Performance of a Digital Twin platform for tracing quality changes in fruits

Kunal Singh[1], Dr. Reiner Jedermann[1]
[1]Institute for Microsensors, Actuators, and Systems (IMSAS), University of Bremen, Germany

Abstract

The Intelligent Container for remote monitoring of refrigerated ocean food transports already included biophysical models to give better insight into the actual condition of the transported fruits. In this presentation we show, how a Digital Twin platform was developed to host such models and measure its reaction time between live sensor update and a control action triggered by the models.

1 Motivation

Sensors are often integrated into physical objects, such as construction elements, spare parts, or even boxes for fresh food products. Measured data are feed into simulation models to predict aging of the part, mechanical stress, or the expected quality loss of foods [1].
The networking of sensors and models was mostly implemented by manual wiring of the components by proprietary interfaces. In the recent years, more and more objects have been represented by digital twins (**DTs**) on cloud and server platforms [2]. This trend drives the need to provide generic solutions to combine sensors and models from different software frameworks in a flexible way.
Models for DTs are no longer processing a set of recorded data. Instead, they update their prediction on each new sensor reading. This requirement can be best met by a so-called event-driven architecture.

1.1 Related Work

There are presently various solutions of digital twins available in the market. Depending on the requirements, the applications of digital twins differ from industry to industry. Beside the open-source Kafka-based platform that will be explained in detail later, following commercial solutions are common: Microsoft Azure Digital Twin provides a platform as a service and helps to create twin graphs based on digital models of real-world environment [5]. It uses digital twin definition language (DTDL) to define digital entities representing objects from physical world as twin models. The relationships in DTDL models are used for twins to connect into a live graph which can kept up to date by connecting the twin to an Internet of Things (IOT) hub. Amazon web service (AWS) also provides its own solution of digital twin representing digital entities as JSON file which contains state information, timestamps, and other useful data. The shadows can be updated, created, and deleted from other devices and web clients using Message Queuing Telemetry Transport (**MQTT**) reserved topics and REST APIs [6].
Eclipse Ditto is another open-source digital twin solution using the eclipse ecosystem. The data from Eclipse ditto can be also send to the streaming platform Kafka.
Ditto can be applied as middleware to digital twins and IOT devices. A thing is represented in a JSON format in Eclipse Ditto. It focuses on back-end services such as providing APIs abstracting from hardware, or ensuring authorized access, for example. The data can be collected from IOT devices using Eclipse Hono and fed to Eclipse Ditto which can be connected to other web services using APIs [7].

1.2 Definition of Digital Twins

There are presently different definitions and understandings of digital twins in the research field. Grieves, who first introduced the term digital twin, defined it as "*a set of virtual information constructs that fully describes a potential or actual physical manufactured product from the micro atomic level to the macro geometrical level […]" from which "[…] any information that could be obtained from inspecting a physical manufactured product can be obtained […]*" [8]. To simplify the definition further a digital twin is a virtual representation of a physical object from a real world. It gets continuously updated from data coming from physical entities and thus provides us with accurate information about the state of entities. The data from the digital twin can be used for early prediction of failures in industry thus avoiding calamities and loss of property.

1.3 Event Driven Architecture

The implementation of this paper is based on event-driven architectures. An event is a notable thing like change of a state of an object or an occurrence. Event driven architectures uses events which happens at regular or irregular intervals to communicate between other applications built with micro-services so the information can be transferred quickly to decision makers. In the implementation of the digital twin discussed further in the paper the data-streaming platform Apache Kafka is used, which implements the software pattern of event-driven architecture. It reacts on events coming from sensors of physical entity and passes the information to the decision maker and takes actions accordingly. Event driven architectures provides benefits such as reduced operation cost, more resilient to failures, and scalability as these events can be transferred to different applications and various actions and analysis can be done. It also increases the responsiveness as the most important information is dealt meticulously and in much less time as many operations can be done in parallel.
Different components can be programmed independently, in various programming languages, running of different workstations.

ISBN 978-3-8007-5835-7

Although it provides many benefits, there are some demerits attached to it. While handling larger and complex systems, it is difficult to keep track of the state of event and also which application is in control of a particular event hence the error handling of the events becomes increasing difficult. It becomes also difficult to control the work flow as most of events are non-deterministic, since the event-timing is not exactly known. If one event has to be processed before another, the event-timing should be determined accurately. The unit testing of an event-driven architecture system can be performed easily but the scenario testing becomes complex with larger systems as it becomes difficult to analyse which events are triggered to get a particular result and in case of unknown event been released, the diagnosis of the problem is fairly hard. Sensors publish their data to a message queue or topic. Models subscribe to certain topics. Linking between different sub-models and connection to visualization services is also handled through topics.

2 Software platform

We present our implementation of a DT software platform based on the architecture suggested by [3]:

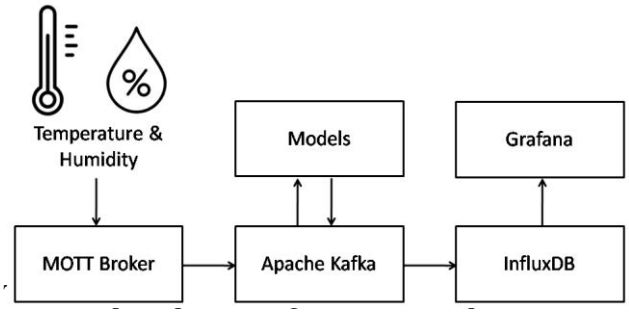

Figure 1 Overview of Digital Twin Architecture

The basic principle of a digital twin is to replicate the physical entity and provide necessary information in case of unwanted event. The architecture according to [3] provides access to the sensor and model data in multiple ways, although at the costs of redundancy by the parallel use of three different brokers and data base solutions. Data can be accessed by standard IoT protocols via MQTT. Immediate reactions to sensor changes can be triggered by events via Kafka. Finally, InfluxDB provides easy access to recorded data via tables or data base queries. Furthermore, the four different software tools increases the scalability of the digital twin solution as they can be easily integrated with many other existing applications. Many devices connected through Internet send their real time data through MQTT protocol hence linking such devices with this solution is fairly simple. Apache Kafka provides connections to many different applications and provides a throughput of 605 MB/s [11]. The storage of data is of primary importance so as to analyse it and find any discrepancies which can help us to predict the failure of a component. InfluxDB provides the real time storage and can be easily installed on any system. It also provides us with many dashboards to visualise the data. Grafana has capability to analyse and visualise data using better and efficient dashboards than InfluxDB.

The **Message Queuing Telemetry Transport** (MQTT) is used as a common standard to transmit sensor readings to the platform. It uses publish subscribe mechanism to transfer the information among clients. The real time data from the sensors are published to a MQTT topic and the data is sent to the MQTT broker which serves as a bridge between subscribers and publishers. The MQTT client subscribes to the topic and receives the data from the broker. MQTT protocol unlike other internet protocols provides Quality of Service of message, ensuring the messages are delivered to the clients without any loss of data.

The open-source Kafka is a distributed **streaming platform** that streams data using publish subscribe mechanism. Unlike centralized systems of the past where information was stored in one location making them prone to data loss in case of disruption of the system, Kafka uses distributed environment where either the information or data is broken into smaller parts and stored in different locations in different systems or making copies of entire information in different locations. Both distributed approaches have its own merits and demerits. If a system crashes some part of data is lost in the former case. In the latter case even if a system crash there is no data loss as its copies are available in other systems, but it increases the redundancy of data and requires more space. Kafka uses a publish and subscribe messaging system to communicate between applications so that applications can focus more on data and less on data transmission and sharing. Hence it less complex and easier to integrate with other applications. In Kafka the messages are produced to a single topic which can be consumed by any number of consumers. The consumers can be written in different programming languages making Kafka hugely scalable. In the implementation discussed in the paper, the messages from MQTT client are produced is a Kafka topic which is consumed by a Kafka consumer which stores the data in real time series database InfluxDB.

InfluxDB is open-source real-time series database and is used for long-term storage of time critical information. The data is stored and read in real time in InfluxDB. It attaches a timestamp to every incoming data that is ingested to the database compared to other ordinary relational databases in which a timestamp needs to be added explicitly. It provides more speed in storing and processing the real time data. InfluxDB can also be used for visualization of incoming data as it provides different dashboards for better analysing the data. Querying of data from the database can be done using Flux query.

Visualization of measured and predicted values is provided by **Grafana**, as another open-source tool. It provides us with various dashboards to analyse the data better. The data can be ingested into Grafana from database such as InfluxDB and is continuously updated with real time values.

3 Model interfaces

Example models were equipped with interfaces to comply to the publish/subscribe pattern. They are notified by Kafka about the availability of new sensor readings. Models can be written in either Java, Matlab or Python. Due to limitations of the library for Kafka-Matlab integration [4], a new Java tool was programmed to bridge between Kafka and Matlab.

3.1 MQTT MATLAB Interface

Models written in Matlab can be integrated with MQTT protocol. The MQTT toolbox can be installed in Matlab and it can be connected to the MQTT broker by providing the authentication. The models can be updated with real time values by subscribing to the desired MQTT topic. The data from the Matlab model can be published to another MQTT topic and can be then send to Apache Kafka which can be further integrated with different applications. This interface works best when the source is sending data through MQTT protocol, while in other cases this approach may not be feasible as separate toolbox for different protocols may not be available in Matlab [9].

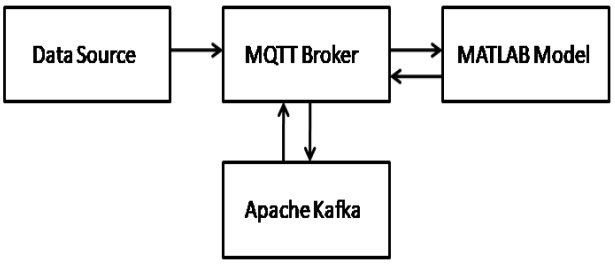

Figure 2 MQTT with MATLAB Interface

3.2 Kafka MATLAB Interface

Models written in Matlab can be integrated with Kafka directly as well. The Kafka library can be installed in Matlab which contains Kafka clients, producers and consumers. Kafka consumer can be used to consume real time data from Kafka topic which updates the model with real time values and produces it back to a Kafka topic which can be further stored in a database and later sent to a visualization tool. This approach limits the dependency on MQTT protocol and provides direct linkage of Kafka and Matlab [10]. The Kafka integration with Matlab works fine with basic operations but there are certain limitations of the library. There is no parameter to get the latest values from a Kafka topic so, if the Matlab interface is started after some delay the model operates on old values. A solution would be to change the group id of a Kafka consumer every time it starts but it is not desirable in some cases when we actually need some offset values of Kafka topic from past.

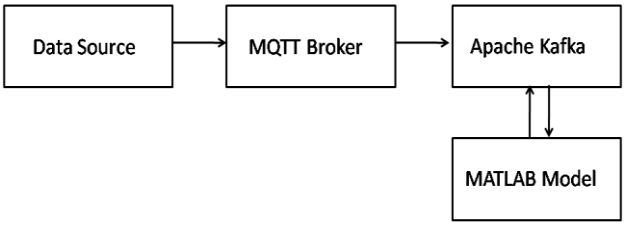

Figure 3 Kafka with MATLAB Interface

3.3 Kafka Java Interface

The library kafka-clients-2.8.0 provides an interface between Java and Kafka [12]. As discussed earlier, Kafka is hugely scalable, and the models written in Java can also be integrated with Kafka by installing the Kafka Producers and consumers libraries in Java. Kafka consumer written in Java consumes the real time values from a Kafka topic and updates the model accordingly. The desired values from the Java model can be produced to a Kafka topic which can further stored in a database or send to other applications. The integration of models written in different programming languages is what makes Kafka a really powerful tool in today's industry environment.

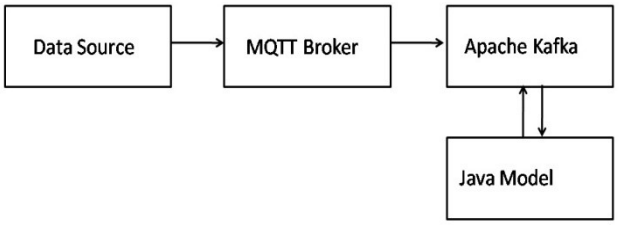

Figure 4 Kafka Java Interface

3.4 Kafka Java MATLAB Queue

To eliminate the issues of the interface discussed in section 3.2 Kafka is integrated with Matlab in a more reliable way, by implementing a Java-Matlab queue.

Kafka consumer written in Java consumes the real time values from a Kafka topic and sends it to a Java queue. The Matlab client continuously keeps a check on the Java queue and as soon as some data is received in the queue, the data is extracted by Matlab client, processed by a Matlab model and sent back to Java queue where the Java client gets the data and produces it a Kafka topic which can be further sent to different applications. This implementation provides better flexibility as all the features of Java Kafka library can be utilized for data acquisition from Kafka.

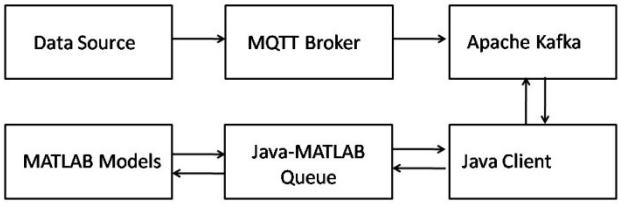

Figure 5 Kafka Java-MATLAB Queue Interface

4 Performance tests

The flexibility of DT platforms comes at some computational costs for data base interactions and cloud services. Especially the response time (**RT**), until the physical object receives a control action from the DT has to be considered. The RT might be slower in comparison with manual direct software linking of models and sensors. The RT was measured for different model frameworks and configurations of the platform. The contribution of each platform component was analysed separately.

4.1 Methods

The SHT31 temperature and humidity sensor from Sensiron was used for attaining temperature and humidity data. Sensor data were either send by hardware sensors, connected via Raspberry Pi 3 to Ethernet or playback previously recorded data during transportation of bananas from Costa Rica to Europe [13].
Tests were carried out on a virtual server (max 4 cores, 8 GByte memory) running on an AMD EPYC-Rome machine with 3 GHz CPU clock. Connectivity to local PCs was within the same local network of our University via Ethernet. Additional tests were done over Wi-Fi.
Each software component added a timestamp, so that time differences could be recorded for each processing step. Clock deviations of different PCs had to be compensated. A series of at least 30 timestamps was recorded for each test and evaluated.

4.2 Timing Analysis

The time delays of different software components of each model interface were recorded and analysed. The time delays are in milliseconds and are shown in the form of boxplots.
A boxplot is used to show the distribution of numerical data. The minimum value of the data is shown by the left most point as in Figure 6 below, while the right most point show the maximum value. The red line in the middle show the median. The lower quartile that is between Q1 and median shows almost 25th in the data, while the upper quartile between median and Q3 in Figure 6 shows 75th percentile. The interquartile is between Q1 and Q3. The lower and upper whiskers represent lower 25% of data values and upper 25% of data values respectively. Outliners in boxplot are values that is distant from the rest of the data. For demonstration outliners were excluded in this implementation.

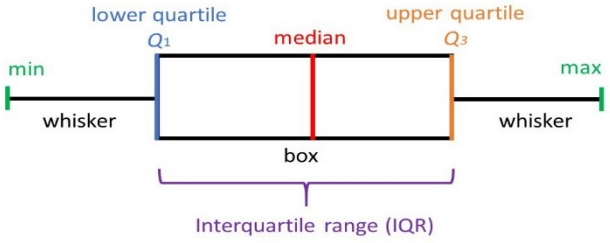

Figure 6 Parts of Boxplot [14]

Two scenarios were tested for the DT solution:

a) Real time humidity and temperature sensor data was sent from Raspberry Pi 3 to the MQTT broker installed in on a virtual server. The sensor data was then received and processed by the different model interface and the result was send back to Raspberry Pi 3 through Kafka.
b) Detailed testing of contribution of each software components involved in the DT solution was done. The recorded data set was used in this case instead of real time data [13]. The data was sent to MQTT broker from where it was produced to a Kafka topic. A model written in Java processes the data and return the result back to Kafka. The final output from Kafka was sent to a MQTT broker and the round trip time was recorded along with the delay from each software components.

4.3 Scenario a) comparing model interfaces

The boxplot below shows the round trip time of the different model interfaces. The first measurement was done without a model interface.

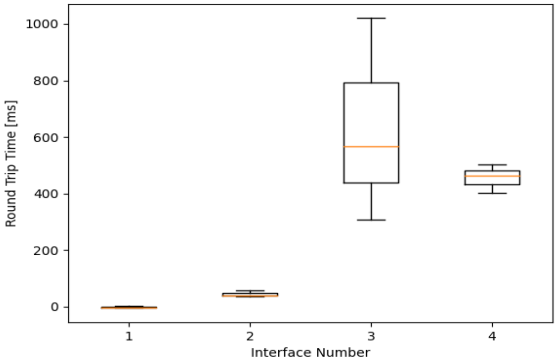

Figure 7 Round Trip Time for different model Interfaces 1) Python Kafka Consumer 2) Kafka Java Interface 3) Kafka Matlab Interface 4) Kafka Java Matlab Queue

4.4 Scenario b) contribution of software components

The below figure represents the different timing measurements done in our implementation of different software components.

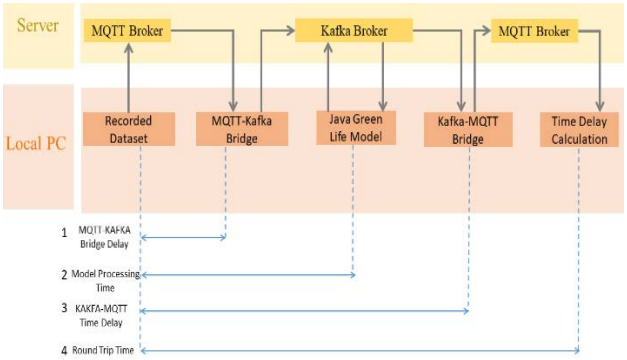

Figure 8 Timing Measurements of Software components 1) Data source to MQTT-Kafka bridge 2) Model processing Time 3) Kafka-MQTT bridge 4) Round trip time

The boxplot below shows the time delays each software components according to **Figure 8.**

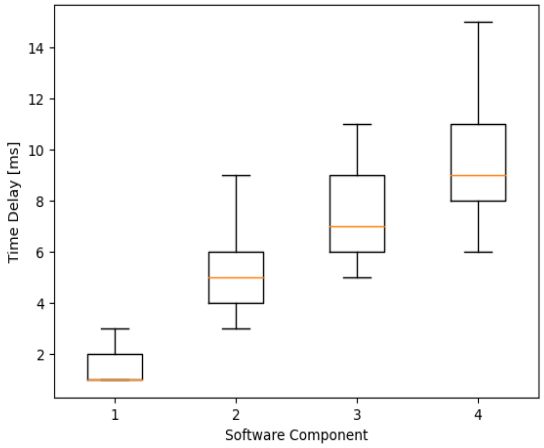

Figure 9 Accumulated time for components running on PC. From Data source to 1) MQTT-Kafka bridge 2) Java Model 3) Kafka-MQTT bridge 4) Data sink

The different software components were also ran on the server for the scenario b) and the result of the timing delays is shown in the boxplot below.

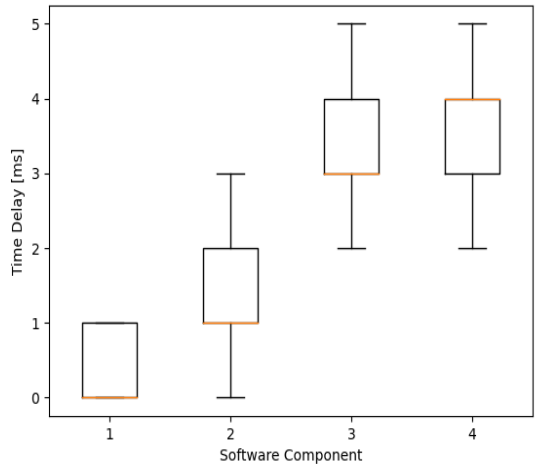

Figure 10 Accumulated time. Repeated test for all components running on the server.

4.5 Green-life prediction

Quality changes of fruits in relation to temperature deviations were predicted by the models [13]. A green-life model for bananas was programmed in Java and Matlab as example for such a quality prediction model. Figure 11 shows the continuous loss of green-life over the duration of the transport, i.e., the remaining timespan until an unwanted ripening process starts, accompanied by colour change from green to yellow.

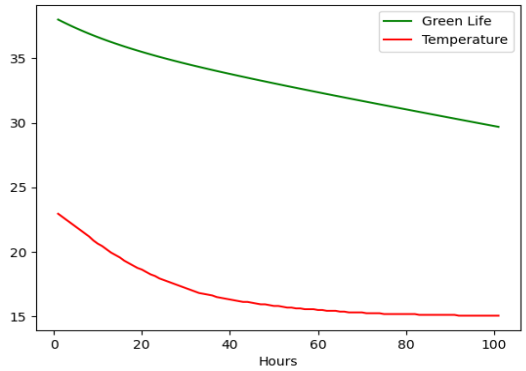

Figure 11 Green Life and Temperature curve with respect to Time in hours.

5 Discussion and Conclusion

Our DT sample implementation shows how sensor data can be integrated with multiple models in a flexible manner with acceptable network overhead. The timing analysis done in section 4.2 for scenario a) demonstrated that the Matlab Interface with Kafka took the maximum time to process a model and return the result back to Kafka while the Kafka Java-Matlab Queue was significantly faster which can be a better alternative. The Kafka Java interface had a delay of around 30 ms including communication with the Raspberry Pi.

The testing done for scenario b) concluded that the virtual server where each software component was running on the server itself was much faster than the local PC with embedded environment in which MQTT and Kafka Broker was run on the server. Moreover, the testing with Ethernet was substantially faster than the Wireless network, which had an overhead of almost 200ms.

The tests performed with pure server database with fast Java interface demonstrated that the problem is not the database but the different environments on which the tests were run. Local network added an extra overhead of almost 8ms in comparison to pure server testing of Java interface while the Wi-Fi gave us an overhead of almost 200ms. The most timing delays were observed while running the Matlab model which added an overhead of roughly 800ms. The tests performed on Raspberry Pi 3 were also significantly slower than the server adding a delay of around 35 ms. The initialization and creation of topics and consumer groups in Kafka also contributes around 1sec to the overhead in this DT solution. If all initialization is done in advance, our Kafka based solution provides a fast and flexible digital twin platform.

ISBN 978-3-8007-5835-7

6 Literature

[1] Jedermann, R., Praeger, U., Geyer, M., Lang, W.: Temperature deviations during transport as a cause for food losses. In E. Yahia (Ed.), Preventing food losses and waste to achieve food security and sustainability. Sawston, UK, BurleighDodds (2019). doi:10.19103/AS.2019.0053.12

[2] Defraeye, T., Shrivastava, C., Berry, T., Verboven, P., Onwude, D., Schudel, S., et al..: Digital twins are coming: Will we need them in supply chains of fresh horticultural produce? Trends in Food Science & Technology, 109, 245-258 (2021). doi:10.1016/j.tifs.2021.01.025

[3] Kamath, V., Morgan, J., Ali, M. I. Industrial IoT and Digital Twins for a Smart Factory: An open-source toolkit for application design and benchmarking. In: 2020 Global Internet of Things Summit (GIoTS), 1-6 (2020) doi:10.1109/GIOTS49054.2020.9119497

[4] Sollander, A., Hosagrahara, A.: MATLAB Interface for Apache Kafka. https://github.com/mathworks-ref-arch/matlab-apache-kafka, last accessed 27. Jan. 2022

[5] Overview of Microsoft Azure Digital Twins https://docs.microsoft.com/en-us/azure/digital-twins/overview

[6] AWS IOT device shadow service. https://docs.aws.amazon.com/iot/latest/developerguide/iot-device-shadows.html

[7] Eclipse Ditto documentation Overview. https://www.eclipse.org/ditto/intro-overview.html

[8] M. Grieves and J. Vickers, “Digital Twin: Mitigating Unpredictable, Undesirable Emergent Behaviour in Complex Systems,” in Transdisciplinary Perspectives on Complex Systems: New Findings and Approaches, F.-J. Kahlen, S. Flumerfelt, and A. Alves, Eds., Cham: Springer International Publishing; Imprint; Springer, 2017, pp. 85–113.

[9] MQTT in Matlab: https://github.com/HighVoltages/MQTT-in-MATLAB

[10] Kafka Matlab Interface: https://github.com/mathworks-ref-arch/matlab-apache-kafka

[11] Benchmarking Apache Kafka. Alok Nikhil, Vinoth Chandar (2020) https://www.confluent.io/blog/kafka-fastest-messaging-system/

[12] Kafka-client library of Java. https://jar-download.com/artifacts/org.apache.kafka/kafka-clients

[13] Jedermann, R.; Lang, W.: 15 Years of Intelligent Container Research. In: Freitag, M.; Kotzab, H.; Megow, N., (eds.): Dynamics in Logistics: Twenty-Five Years of Interdisciplinary Logistics Research in Bremen, Germany, Springer International Publishing, Cham, 2021, pp. 227-247. (doi: 10.1007/978-3-030-88662-2_11)

[14] Saul McLeod, 2019: Description of Boxplot, https://www.simplypsychology.org/boxplots.html

