April, 2020)

<sup>17</sup> <https://kafka.apache.org/uses> (accessed on: Thursday 30<sup>th</sup> April, 2020)

<sup>18</sup> <https://kafka.apache.org/uses> (accessed on: Thursday 30<sup>th</sup> April, 2020)

<sup>19</sup> <https://kafka.apache.org/intro> (accessed on: Thursday 30<sup>th</sup> April, 2020)

<sup>20</sup> <https://kafka.apache.org/documentation/#security> (accessed on: Thursday 30<sup>th</sup> April, 2020)

### 3.3.3.2 Docker

Docker, or more specifically Docker Engine, is a container runtime that allows bundling all application dependencies inside containers. On Docker Engine, containerized applications may be run on any machine consistently independent of the underlying infrastructure. Docker Engine is built upon containerd as its container runtime.<sup>21</sup>

The following non-functional requirements are met by docker:

- (a) **Scalability:** More containers of the same component may be started on demand to scale. Additionally, Docker swarm may be used to build a cluster of docker engine that can be scaled easily.<sup>22</sup>
- (b) **Modifiability:** Since components run inside docker containers, they may be modified or replaced without impacting other components.
- (c) **Extensibility:** More components can be added in the future in the shape of new docker containers.<sup>23</sup>
- (d) **Robustness:** containerd, docker's container runtime, emphasizes particularly on robustness.<sup>24</sup>
- (e) **Availability:** Multiple containers of the same component may be deployed simultaneously with a load balancer. During maintenance of one container, requests will be redirected to the other.
- (f) **Reliability:** Multiple containers of the same component may be deployed simultaneously with a load balancer. If one container fails, requests will be redirected to the other.

## 3.4 Implementation

In this section, the implementation of the system is described. First an overview of the infrastructure is presented, then the implementation of the various components is shown. The next part further elaborates on the interaction of the components within the system. Finally, the user applications and their user interfaces are presented in a functional description.

### 3.4.1 System Overview

Applying the gathered knowledge from the design phase in Section 3.2 and the evaluation of related technologies in Section 3.3, this overview seeks to present a coarse grained overview of the system and the technologies used. Figure 3.2 visualizes the system infrastructure with a UML component diagram. The diagram is built upon the more high level design diagram from Section ?? and specifies more implementational details.

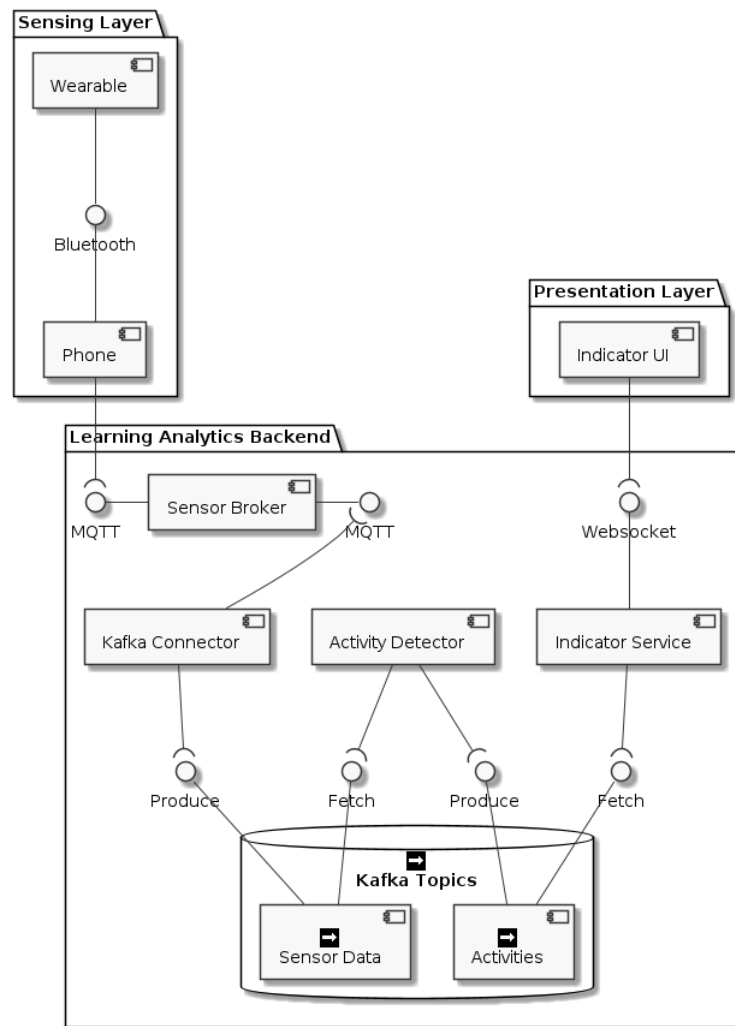
The sensing layer consists of all necessary sensor components, namely the applications installed on the wearable device and the phone, depicted as the components *Wearable Sensor*

<sup>21</sup> <https://docker.com/products/container-runtime> (accessed on: Thursday 30<sup>th</sup> April, 2020)

<sup>22</sup> <https://docs.docker.com/engine/swarm> (accessed on: Thursday 30<sup>th</sup> April, 2020)

<sup>23</sup> <https://containerd.io/scope> (accessed on: Thursday 30<sup>th</sup> April, 2020)

<sup>24</sup> <https://containerd.io> (accessed on: Thursday 30<sup>th</sup> April, 2020)



**Figure 3.2:** Component diagram of the system design.

and *Phone*. Communication between *Wearable* device and *Phone* is conducted via Bluetooth. This is due to the fact that Wear OS by Google can use either Wi-Fi or Bluetooth for data transmission, and Wi-Fi transmission does not provide the desired performance.

The sensing layer communicates with the learning analytics backend through the *Sensor Broker*, an MQTT broker, since MQTT was chosen as the protocol of choice. The *Sensor Broker* component provides an MQTT web interface to which the phone application connects and sends preprocessed sensor data. The *Phone* application utilizes the open source Eclipse Paho MQTT Java library for MQTT connections.

For the messaging system, the technology of choice is Kafka. Kafka topics are depicted as components to clarify responsibilities of the various applications. In the actual implementation, the Kafka broker alone manages communication between applications and topics by exposing *produce* and *fetch* API. In this diagram, to focus on interaction, the topics are components that each expose a fetch and a produce API.

The *Kafka Connector* is a Kafka source connector that implements the *Kafka Connect* API and subscribes to the MQTT broker to produce sensor data to Kafka topics, connecting the broker to the messaging system. All further processing is conducted by the activity detector, a Kafka stream processor implemented with the Kafka Streams binder for Spring Cloud Stream in Java.

The *Activity Detector* fetches raw sensor data from the sensor data topic and produces results to the activities topic. To present the results, the responsibility of the *Indicator Service* is to translate the results in a format suitable for visualization by the presentation layer. The *Indicator Service* consumes from the *Activities* topic to translate data in a suitable format for the *Indicator UI* and exposes it through a WebSocket endpoint. Therefore, the *Indicator UI* can subscribe to the results without a dependency to the Kafka protocol. This allows for easier replacement of front or backend.

The backend is secured by username and password authentication on both the indicator service and sensor broker. All external transmissions are TLS encrypted to ensure transmission security.

### 3.4.2 Technical Component Description

The technical description of the system's components presents the inner structure of the components without diving into the source code. Sequence and activity diagrams are used for most components to achieve a coarse-grained overview of the technical implementation. Furthermore, specific libraries and frameworks used are mentioned.

#### 3.4.2.1 Wearable Companion

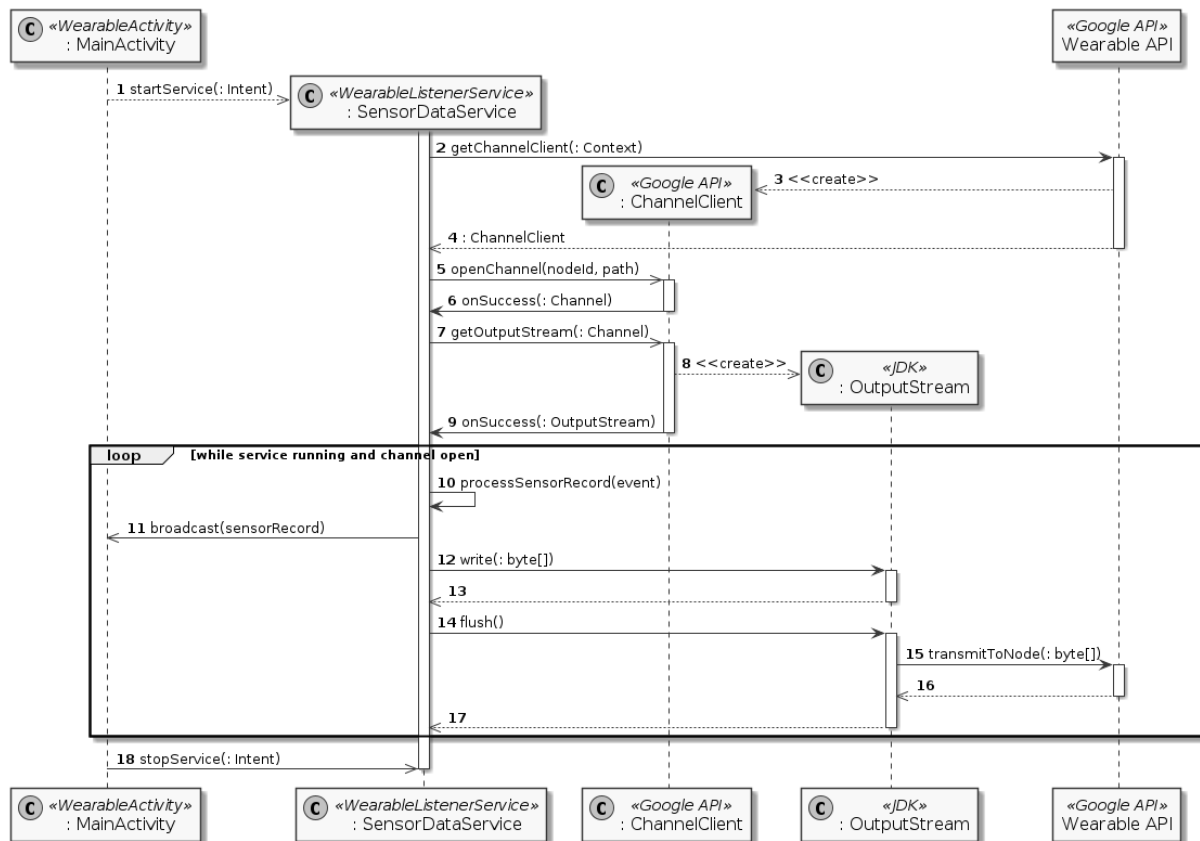
The wearable companion is installed on a wearable device of each student. Its responsibilities are to start and stop learning sessions, transfer sensor data to the phone companion and prompt the student with questions about their constitution and the learning context. After the start of the learning session is triggered, the transmission of sensor data to the phone companion is started.

We implemented the wearable companion as an Android App using WearOS. It was tested and optimized for the Mobvoi Ticwatch Pro. The connection to the phone companion was conducted using WearOS and Google Play Services. A *WearableListenerService* was implemented to run in the background and establish a connection to the phone companion as soon as it detected.

As soon as the student starts a session, the application initiates the following actions: An instance of *SensorDataService* registers a *SensorEventListener* for each desired Sensor. Registering is initiated only if started explicitly. *CapabilityClient* is then used to acquire nearby wearable nodes, and a suitable node is picked. A new *ChannelClient.Channel* for the current node ID is opened through *ChannelClient* and passed to *SensorDataService.AccelerometerListener*. Finally, a *HandlerThread* is created and *AccelerometerListener* is registered for each sensor in the created thread.<sup>25</sup> The listener uses the *OutputStream* of the channel to transmit data to the

<sup>25</sup> The separate thread prevents the UI performance from being impacted by the listeners.

phone. Figure 3.3 depicts the interactions of the various classes in a sequence diagram.



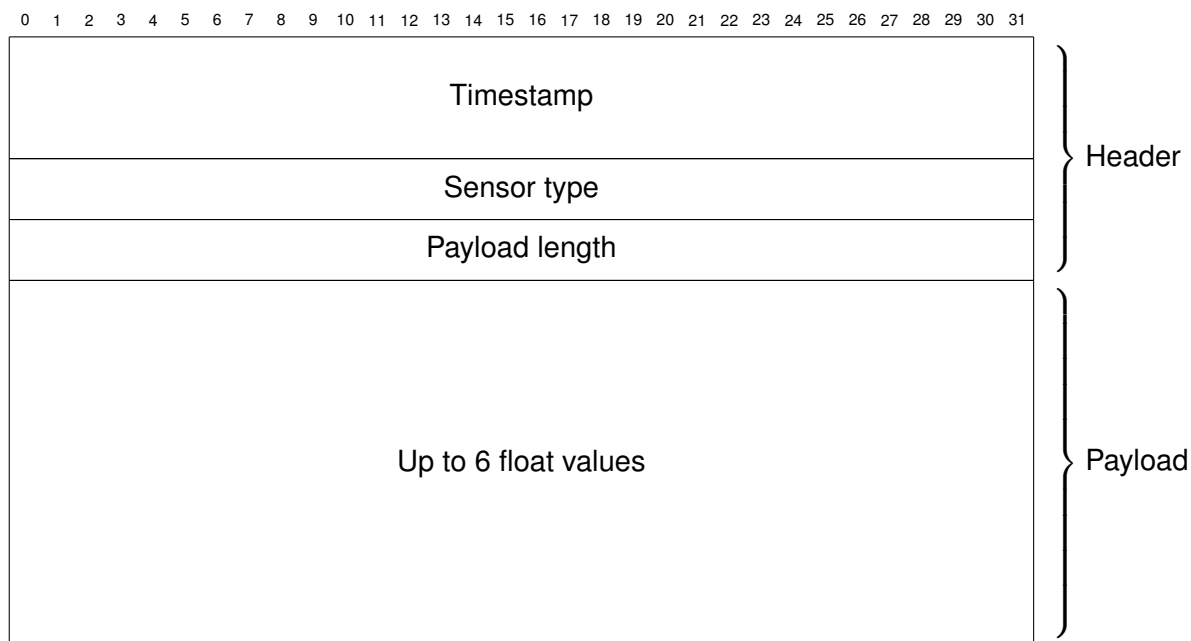
**Figure 3.3:** Sensor record transmission interactions within the phone application.

If the service is not started explicitly, it will still listen to changes in capabilities to keep the node ID up to date with the most suitable node, but no records will be recorded. If capabilities change whilst the service is already running, sensor listeners will be re-registered to publish to a newly opened channel for the updated node ID. This feature allows for automatic resume of recording if nodes are momentarily disconnected and then reconnected. As long as running is set to true in *SharedPreferences*, changes in capability will always re-register the listeners and start recording.

Sensor data is transmitted as a stream of bytes through the provided *OutputStream* of the channel. To be able to deserialize the records on the receiving end, a simple protocol is used. The protocol uses the following packet format:

- (a) Timestamp (8 bytes)
- (b) Sensor type (4 bytes)
- (c) Length of the record array (4 bytes)
- (d) Record payload (n bytes)

The parts are all serialized to byte arrays and merged into one byte array in succession. The start of the byte array is always 16 bytes long, whereas the size of the actual records may vary



**Figure 3.4:** Byte fields of the wearable transmission protocol. Each row represents a word with a word length of 32 bits.

depending on the sensor type. The protocol assumes a maximum length of 40 bytes, which leaves 24 bytes for sensor data per message. 24 bytes equates to 6 float values.<sup>26</sup> Figure 3.4 shows a graphical representation of one packet's byte fields of the protocol. The six-lines-long payload represents the maximum payload, which may be partly empty depending on the record type.

The UI of the companion app is updated through the *SharedPreferences* API. It listens to preference change events, which may be emitted by the UI itself on user input or by the background services if the connection or transmission status changes.

### 3.4.2.2 Phone Companion

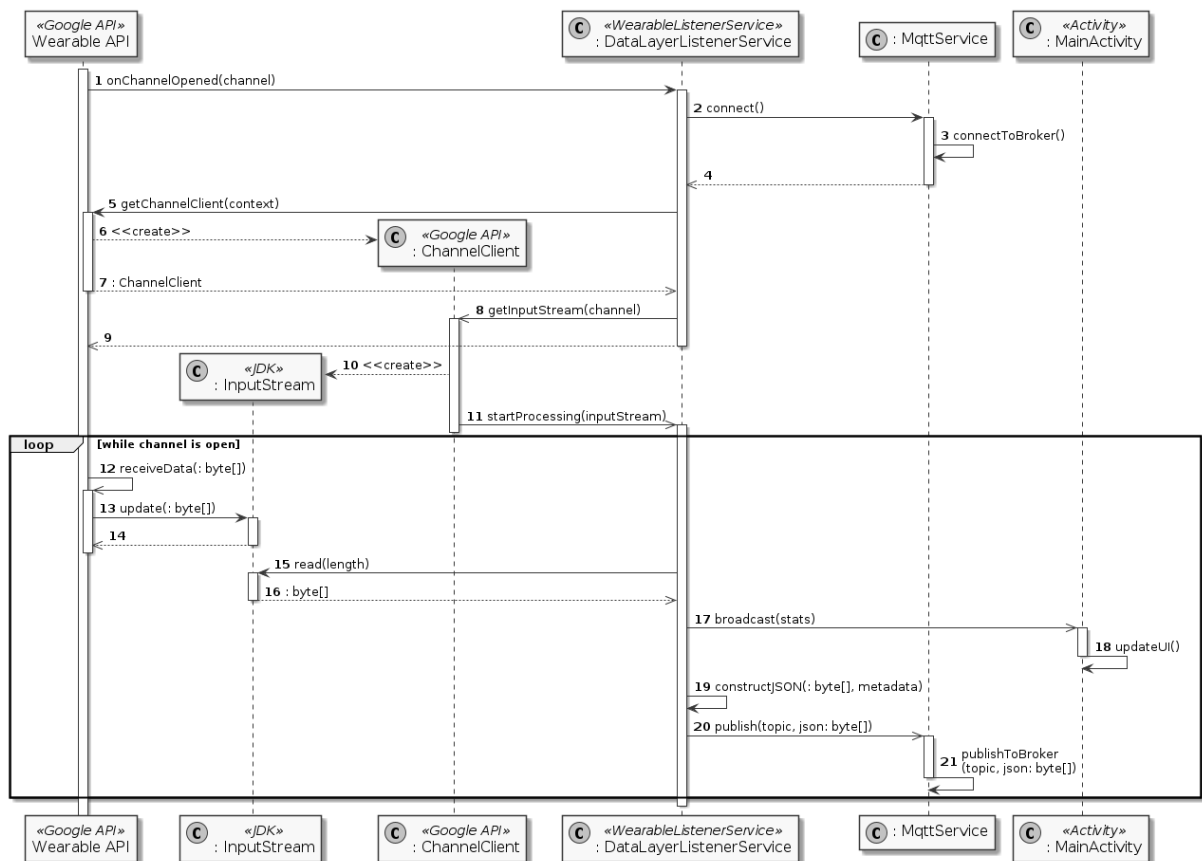
Just as the wearable companion, we implemented the phone companion using Android and WearOS. It consists of a login screen, a settings page and an implementation of a *WearableListenerService*. The application runs as a daemon in the background if the wearable companion opens a channel through WearOS, although an initial login to the MQTT broker is necessary. If the client is not yet authenticated, the *LoginActivity* is launched. If host and port of the broker are not yet configured, the *LoginActivity* launches the *SettingsActivity* so that the student may configure the connection to the broker prior to logging in.

An instance of *DataLayerListenerService* is started in the background whenever *ChannelClient* opens up a channel to the node on which the service is installed. It is likewise stopped when the channel is closed. The channel is either closed by the publishing side or if an error

<sup>26</sup> 6 float values is the maximum length of a sensor record according to the Android documentation ([https://developer.android.com/guide/topics/sensors/sensors\\_motion](https://developer.android.com/guide/topics/sensors/sensors_motion) (accessed on: Thursday 30<sup>th</sup> April, 2020))

occurs during data processing in the service.

Once a channel is opened by the wearable companion, the *InputStream* provided by *ChannelClient* is being processed in a loop. The processing stops once an *IOException* is caught, which indicates that either an error has occurred or the channel has been closed. Since processing errors at this point are non-recoverable, the exception is merely logged and otherwise handled the same as a successful closure of the channel. After a message was successfully extracted, it is serialized into a JSON string following the Avro format expected by the Kafka processors and sent to the MQTT broker. Figure 3.5 visualizes these interactions in a sequence diagram.



**Figure 3.5:** Sensor record transmission interactions within the phone application.

Account and credential management are outsourced to Android's *AccountManager*. Redirects regarding account management are not conducted manually but rather through *AccountManager*. A class named *AccountConnector* was defined with the responsibility to redirect to the appropriate actions of *AccountManager*. If no accounts are found for the appropriate account type, the user is redirected through *AccountManager.addAccount()*. The corresponding logic to add an account is implemented in *AccountAuthenticator.addAccount()*. If login credentials are invalid, the process is continued by *AccountAuthenticator.updateCredentials()*. If necessary settings to connect are not found in *ShardPreferences*, the user is redirected to the *SettingsActivity* manually. Figure 3.6 depicts a sequence diagram, visualizing the interactions concerning account and credential management.



The byte array of sensor data in the aforementioned format is read in chunks of 40 bytes. 40 bytes is the maximum size of one message from the wearable companion. The maximum size is computed as follows: The maximum length of a float array representing one sensor record is 6,<sup>27</sup> where each float uses 4 bytes. The metadata header of the message — as mentioned in Section 3.4.2.1 — has a size of 16 bytes. This amounts to a maximum message size of 40 bytes.

The implementation of the stream processor leads to a varying number of unused capacity depending on the length of the sensor record's float array. For instance, one record of a one dimensional sensor uses only 20 bytes, which is half of the maximum size. The compound size of multiple messages is, thus, twice as large as the feasible minimum. This approach, on the other hand, simplified message processing greatly, as no complex format was necessary to separate the messages from the byte stream. Reading chunks of 40 bytes at a time seemed to be a reasonable trade-off between performance and simplicity.

We used the Eclipse Paho MQTT client library to implement publication of sensor data to the MQTT broker. Initially, the regular *MqttClient* implementation was used, but later tests showed that the *MqttAsyncClient* yielded better performance results, which is why the async client was used. A possible reason for the better performance could be the fact that the non-blocking nature of the async client avoids blocking the processing loop and rather discards records if too many messages are in flight. See Section 4 for more details about the in-flight window and concerns about increased memory usage on a slow network connection.

### 3.4.2.3 Mosquitto MQTT Broker

The Mosquitto MQTT broker was deployed using the official docker image of the Eclipse Foundation. TLS support was enabled by specifying the options *cafile*, *certfile* and *keyfile* in the configuration file *mosquitto.conf*. These options refer to file paths for central authority (CA) certificate, server certificate and private key for the server certificate.

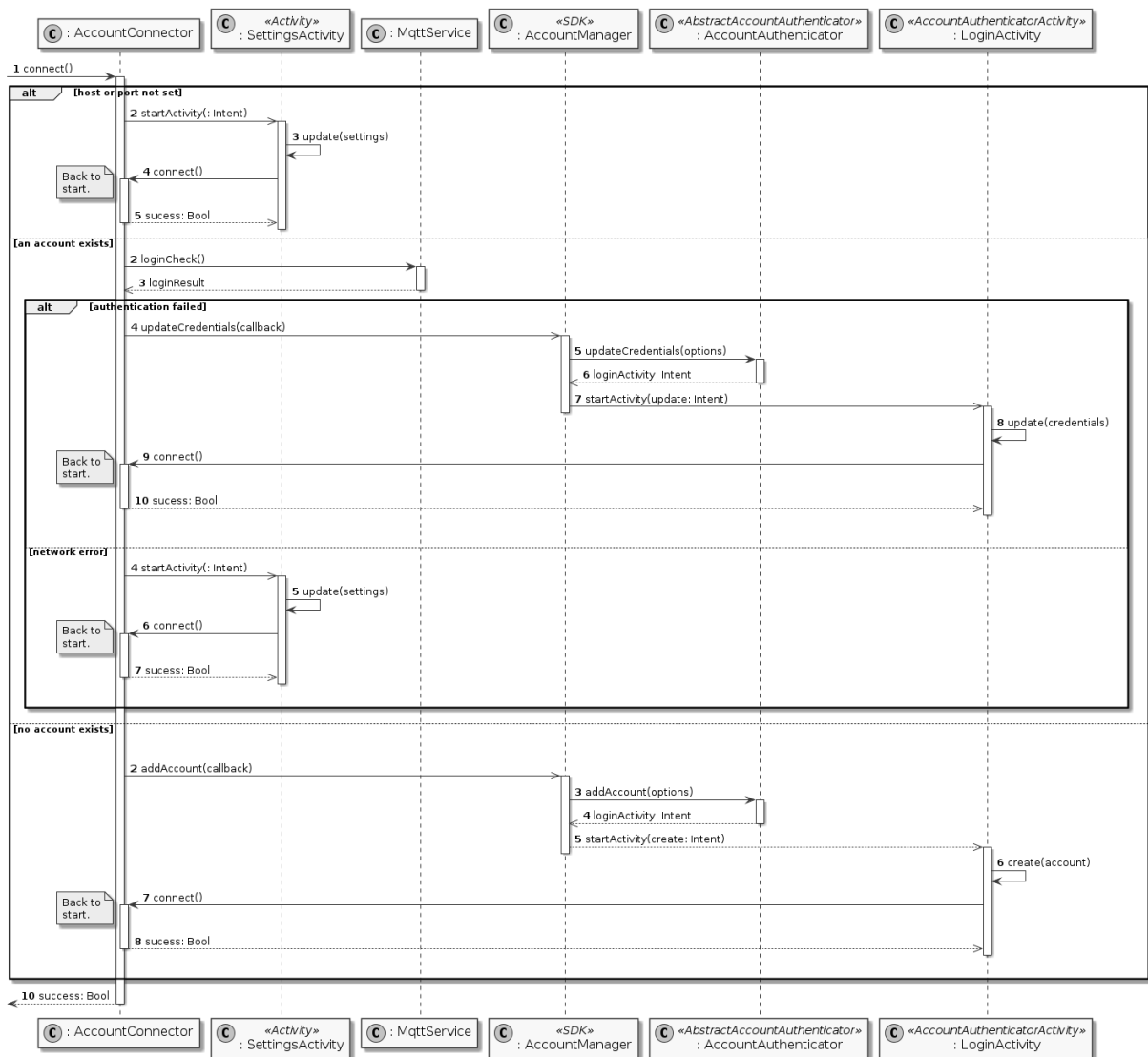
For authentication, a password file containing the list of users and corresponding passwords was provided with the option *password\_file* and the option *allow\_anonymous* was set to *false* to only allow authenticated users to access the broker. The password file currently only contains one user that is used for all testing purposes. Authorization may be managed with an *acl\_file* containing an access control list. It supports defining access rights to topics by username or client id. Currently, no access rights are defined, as the test user is allowed to access all topics.

More versatile authentication and authorization approaches are possible using Mosquitto's *auth\_plugin* option. This feature allows external “auth plugins” to be installed to cover more complex use cases, such as maintaining a centralized user database for the MQTT broker and additional services or integrating the authorization and authentication mechanism of the Mosquitto broker with the learning organization's centralized identity management. For instance, the plugin *mosquitto-auth-plugin* provides interfaces for Structured Query Language (SQL) databases, JSON Web Tokens (JWT), Lightweight Directory Access Protocol (LDAP) or a custom HTTP API.<sup>28</sup> A custom HTTP API in conjunction with JWT support could be used to integrate MQTT

<sup>27</sup> <https://developer.android.com/reference/android/hardware/SensorEvent#values> (accessed on: Thursday 30<sup>th</sup> April, 2020)

<sup>28</sup> <https://github.com/jpmens/mosquitto-auth-plugin> (accessed on: Thursday 30<sup>th</sup> April, 2020)





**Figure 3.6:** MQTT account and credentials management within the phone application.

authentication and authorization with an OAUTH2 infrastructure.

The topic structure follows the following convention: `/sensors/<SENSOR_NAME>/<USER_ID>`. A future implementation of an authorization endpoint could check the final topic sub path against the actual user id to manage access to a topic by a user. The user id may be stored inside the JWT.

### 3.4.2.4 Kafka MQTT Connector

There is no free Kafka MQTT connector provided by Confluent, only a proprietary, closed-source solution.<sup>29</sup> The proprietary Kafka MQTT connector by Confluent could be used for 30

<sup>29</sup> <https://confluent.io/hub/confluentinc/kafka-connect-mqtt> (accessed on: Thursday 30<sup>th</sup> April, 2020)

days for free, but a sustainable solution without any commercial dependencies was desired. As an alternative, one of the various community-provided, open-source connectors was chosen; a project named *evokly/kafka-connect-mqtt*, published on github.<sup>30</sup> The project is published under the MIT licence.

Although the MQTT connector supported most desired features, some modifications were necessary to fit the use case. Therefore, we forked the project and implemented the following additional features to meet the requirements:

- (a) **Docker support:** We created a Dockerfile to build the Connect MQTT library platform independently. A docker container with the associated image may be started to hold the Java Archive (JAR) file in a docker volume that may be mounted with a named volume and used by a *Kafka Connect* instance.
- (b) **JSON to Avro conversion:** The library uses an interface called *MqttMessageProcessor* to process each MQTT message. The only implementation of this interface provided by the original sources was *DumbProcessor*, which generates Kafka *SourceRecords* with a *String* key and a *byte[]* value without any additional processing logic. Conversely, Avro was chosen as the data format for the Kafka pipeline, that is why we implemented an extension to map MQTT message payloads to Avro records. Avro support may now be enabled through the configuration file. If enabled, the *SourceTask* of the Connect library retrieves the schema from the schema registry under the specified address. If the schema in the registry changes, a restart of the Connect worker is required.
- (c) **TLS without client certificates:** Originally, the library did not support using self-signed CA certificates without specifying client certificate and key, which did not fit the development requirements. At the early stage of development, no CA certificate signed by an official CA was used on the server, which is why we modified the library implementation to use the configured self-signed certificates even if client certificate or key are not configured.
- (d) **Multiple Kafka topics per instance:** While the original implementation allowed configuring one Kafka topic which all MQTT messages are produced to, it was desired to separate the various sensors in different Kafka topics. This reduces complexity on the consumer side and facilitates adding and removing separate analytical processors that consume from a dedicated topic at will. We extended the behaviour of the library so that the configuration of a single Kafka topic may be omitted for a connector, in which case the destination topic of the connector is derived from the MQTT topic name at runtime. This allows for adding additional sensor at any time without the requirement of changing the configuration. Connector instances may be configured to listen to a specified MQTT topic pattern and the connector picks the target topic independently.

An MQTT connector is configured using a JSON configuration file. Multiple connectors may be configured by creating multiple configuration files. A configuration file must adhere to the format expected by the Kafka Connect representational state transfer (REST) API, as connectors are managed through an HTTP request.<sup>31</sup> Configurations are applied by a docker container running a custom docker image that was developed just for that purpose.

<sup>30</sup> <https://github.com/evokly/kafka-connect-mqtt> (accessed on: Thursday 30<sup>th</sup> April, 2020)

<sup>31</sup> <https://docs.confluent.io/platform/current/connect/references/restapi.html> (accessed on: Thursday 30<sup>th</sup> April, 2020)

The docker image *connector-creator* was developed to create and recreate connectors using the provided configuration file. This approach was preferred to using Confluent's *Control Center*, a web-based user interface to manage the entire Kafka infrastructure. Control Center was not used for two reasons:

- (i) *Control Center* is licensed under the commercial *Confluent Enterprise Licence*, meaning it is not free to use but a paid licence.
- (ii) Although a graphical user interface may be beneficial to view and change configurations quickly without much technical expertise, a text based configuration approach increases reproducibility, may be managed by the version control system and simplifies configuration automation.

The *connector-creator* image consists of one *entrypoint.sh* shell script that runs once on startup and terminates with the container after running. It uses a combination of configured environment variables and the aforementioned connector configuration files to create — or recreate if already running — desired connectors by accessing the Kafka Connect REST API with *curl*. If the connector is configured with avro and schema registry support, the script will look for appropriate *.avsc* files in the *auto-schemas* subdirectory and post the schemas to the schema registry for later use by the Kafka connectors.

With each *docker-compose up* command, the management script is executed on container startup. This behaviour may not be desirable for production use, in which case the image may be run with a standalone *docker run* command outside the Compose infrastructure and the service may be removed from the *docker-compose.yml* file. In a development environment, however, the automatic recreation simplifies the development process greatly.

#### 3.4.2.5 Activity Detector

We implemented the *Activity Detector* component using *Spring Boot* with *Spring Cloud Stream* and the *Spring Cloud Stream Kafka Streams* binder. Gradle was used as the build tool.

The application consists of numerous *StreamListeners* which process the sensor data streams that are produced by the Kafka MQTT connector. While it is intended to detect fine-grained hand activities from the accelerometer data with a convolutional neural network (CNN) in the future — similar to the approach of [LH19] — it is currently merely a proof of concept that performs arbitrary analytical processing on the streams, such as calculating records per second, total record count and time elapsed for the learning session.

Since the *Indicator Service* component merely exposes the sensor records through a Web-Socket endpoint as-is, the *Activity Detector* component proxies the records to the output bindings without further processing. In addition, the mentioned analytics and a set of active user IDs are computed by processing a dedicated stream holding data of all sensor types and then produced to separate topics.

A set of IDs from active users is maintained in a Kafka topic to simplify querying user data. The stream listener is triggered whenever any sensor record is published to Kafka and extracts a set of unique user IDs. The set is created by grouping the stream of sensor records by a

constant key, which puts all records in a single grouping. The grouping is aggregated to a resulting set of user IDs. A new set of user IDs is only published if it differs from the previous set. Figure 3.7 details the executed stream processing operations to maintain a list of user IDs in a Kafka topic.

Avro key and value serdes are configured for all bindings, so the data of the streams this application listens to must be in Avro format. To access the latest data models as Java classes in the source code, the plugin *com.github.imflog.kafka-schema-registry-gradle-plugin* was used. This keeps the source code free of any data model related code, which could be subject to future changes to the schema. Only the *StreamListeners* must be implemented and the interfaces for the bindings defined.

#### 3.4.2.6 Indicator Service

Just as the *Activity Detector* component, the *Indicator Service* component is a *Spring Boot* application using the *Spring Cloud Stream Kafka Streams* binder that we developed using *Gradle* as a build tool. It leverages the WebSocket support of the *Spring Boot* starter dependency *spring-boot-starter-websocket*.

The WebSocket endpoint is configured by implementing a *WebSocketMessageBrokerConfigurer*. The Spring bean *SimpMessagingTemplate* can then be used elsewhere in the application to publish messages to various WebSocket topics through the endpoint.

For each WebSocket topic that shall be published to, a *StreamListener* was implemented to retrieve data from the various Kafka streams, map it to data transfer objects (DTOs) that are expected by *Indicator UI* and publish to the corresponding WebSocket topics.

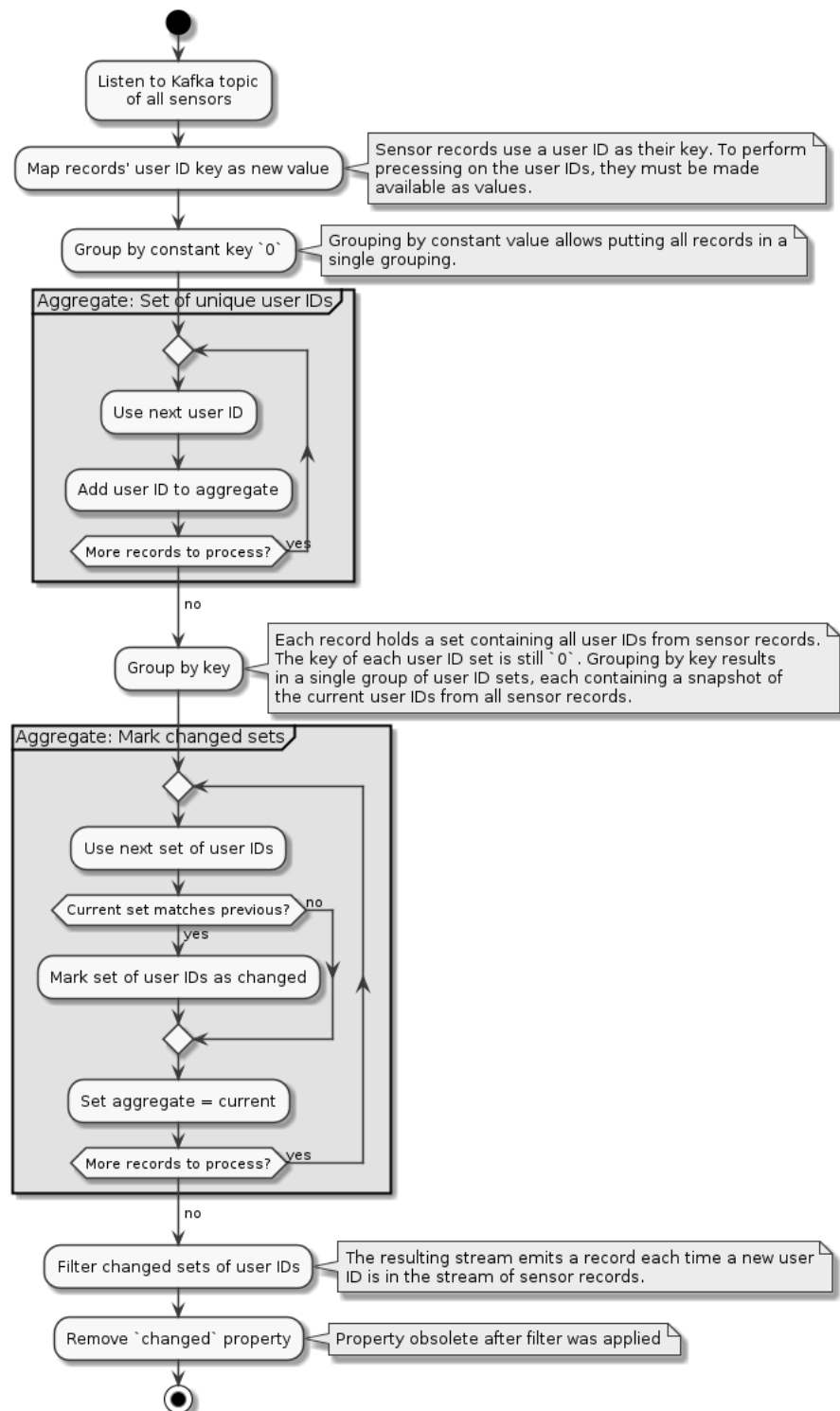
The service does not perform any analytical processing on the data but rather serves as an interface between the Kafka topics and the *Indicator UI* frontend. Data analytics may be performed by other stream processing applications, such as *Activity Detector*. *Indicator Service* collects resulting knowledge data that is of interest for *Indicator UI* from the Kafka topics and exposes it through a more common interface for web applications, the WebSocket endpoint.

#### 3.4.2.7 Indicator UI

The *Indicator UI* component is a web-based GUI application which we implemented in plain Javascript. It uses the libraries *SockJS*, *StompJS* and *ChartJS* to subscribe to the WebSocket endpoint of *Indicator Service* and visualizes real-time data streams in various graphs.

Multiple WebSocket topics with sensor data may be visualized in a chart by specifying topic name and parameter length in an array that holds configured charts. This suffices for the application to render a graph for the specified sensor topic. In addition, the application subscribes to a topic holding the active user IDs to provide the user with a dropdown to choose which user's sensor data shall be visualized.

When the page loads, the client first establishes a connection to the STOMP endpoint exposed by *Indicator Service* over a WebSocket connection. It then subscribes to the user IDs topic and adds a callback that updates a dropdown with user IDs on each new user ID received. When a user ID is selected from the dropdown, the client subscribes to the performance metrics

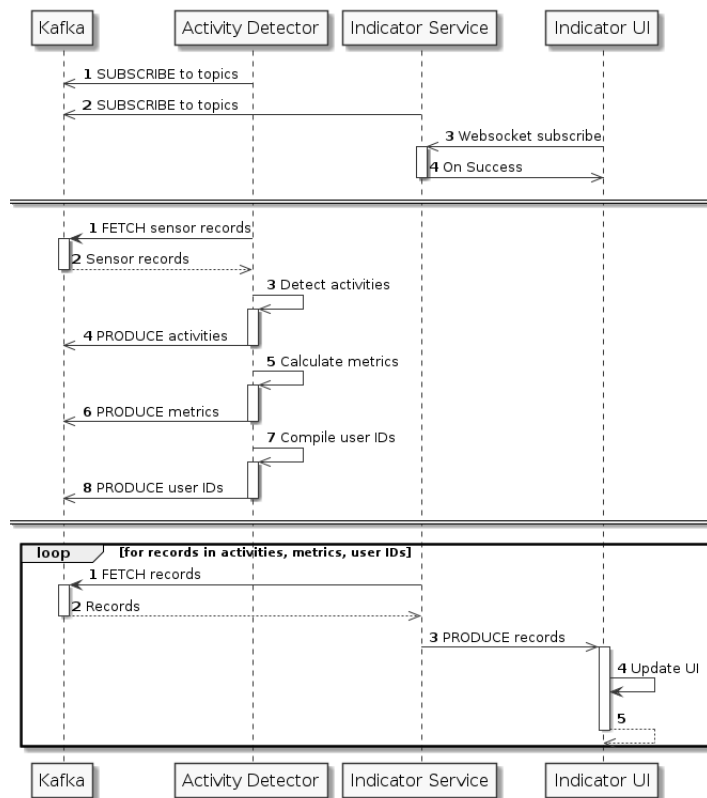


**Figure 3.7:** Kafka stream listener logic in the activity detector to maintain a list of user IDs.

and sensor data topics for the selected user ID.

After subscribing to the relevant STOMP topics for the user ID, the UI reacts to every data event from the STOMP subscriptions by updating the respective ChartJS chart data with the new sensor data.

Figure 3.8 shows a sequence diagram detailing the interactions between Indicator Service, Indicator UI and the Kafka data pipeline.



**Figure 3.8:** Interactions between Indicator Service, UI and the Kafka pipeline.

### 3.4.3 Component Interaction

During a sensor recording session, multiple components interact with one another to act as a data pipeline powered by Kafka and MQTT brokers. The MQTT broker is the entry point to the pipeline; it exposes the MQTT endpoint for clients to connect. The Kafka MQTT Connector migrates sensor data from MQTT topics to Kafka topics. When new data arrives in the Kafka pipeline, various Kafka stream processor applications may perform analytical processing on the data.

On application startup, the Kafka MQTT Connector and Activity Detector subscribe to their configured MQTT and Kafka topics respectively. These subscriptions will later provide the real-time sensor data.

A recording session is initiated by the student. This is usually done through the wearable

device, but for the sake of simplicity, the wearable device and the phone are abstracted into a single phone component. The student chooses to start a session on the phone, which launches the recording session. The phone application signals to the student that the session is running as soon as it registers the session start.

First, the phone establishes a connection to the MQTT broker. After the connection was established successfully, sensor event listeners are registered. Every sensor event is processed into an MQTT message in JSON format, serialized to bytes. The message is then published to the MQTT broker and the corresponding topic named after the sensor type. For maximum performance, QoS0 is used.

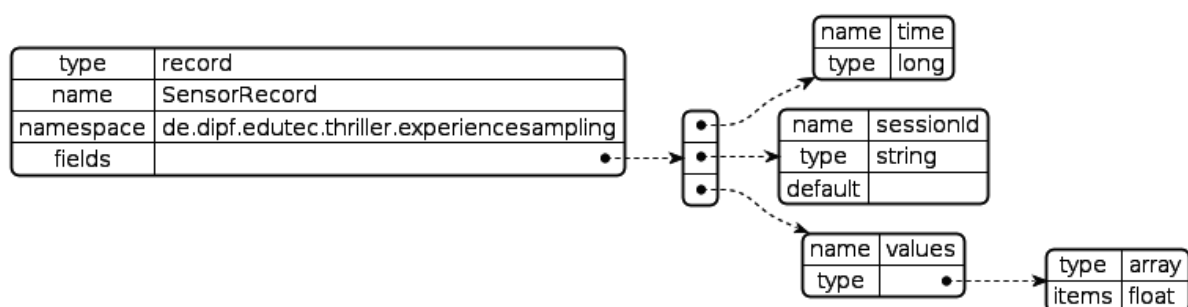
The MQTT broker then writes the new sensor data to the appropriate topics, which causes the broker to publish the data to subscribers afterwards. This means the data is published from the MQTT broker to the Kafka MQTT connector, which subscribed to the broker beforehand.

Within the connector, each new byte message is deserialized into an Avro record, which in turn is mapped to a Kafka source record. The connector is frequently polled by the Kafka broker for new records, and responds if it holds any not yet consumed records. Now the data has entered the Kafka messaging system.

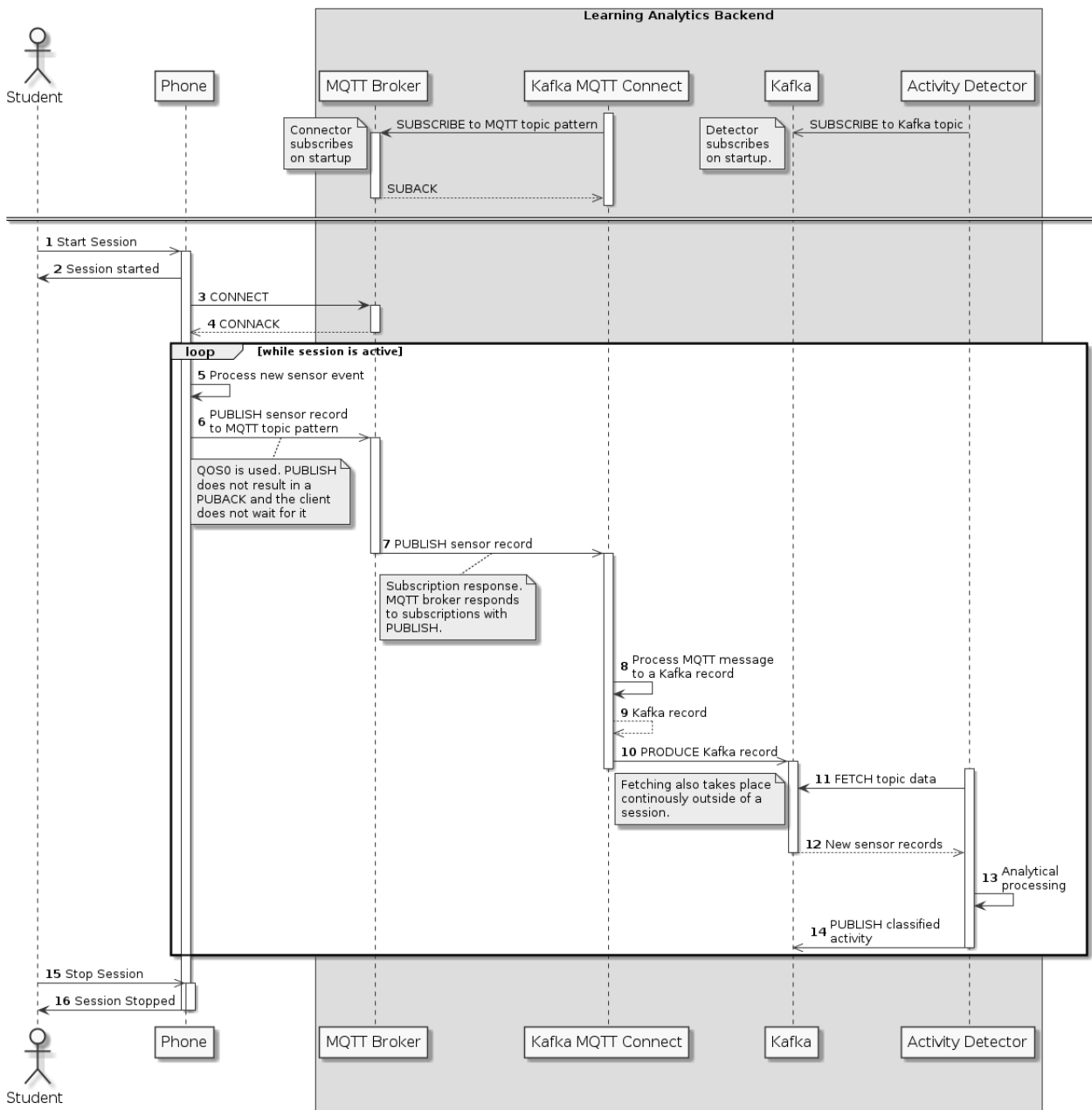
Any Kafka stream processor application that listens to sensor topics will be provided the real-time records to perform desired analytical processing. Currently, the Activity Detector listens to these topics and merely compiles performance metrics and maintains a list of active user IDs. When the session is stopped by the student, the listeners are de-registered. Figure 3.10 depicts the described interactions in a sequence diagram.

The MQTT messages must be in JSON format and contain a valid Avro record so that the connector can deserialize the record properly in the expected Avro schema. The Avro record must follow the expected Avro schema shown in Figure 3.9.

External endpoints of the systems are the MQTT broker on the sensing side and the indicator service on the presentation side. Both are protected with TLS. Whilst the indicator service endpoint is protected by a reverse HTTP proxy that manages the TLS connections, the MQTT broker, on the other hand, handles TLS itself. There is a performance and, consequently, energy overhead induced by TLS secured MQTT connections. However, the overhead is quite low and should only affect constrained devices [SZ20].



**Figure 3.9:** Avro schema for a single sensor record within the Kafka ecosystem.



**Figure 3.10:** Interaction of system components for a sensor recording session.

### 3.4.4 Functional Description of User Applications

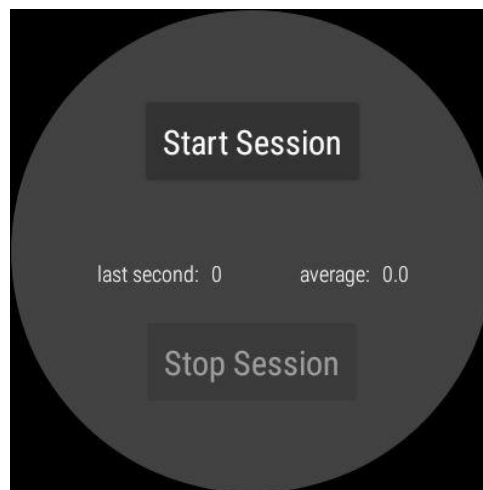
Functionalities of the system follow the functional requirements analysis of Section 3.1.1. In this section, all implemented functionalities that address a specific use case are summarized and visualized with screenshots of the resulting user interfaces.

#### (a) Trigger learning sessions

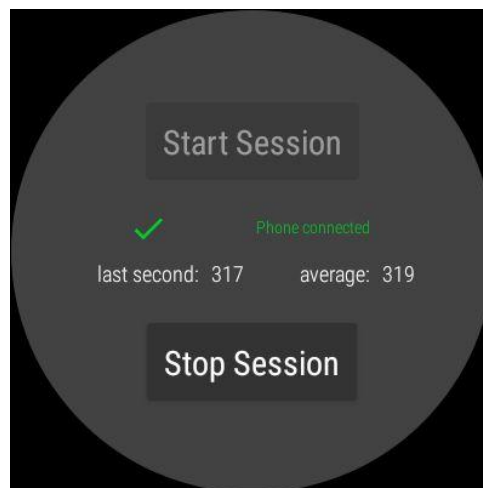
The student starts learning session through a start button in the wearable app. If the ses-



sion was started successful, the button is disabled, a label indicates that the session was started and the stop button is enabled. When the session is either stopped by pressing the stop button or by the phone device, the stop button is disabled and the start button enabled again. The indicator that the session is running disappears. If the session cannot be started, an error message is displayed. Other means to start a session, such as gestures, may be implemented to replace the button without much effort. Figure 3.11 shows a screenshot of the wearable user interface for session management in the state where no session is running. Conversely, a screenshot of the same user interface, but in the state where a session was started is shown in Figure 3.12.



**Figure 3.11:** Wearable user interface for session management when no session was started.



**Figure 3.12:** Wearable user interface for session management when the session was started.

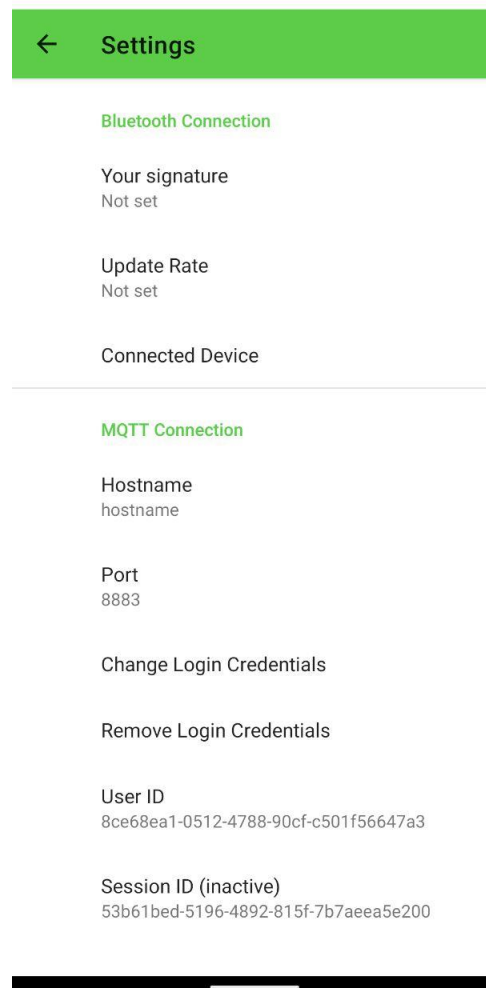
**(b) Record sensor data for engagement detection**

When the learning session is started, the wearable device starts recording data from various sensors. Currently, a set of default sensors is selected at build time of the app, but an option for the user to disable these sensors was added as well. Sensor data is

transmitted to the phone and from there forwarded to the broker in real-time. Sensor data is not aggregated but rather sent to the broker as-is, leaving it open for future development to either perform some form of aggregation by time interval on the phone or to conduct all processing on the server side.

After the sensor data is streamed into the data pipeline by the broker, a stream processor application calculates simple statistics, such as the amount of sensor records sent per second per user. This processor is intended to be used to perform more sophisticated analytical processing in the future for engagement detection.

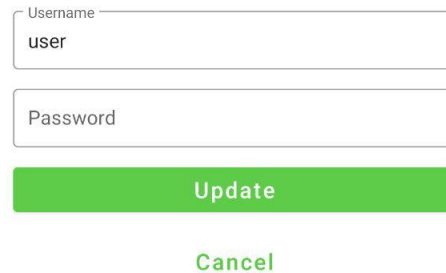
A settings screen in the phone app is used to allow the user to configure the server connection. It is opened automatically if host and port are not configured yet. Figure 3.13 depicts a screenshot of the settings dialogue. A preconfigured server address may be used for production environments for a better user experience.



**Figure 3.13:** Settings dialog of the phone application.

For authentication a login dialogue was implemented. It opens automatically if no account data was found. The same dialogue is used to modify the credentials used to log in with

the server. A screenshot of the login dialogue is shown in Figure 3.14.



The screenshot displays a login interface with two text input fields. The first field is labeled 'Username' and contains the text 'user'. The second field is labeled 'Password' and is currently empty. Below these fields is a prominent green button labeled 'Update'. Underneath the button is a green text link labeled 'Cancel'.

**Figure 3.14:** Settings dialogue of the phone application.

(c) **Presentation**

A graphical web UI that renders real-time charts displaying the sensor record stream and aforementioned statistics was implemented. Apart from the statistics, the charts show the unmodified values of each sensor event. A student is chosen by ID to show data for the particular user. Due to the absence of analytical processing, there is yet no engagement related data to be visualized. Targeted interventions were not implemented for the same reason.



## 4 Performance Evaluation

This section evaluates the performance of the system under high loads. To reliably classify fine-grained hand activity, 500 accelerometer record per second (Hz) should be processed. At 500Hz, an accuracy of over 85% may be achieved; at 4000Hz, an accuracy of 92.2%. Going below the threshold of 500Hz, accuracy drops rapidly [LH19]. In addition to the accelerometer data, additional lower frequented sensor data may be necessary to be published, such as light sensor data for context recognition. For this additional data, a buffer of 250Hz was assumed. Therefore, the system is expected to be able to process records at a rate of at least 750Hz per client.

Since each accelerometer record is transferred as a single MQTT message, 500 MQTT messages must be processed by the Mosquitto broker per second per user. Furthermore, the wearable devices as well as the network connection must be able to handle the throughput. Kafka is expected to handle the throughput without issues, as it is designed for high-throughput low-latency messaging. The custom written Kafka MQTT connector, on the other hand, was tested because it could contain custom Java code that negatively impacts performance.

Apart from meeting the expected throughput, other performance requirements shall be evaluated, such as resource usage on the wearable devices, which may impact user experience and battery usage.

For the mobile phone the question arises if messages can queue up indefinitely and use an increasing amount of memory if the network connection cannot keep up with the sensor rate. This seems, however, not to be an issue, because the Eclipse Paho MQTT client library keeps these messages as “in-flight messages” until a certain threshold is reached. After that threshold is reached, following records will be discarded until there is room in the in-flight window again.<sup>12</sup>

The maximum of in-flight messages for the Eclipse Paho MQTT client is always the maximum value of an unsigned int16, which amounts to 65 536 message ids. If `max_inflight` is higher, there is no effect. This means the maximum memory usage can be calculated as follows, where max messages in flight is  $N_m$ , memory usage is  $M$  and the size of a message is  $S_m$ :

---

<sup>1</sup> <https://eclipse.org/paho/files/javadoc/org/eclipse/paho/client/mqttv3/internal/ClientState.html> (accessed on: Thursday 30<sup>th</sup> April, 2020)

<sup>2</sup> <https://eclipse.org/paho/files/javadoc/org/eclipse/paho/client/mqttv3/MqttException.html> (accessed on: Thursday 30<sup>th</sup> April, 2020)

$$\begin{aligned}
S_m &= 100 \text{ B} \\
N_m &= 65536 \\
M &= S_m * N_m \\
&= 100 \text{ B} * 65536 = 6553600 \text{ B} \\
&= 6400 \text{ KiB} \\
&= 6.25 \text{ MiB}
\end{aligned}$$

Therefore, the maximum memory consumption of the Eclipse Paho MQTT client due to a slow network connection is negligible compared to the memory usage of the app itself. This assumes that the message size does not exceed 100 bytes. The message size of accelerometer records in the chosen format was usually ~90 bytes.

The following scenario was assumed for the Mosquitto performance test: 50 students continuously transfer data with wearable devices at 500Hz. One Mosquitto broker and one Kafka broker running in docker containers process the records. The docker containers are deployed on a virtual machine.

To test the performance of the Mosquitto broker, the benchmark tool `mqtt-bench` was used. It allows specifying a number of clients that will publish data, the number of total records to publish per client and the size of one message in bytes. The test was performed locally so that network bandwidth is out of the equation and only the broker's performance is evaluated. The test was conducted for 50 clients, 10 000 messages per second and a message size of 100 bytes.

The results suggested that the broker was able to achieve throughputs of ~38 980 messages per second and ~38 684 for publication and subscription respectively. Interpreting the slightly slower subscription rate value as the maximum throughput, this amounts to a per-client throughput of ~773 messages per second. Therefore, one Mosquitto broker is able to meet the throughput requirements for accelerometer data of at least 50 students with a buffer of ~279 messages for other data with a lower frequency, for instance the light sensor to make assumptions about the context of the learning session.

Interestingly, a performance test from related sources yielded significantly different results, likewise using the tool `mqtt-bench`. Maximum publication and subscription throughputs of ~25 000 and ~34 447.12 were observed respectively [Mis18]. Considering the fact that the mentioned performance test was conducted with public cloud brokers, other factors may have contributed to these differing results.

Table 4.1 depicts the aforementioned results; the first row shows results from this performance evaluation, whilst the second row shows results from the source.

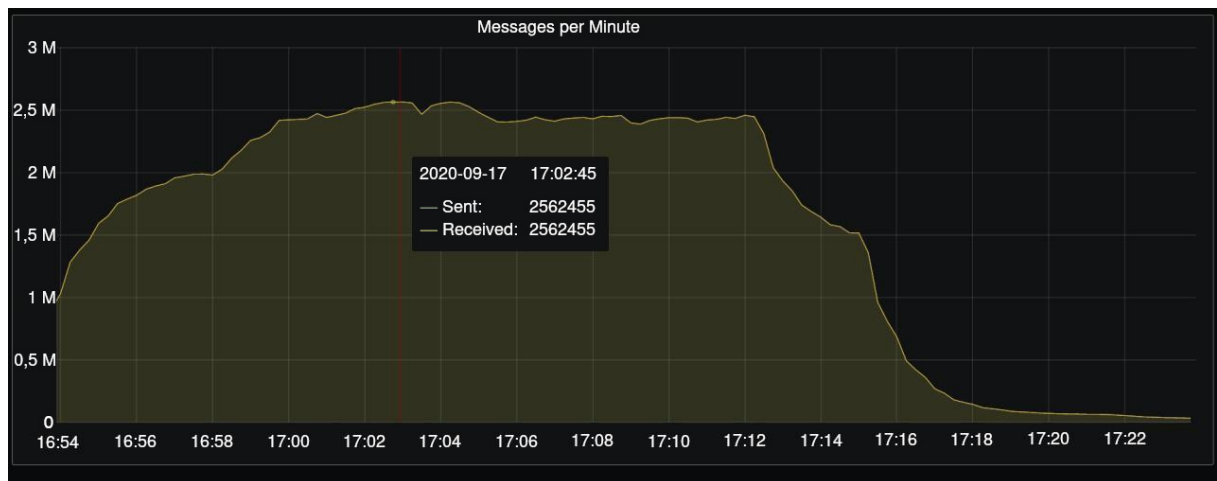
**Table 4.1:** Isolated throughput test results for only the MQTT broker.

	Rate (messages/s)	Clients	Message size (bytes)	Rate/client (messages/s)
Test result	38 684	50	100	773
[Mis18]	25 000	50	unknown	500

In addition to the test scenario above, a stress test was conducted with MQTT messages containing data in a realistic format, as opposed to the `mqtt-bench` tool, which sends generated data of the specified byte size to the broker. For the stress test, one accelerometer record was included in each message and sent to the topics which the Kafka MQTT connector was subscribed to. This would cause the connector to produce the records to an actual Kafka topic.

A custom python script was developed for the stress test, to continuously publish the same pre-defined sensor record as fast as possible to the Mosquitto broker. Multiple instances of the script were started in the background until the throughput reached its peak. The results were gathered using Prometheus exporters for MQTT and Kafka Connect and visualized in a Grafana dashboard. Using this method, network bandwidth does not impact the results, as the broker is loaded with messages to its full capacity and the result is not computed on the client side, but rather exported from the server applications themselves. The test was thus conducted on a remote server that mirrors the production environment.

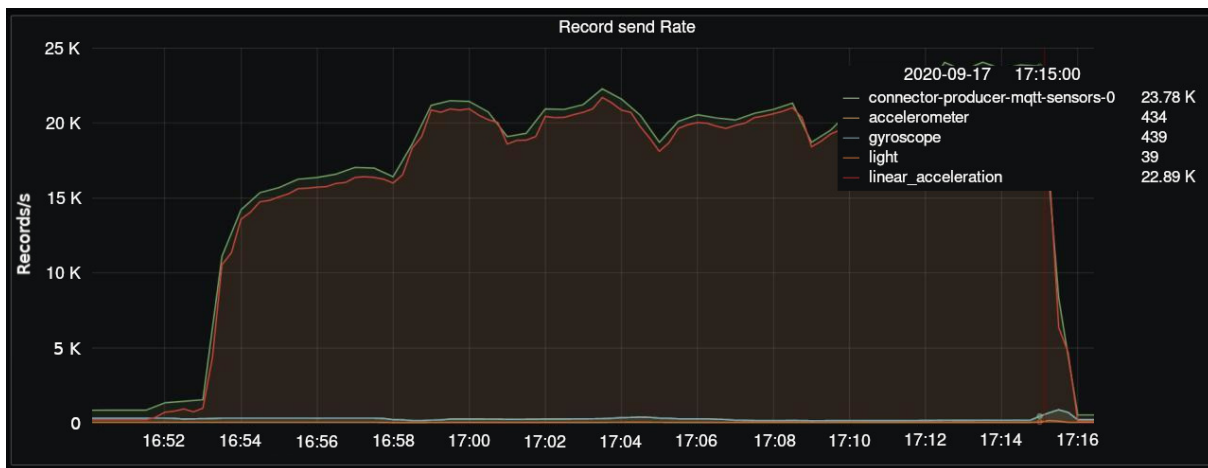
Figure 4.1 depicts the “Messages per Minute” metric, provided by the Mosquitto broker itself and visualized as a graph through the Grafana dashboard. The observed peak was 2 562 455 messages per minute, resulting in a peak of ~42 707 messages per second. Unfortunately, the Mosquitto broker does not expose any metric for messages per second, thus not allowing to see any fluctuations within that minute range.



**Figure 4.1:** “Messages per minute” metric of the Mosquitto broker.

Figure 4.2 depicts the “Record sent Rate” metric of the Kafka MQTT connector. It peaked at ~23 780 messages per second. The reason for the disparity between the performance of the Kafka connector and the Mosquitto broker is unknown, but it could be assumed that the culprit is the open-source Java implementation of the Kafka connector or the custom customizations for this project. The Mosquitto broker’s sent rate does not seem to be the cause of this issue, as it is shown to be equal to the receive rate, as seen in Figure 4.2.

Assuming the above defined throughput requirement of 750Hz per client, only ~31 instead of 50 could be served considering the Kafka connect bottleneck. It may be possible to improve the Kafka connector’s performance by altering the implementation in the future or by using multiple instances that subscribe to a fixed amount of clients each. Table 4.2 provides an overview of the throughput test results.



**Figure 4.2:** “Record sent Rate” metric of the Kafka MQTT connector.

**Table 4.2:** Throughput test results for Kafka MQTT connector.

Send rate (records/s)	Message size (bytes)	Rate/client (messages/s)	Clients
23 780	90	750	31



## 5 Conclusion

### 5.1 Key Findings

In this thesis, an infrastructure for a Learning Analytics System intended to leverage wearable computers to detect students' engagement was developed. Approaches of similar projects were discussed, the proposed design of the system presented, functional and non-functional requirements gathered, related technologies evaluated and the detailed implementation discussed.

Appropriate technologies for the development of the system were gathered and evaluated in Section 3.3, corresponding to *RQ1: What technologies and protocols are appropriate for a system capable of real-time analytical processing of sensor data?* The result of the evaluation was that MQTT is the most appropriate technology to transmit sensor data continuously over the network, as compared to CoAP, HTTP and WebSocket. Security features that are built into the protocol as well as the publish-subscribe architecture of MQTT were identified as the most notable advantages.

As performance and a low memory footprint were deemed mission critical for the sensor data transmission format, a binary format was desired. Avro was found to be the best suited option for that purpose, since it seemed to be the de-facto standard for binary data transmission with Kafka. For transmission outside of the Kafka messaging system, JSON and XML were compared with one another, where JSON was favoured mainly due to its compatibility with Avro and its general lightweight nature compared to XML.

Mosquitto MQTT was chosen as the MQTT broker because of its large user base, its open source licence and its minimalistic approach, which limited the amount of documentation to read and configuration to understand in the process of integration. A combination of Kafka and Docker was found to be an appropriate backbone to meet relevant non-functional requirements such as scalability, performance, extensibility and availability.

The infrastructure was able to meet the throughput requirements of 750Hz for ~31 clients, where the Kafka MQTT connector was identified as the bottleneck. As stated above, an improved implementation of the open source connector or running multiple instances of the connector could improve the performance to up to 50 clients, which is the maximum throughput of the MQTT broker.

Scaling the system up to a large amount of users requires a similar approach. The presented infrastructure does not include explicit scalability features, but is rather intended to be scaled by running multiple instances to handle more clients.

Authentication, authorization, transmission security and software licences were considered to answer *RQ3: What security mechanisms shall be implemented to assure data security and privacy for students?* Authentication and authorization were implemented in the system, but is not yet ready for use in a production environment. Users and roles for the Mosquitto broker are currently managed through a password file, which is the default, built-in method. To integrate authentication and authorization with the identity management of the learning organization, open-source plugins may be added to Mosquitto in the future.

Securing the data transmission from wearable to server was achieved without a noticeable performance impact using TLS. This ensures transmission encryption and the authenticity of the server. In terms of privacy, however, concerns arose due to the use of Google's proprietary Google Play Services APIs to develop the mobile applications. There was no alternative identified for transferring sensor data from wearable to mobile phone, other than abandoning Google's WearOS altogether. Although no cloud services were used to implement the data transmission from wearable to phone, the library is still proprietary and closed source, meaning a private transmission cannot be guaranteed without any doubt in mind. By avoiding APIs that sync data with the cloud, it could be argued that the threat was somewhat reduced.

Other than the privacy concerns, using Android to develop the client software showed to be a viable solution. Standardized APIs for app development helped to develop a modern user interface, and the Eclipse Paho MQTT client library could be integrated in the mobile phone application without issues. Background services were likewise simple to implement through the Android framework.

However, the use of the Android operating system proved to be problematic in terms of publishing high frequency sensor data. Research showed that the accelerometer sensor may be able to achieve a frequency of ~4000Hz, but the Android operating system seemed to not support rates higher than 100Hz without modifications of the device itself. Since the goal is to distribute the software through the Google Play Store, ready to use on students' devices, modifications to the devices was not an option. Therefore, the mobile sensing layer is not yet capable of producing accelerometer data with a minimum frequency of 500Hz that is required to detect fine-grained hand movements, even if the analytic layer is capable of processing with the required throughput.

## 5.2 Outlook

The most obvious candidate for future improvements is machine learning and multimodal learning analytics. Whilst this project aimed to build the infrastructure for a Learning Analytics System that uses machine learning to perform analytical processing, the processing itself was not implemented. The next step is hence to implement analytical Kafka processors that leverage methods such as convolutional neural networks to detect students' engagement from the sensor data [LH19].

Future work could also be done on authentication and authorization. As mentioned above, open source Mosquitto plugins could be used to implement authentication and authorization on the MQTT broker, such as *mosquitto-auth-plugin*, which provide integration points for — among others — LDAP, JWT, SQL Databases or custom HTTP APIs. The LAS backend could thus be

integrated in the teaching organization's identity management.

In terms of scalability, another layer may be added before the infrastructure to manage multiple instances of the backend. It may also be considered in the future to reduce the throughput by compressing sensor data of a certain time interval into a more compact representation, such as the resulting spectrograms of Fourier transformation for accelerometer sensor data. This would allow handling more clients with the same throughput.

To improve privacy, an open source alternative may be found to Google Play Services for WearOS. Data transmission from wearable to mobile phone could then be implemented using only open source code, therefore mitigating any risks that come from proprietary libraries. Android's Bluetooth API could be an eligible open source library for data transmission.<sup>1</sup>

It may be worth noting that Confluent Platform products licensed under the *Confluent Community Licence* are not open-source products, defining open source as being licensed under a licence that is approved by the Open Source Initiative (OSI). However, since the Confluent Community Licence falls into the category of "source-available", the usual security and privacy implications of proprietary software do not apply.<sup>2</sup> On the other hand, some products of Confluent Platform are licensed under the *Confluent Enterprise Licence* and are hence closed-source products. Therefore, when considering purchasing a subscription to Confluent Platform in the future, the implications of using closed source solutions for handling sensitive data should be considered likewise.

---

<sup>1</sup> <https://developer.android.com/reference/android/bluetooth/package-summary> (accessed on: Thursday 30<sup>th</sup> April, 2020)

<sup>2</sup> <https://atlassian.com/licensing/purchase-licensing#eccn> (accessed on: Thursday 30<sup>th</sup> April, 2020)



## 6 Source Code

The source code of the software developed as part of this thesis has been published at <https://github.com/fd-jian/master-thesis> (accessed on: Thursday 30<sup>th</sup> April, 2020).



# List of Tables

3.1	Summmary of MQTT and CoAP comparison. . . . .	13
4.1	Isolated throughput test results for only the MQTT broker. . . . .	38
4.2	Throughput test results for Kafka MQTT connector. . . . .	40





## List of Figures

3.1	Component diagram of the system design. . . . .	10
3.2	Component diagram of the system design. . . . .	19
3.3	Sensor record transmission interactions within the phone application. . . . .	21
3.4	Byte fields of the wearable transmission protocol. Each row represents a word with a word length of 32 bits. . . . .	22
3.5	Sensor record transmission interactions within the phone application. . . . .	23
3.6	MQTT account and credentials management within the phone application. . . . .	25
3.7	Kafka stream listener logic in the activity detector to maintain a list of user IDs. . . . .	29
3.8	Interactions between Indicator Service, UI and the Kafka pipeline. . . . .	30
3.9	Avro schema for a single sensor record within the Kafka ecosystem. . . . .	31
3.10	Interaction of system components for a sensor recording session. . . . .	32
3.11	Wearable user interface for session management when no session was started. . . . .	33
3.12	Wearable user interface for session management when the session was started. . . . .	33
3.13	Settings dialog of the phone application. . . . .	34
3.14	Settings dialogue of the phone application. . . . .	35
4.1	“Messages per minute” metric of the Mosquitto broker. . . . .	39
4.2	“Record sent Rate” metric of the Kafka MQTT connector. . . . .	40



# Bibliography

- [ACS17] Francisco de Arriba-Perez, Manuel Caeiro-Rodriguez, and Juan Manuel Santos-Gago. "Towards the use of commercial wrist wearables in education". In: *2017 4th Experiment@International Conference (exp.at'17)*. IEEE, June 2017. DOI: <https://doi.org/10.1109/expat.2017.7984354>.
- [Ari+15] Orlando Arias et al. "Privacy and Security in Internet of Things and Wearable Devices". In: *IEEE Transactions on Multi-Scale Computing Systems* 1.2 (Apr. 2015), pp. 99–109. DOI: <https://doi.org/10.1109/tmscs.2015.2498605>.
- [BFN10] Diane M. Bunce, Elizabeth A. Flens, and Kelly Y. Neiles. "How Long Can Students Pay Attention in Class? A Study of Student Attention Decline Using Clickers". In: *Journal of Chemical Education* 87.12 (Dec. 2010), pp. 1438–1443. DOI: <https://doi.org/10.1021/ed100409p>.
- [Bou05] A. Boulanger. "Open-source versus proprietary software: Is one more reliable and secure than the other?" In: *IBM Systems Journal* 44.2 (2005), pp. 239–248. DOI: <https://doi.org/10.1147/sj.442.0239>.
- [Car+13] Niccolo De Caro et al. "Comparison of two lightweight protocols for smartphone-based sensing". In: *2013 IEEE 20th Symposium on Communications and Vehicular Technology in the Benelux (SCVT)*. IEEE, Nov. 2013. DOI: <https://doi.org/10.1109/scvt.2013.6735994>.
- [CC14] Raphael J Cohn and Richard J Coppen. *MQTT Version 3.1.1*. OASIS. Oct. 2014. URL: <http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html>.
- [CN12] Ciobanu (Defta) Costinela-Luminița and Ciobanu (Iacob) Nicoleta-Magdalena. "E-learning Security Vulnerabilities". In: *Procedia - Social and Behavioral Sciences* 46 (2012), pp. 2297–2301. DOI: <https://doi.org/10.1016/j.sbspro.2012.05.474>.
- [Dio+16] Frédérick Dionne et al. "Using acceptance and mindfulness to reduce procrastination among university students: results from a pilot study". In: *Revista Prâksis* 1 (July 2016), pp. 8–20. DOI: <https://doi.org/10.25112/rp.v1i0.431>.
- [Gli+15] Nenad Gligoric et al. "Smart classroom system for detecting level of interest a lecture creates in a classroom". In: *Journal of Ambient Intelligence and Smart Environments* 7.2 (2015), pp. 271–284. DOI: <https://doi.org/10.3233/ais-150303>.
- [Hen+97] Robert A. Henning et al. "Frequent short rest breaks from computer work: effects on productivity and well-being at two field sites". In: *Ergonomics* 40.1 (Jan. 1997), pp. 78–91. DOI: <https://doi.org/10.1080/001401397188396>.

- [HKP02] Marit Hansen, Kristian Köhnstopp, and Andreas Pfitzmann. "The Open Source approach — opportunities and limitations with respect to security and privacy". In: *Computers & Security* 21.5 (Oct. 2002), pp. 461–471. DOI: [https://doi.org/10.1016/s0167-4048\(02\)00516-3](https://doi.org/10.1016/s0167-4048(02)00516-3).
- [Kar+15] Vasileios Karagiannis et al. "A Survey on Application Layer Protocols for the Internet of Things". In: *Transaction on IoT and Cloud Computing* 1.1 (Jan. 2015). DOI: <https://doi.org/10.5281/zenodo.51613>.
- [Kum17] Manish Kumar. *Building Data Streaming Applications with Apache Kafka: Designing and Deploying Enterprise Messaging Queues*. Birmingham, UK: Packt Publishing, 2017. ISBN: 1-78728-763-7.
- [LH19] Gierad Laput and Chris Harrison. "Sensing Fine-Grained Hand Activity with Smartwatches". In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems - CHI '19*. ACM Press, 2019. DOI: <https://doi.org/10.1145/3290605.3300568>.
- [LRS18] Anna Larmo, Antti Ratilainen, and Juha Saarinen. "Impact of CoAP and MQTT on NB-IoT System Performance". In: *Sensors* 19.1 (Dec. 2018), p. 7. DOI: <https://doi.org/10.3390/s19010007>.
- [LXH16] Gierad Laput, Robert Xiao, and Chris Harrison. "ViBand". In: *Proceedings of the 29th Annual Symposium on User Interface Software and Technology - UIST '16*. ACM Press, 2016. DOI: <https://doi.org/10.1145/2984511.2984582>.
- [Mis18] Biswajeeban Mishra. "Performance Evaluation of MQTT Broker Servers". In: *Computational Science and Its Applications – ICCSA 2018*. Springer International Publishing, 2018, pp. 599–609. DOI: [https://doi.org/10.1007/978-3-319-95171-3\\_47](https://doi.org/10.1007/978-3-319-95171-3_47).
- [Mit+17] Daniele Di Mitri et al. "Learning pulse". In: *Proceedings of the Seventh International Learning Analytics & Knowledge Conference on - LAK '17*. ACM Press, 2017. DOI: <https://doi.org/10.1145/3027385.3027447>.
- [Mit+18] Daniele Di Mitri et al. "From signals to knowledge: A conceptual model for multimodal learning analytics". In: *Journal of Computer Assisted Learning* 34.4 (July 2018), pp. 338–349. DOI: <https://doi.org/10.1111/jcal.12288>.
- [Nar17] Neha Narkhede. *Kafka: The Definitive Guide: Real-time Data and Stream Processing at Scale*. Sebastopol, CA: O'Reilly Media, 2017. ISBN: 1-4919-3616-9.
- [Nur+09] Nurzhan Nurseitov et al. "Comparison of JSON and XML Data Interchange Formats: A Case Study". In: *CAINE*. 2009, pp. 157–162.
- [Rev+17] Girish Revadigar et al. "Accelerometer and Fuzzy Vault-Based Secure Group Key Generation and Sharing Protocol for Smart Wearables". In: *IEEE Transactions on Information Forensics and Security* 12.10 (Oct. 2017), pp. 2467–2482. DOI: <https://doi.org/10.1109/tifs.2017.2708690>.
- [SB00] Kenneth T. Strongman and Christopher D. B. Burt. "Taking Breaks From Work: An Exploratory Inquiry". In: *The Journal of Psychology* 134.3 (May 2000), pp. 229–242. DOI: <https://doi.org/10.1080/00223980009600864>.
- [Sey97] Elaine Seymour. *Talking about leaving : why undergraduates leave the sciences*. Boulder, Colo: Westview Press, 1997. ISBN: 0-8133-6642-9.

- [SH19] Elaine Seymour and Anne-Barrie Hunter, eds. *Talking about Leaving Revisited*. Springer International Publishing, 2019. DOI: <https://doi.org/10.1007/978-3-030-25304-2>.
- [SHB14] Z. Shelby, K. Hartke, and C. Bormann. *The Constrained Application Protocol (CoAP)*. RFC 7252. RFC Editor, June 2014. URL: <https://www.rfc-editor.org/rfc/rfc7252.txt>.
- [Shk+14] Irina Shklovski et al. “Leakiness and creepiness in app space”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, Apr. 2014. DOI: <https://doi.org/10.1145/2556288.2557421>.
- [SZ20] Salsabeel Y. Shapsough and Imran A. Zualkernan. “A Generic IoT Architecture for Ubiquitous Context-Aware Learning”. In: *IEEE Transactions on Learning Technologies* 13.3 (July 2020), pp. 449–464. DOI: <https://doi.org/10.1109/tlt.2020.3007708>.
- [ZOJ17] Ziwei Zhu, Sebastian Ober, and Roozbeh Jafari. “Modeling and detecting student attention and interest level using wearable computers”. In: *2017 IEEE 14th International Conference on Wearable and Implantable Body Sensor Networks (BSN)*. IEEE, May 2017. DOI: <https://doi.org/10.1109/bsn.2017.7935996>.