Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Distributed Data Store for Internet of Things Environments

Jonas Auer

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr.-Ing. habil. Bernhard Mitschang
Supervisor:	Dr. rer. nat. Matthias Wieland
Commenced:	May 08, 2017
Completed:	November 08, 2017
CR-Classification:	H.3.4

Abstract

The Internet of Things (IOT) is a rapidly uprising technology. It is gaining importance in smart factories, often referred to as Industry 4.0. Given the amount of data that is collected in these environments, storing and analysing the accumulated data is a major concern. The aim of this thesis is to develop and evaluate a concept for a distributed data store, in which data is first collected at the edge of the Network, on premise, and then sent on to a central data store where observations spanning multiple edge deployments can be made. The final concept proposes the use of a database, a stream processing framework for reacting to messages as they arrive at the servers and a message queueing system to act as a common interface for the other components on both the data centre and the on premise deployments. The edge and central servers are connected through a service subscribing to selected edge queues and publishing their content on the central data store. An example deployment was implemented and proved to be capable of processing 10.000 messages per second with a mean latency of less than 4ms for the on premise deployment and 67ms for the central data store.

Kurzfassung

Das Internet der Dinge (Internet of Things, IoT) ist eine schnell aufsteigende Technologie. Sie gewinnt an Wichtigkeit in intelligenten Fabriken, oft auch Industrie 4.0 genannt. Wegen der großen anfallenden Datenmengen in diesen Umgebungen ist die Speicherung und Analyse der gesammelten Daten von großer Bedeutung. Ziel der Thesis ist es, ein Konzept für einen verteilten Datenspeicher zu entwickeln, in dem Daten erst am Rande des Netzwerks (on premise) gesammelt werden und anschließend an einen zentralen Datenspeicher weitergeleitet werden, wo Beobachtungen über mehrere Edge-Deployments gemacht werden können. Das finale Konzept sieht die Benutzung einer Datenbank, eines Stream-Processing-Frameworks, zum Reagieren auf Nachrichten, sobald sie bei den Servers ankommen, und eines Message-Queuing-Systems vor, das als gemeinsame Schnittstelle der anderen Komponenten der on-premise und zentralen Deployments dient. Die Edge-Datenspeicher sind durch einen Dienst mit dem zentralen Datenspeicher verbunden, der ausgewählte Edge-Queues abboniert und deren Inhalt auf dem zentralen Datenspeicher veröffentlicht. Ein Beispiel wurde implementiert und aufgesetzt, das sich als fähig erwies 10.000 Nachrichten pro Sekunde mit einer durchschnittlichen Latenz von 4ms bis zum Edge-Datenspeicher und 67ms zum zentralen Speicher.

Contents

List of Figures

List of Listings

1. Introduction

The Internet of Things (IoT) is a rapidly uprising technology. It allows connecting a multitude of sensors, actors and services, to form smart environments. Smart homes and smart factories are both examples for emerging use-cases of IoT environments. The technology is especially interesting in the context of Industry 4.0, where it allows sensors, devices and machines to communicate with each other. Having the members of the IoT network speak a common language simplifies the automation of factories, possibly allowing devices to communicate directly with each others and to supervise and control whole production processes, without requiring extremely specialised software. All parts in the IoT system adhere to certain standards, allowing easy interoperability.

The more devices are connected to such a system, the more data is produced that needs handling. Many messages can be directly reacted upon, for example a machine could be turned off, when its internal temperature exceeds a certain limit.

However, only reacting to device messages is often not enough. The data needs storing to enable later analysis like the identification of trends or predictive maintenance and to guarantee that data is processed, even when parts of the service architecture fail.

For large firms with a multitude of factories, it might not always be feasible to only deploy one central server cluster to manage all factories. A possible solution would be to provide server clusters on premise to perform edge computing and employ a single data centre were all relevant data is collected to perform long term analysis. This allows for the timely processing of device data on premise, reducing the required processing power at the data centre. It also increases fault tolerance, since the on premise deployments can act independently from the central cluster. Even if a whole factory's cluster should fail, the others can continue without interruption.

The aim of this thesis is to design such a system, where data storage and analysis is distributed among fast-acting edge deployments and a central component as illustrated in fig. 1.1. Devices should be connected to the edge data storage, which in turn should relay data to the central data store. Both kinds of data stores should support performing data analysis. The analysis systems should be able to read data from and write data back to the storage systems. The central component should collect all relevant data from the on premise clusters. Collected data needs to be persisted in a way to account

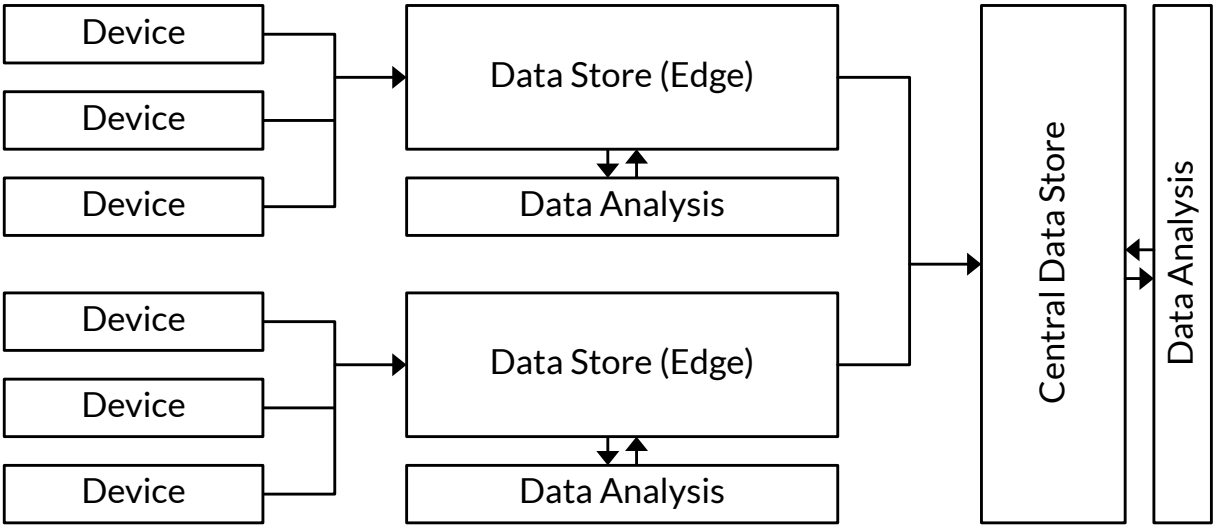

Figure 1.1.: High-Level Architecture of the Distributed Data Store

for failures and long term analysis. The individual components have to be connected appropriately, for example using a message queuing system. How data enters the system is not part of the thesis, but it should be possible to ingest data from a broad spectrum of data sources with little adaptation. The concept should be capable of being deployed on custom hardware and explicitly make use of open source software accessible for free.

In chapter 2, Concept for a Distributed Data Store, existing IoT systems and platforms are investigated and checked for usability. This includes distributed NoSQL database systems, an IoT and data streaming platform and time series databases. A concept for the distributed data store is proposed. It is compared to the Resource Management Platform (RMP) introduced by Hirmer et al. [HWBM16a] and Hirmer et al. [HWBM16b], which was designed and implemented in previous projects.

Chapter 3, Message-Based Implementation Using Apache Kafka, introduces an example use case for the system, focused around Apache Kafka. It is subsequently implemented based on the findings in chapter 2.

Chapter 4, Automatic Deployment of the Distributed Data Store, demonstrates how the solution shown in chapter 3 can be deployed to a cluster of servers using Docker for packaging and distributing the individual components of the architecture.

In chapter 5, Evaluation of the Distributed Data Store, the running infrastructure is evaluated, focussing on the data latency based on the number of messages the clusters are handling.

The thesis ends with chapter 6, Conclusion and Outlook, reflecting on the findings and showing ways in which the results can be utilised in future works.

2. Concept for a Distributed Data Store

For coming up with a concept for the distributed data store, it is important to know what kind of data the system must be able to handle. The use case for IoT devices is to collect data and sending it to the edge data store. The types of sensor measurements that can be transmitted should not be constrained in any way, or at least as freely configurable as possible.

What all measurements share is that they are recorded at a certain time. This time will always have to be transmitted along the measurement data itself, to allow to reconstruct a measurement history and for example plot the sensor data on a graph with a time axis. Therefore, all data passing through the system and originating from the devices are time series data.

While the thesis does not cover a back-channel from the servers to the devices (e.g. for issuing commands), the data required for this would be addressed to a certain device or actor and could bear a timestamp of when the command was originally issued, effectively making these commands time series data as well. Despite the communications channel from the server to devices not being a hard requirement, it is a central aspect of a complete IoT solution to be able to remotely control devices. Therefore the decisions made in this chapter will consider this use case, enabling future work to build on the proposed solution.

To develop the final concept, first potential existing solutions are investigated. Afterwards, a general architecture is proposed and systems are chosen for each of the architecture's components. These will then be used in chapter 3, Message-Based Implementation Using Apache Kafka, and chapter 4, Automatic Deployment of the Distributed Data Store, for implementing and deploying an example application of the concept.

2.1. Evaluation of Possible Solutions

Before coming up with a concept, potential existing solutions for the problem are examined.

2.1.1. Automated Database Replication

Many databases support replicating data between whole data centres out of the box. PostgreSQL, for example, offers a whole list of different solutions for data replication, including replication through log-shipping, where commit-logs are sent to a slave-server which applies them to reach the same state as the master server [Pos17b]. MariaDB explicitly supports replicating data from multiple servers to a single slave server[Mar17].

However, using built-in replication methods of database systems has one key drawback: it doesn't allow to easily react to incoming changes. Seeing that it is a requirement for the server to be able to intercept data arriving from the on premise clusters as it comes in, this kind of replication does not fit the use case.

2.1.2. IoT Platforms

IoT platforms cover a whole set of IoT related use cases, from provisioning devices, over ingesting data from devices into the system to storing and analysing IoT data. Google, IBM and Microsoft provide these solutions with their products Google Cloud IoT[1], Microsoft Azure IoT Suite[2] and IBM Watson IoT Platform[3].

These solutions have in common that they are all cloud hosted and they do not provide self-hosted variants free of charge. They therefore were ruled out as viable solutions.

A free and open source IoT platform was found in the IoT Distributed Service Architecture (DSA)[4].

> "DSA allows for purpose-built products and services (i.e. DSLinks) to interact with one another in a decentralized manner. This architecture enables a network architect to distribute functionality between discrete computing resources. A network topology consisting of multiple DSLinks running on edge devices connected to a tiered hierarchy of brokers allows the system as a whole to be scalable, resilient to failure and take advantage of all computing resources available to it from the edge, the datacenter, the cloud and everything in between" [DSA17].

[1] https://cloud.google.com/solutions/iot/
[2] https://www.microsoft.com/en-us/internet-of-things/azure-iot-suite
[3] https://www.ibm.com/internet-of-things/
[4] http://iot-dsa.org

By constructing a nested hierarchy of Distributed Service Brokers (DSBrokers), it is possible to distribute work among the brokers. Queries can be sent to top-level DSBrokers that are broken down into smaller chunks as they are sent down the hierarchy. Query results are re-assembled and joined as they pass back up through the graph. The leaf nodes of the graph are made up of Distributed Service Links (DSLinks), which can be accessed through an interface using a custom querying language. DSLinks can fill many roles, such as generating or storing data or performing data analysis. Essentially, DSLinks wrap some kind of functionality and expose said functionality through a common interface. DSLinks and DSBrokers can communicate via the Websocket protocol, falling back to HTTP, making it easy to make use of existing infrastructure of webservers like load balancers, proxies and firewalls. Pre-built solutions exist for ingesting data from various fieldbus protocols often used with IoT devices, such as Zigbee or ZWave [DSA17].

However, the promoted decentralisation of tasks is contrary to the use case of the thesis. The thesis strives to centralise the data acquisition, making administrating the system simpler, while the DSA aims to utilise as much computational power as possible by distributing as much work as possible among the individual nodes of the network.

2.1.3. The Resource Management Platform

The Resource Management Platform (RMP) addresses the problem of automatically registering sensors and provisioning sensor data. The concept was introduced by Hirmer et al. [HWBM16a] and further expanded by Hirmer et al. [HWBM16b]. I was prototypically implemented using Node.js as part of the SitOPT research project[5]. A sensor registry isntructs the RMP to create resources for registered sensors and deploys adapters that read and understand sensor data and send them to the RMP. There, access to sensor data is provided for clients through a REST API and MQTT topics are created for every sensor so that the data can also be read using a publish/subscribe pattern [HWBM16a].

Seeing that the RMP on itself does not provide means for persistent data storage, a storage adapter would need to be developed that automatically subscribes to certain topics. It would then have to relay all received data to a database for long term storage.

To realise the hierarchy of on premise RMPs and a central one, a mechanism would need to be added that allowed local RMPs to be registered with the central one. Those also would have to transmit sensor registration information and sensor data to the central data store.

[5] https://www.ipvs.uni-stuttgart.de/abteilungen/as/forschung/projekte/SitOPT

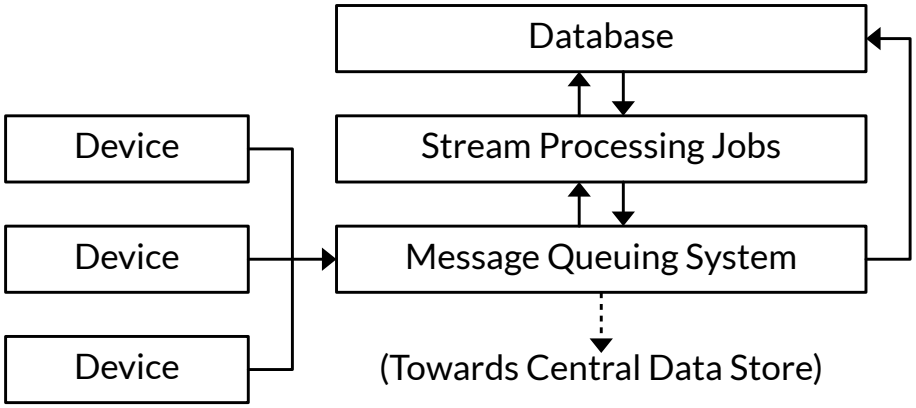

Figure 2.1.: Concept for the On-Premise Data Store

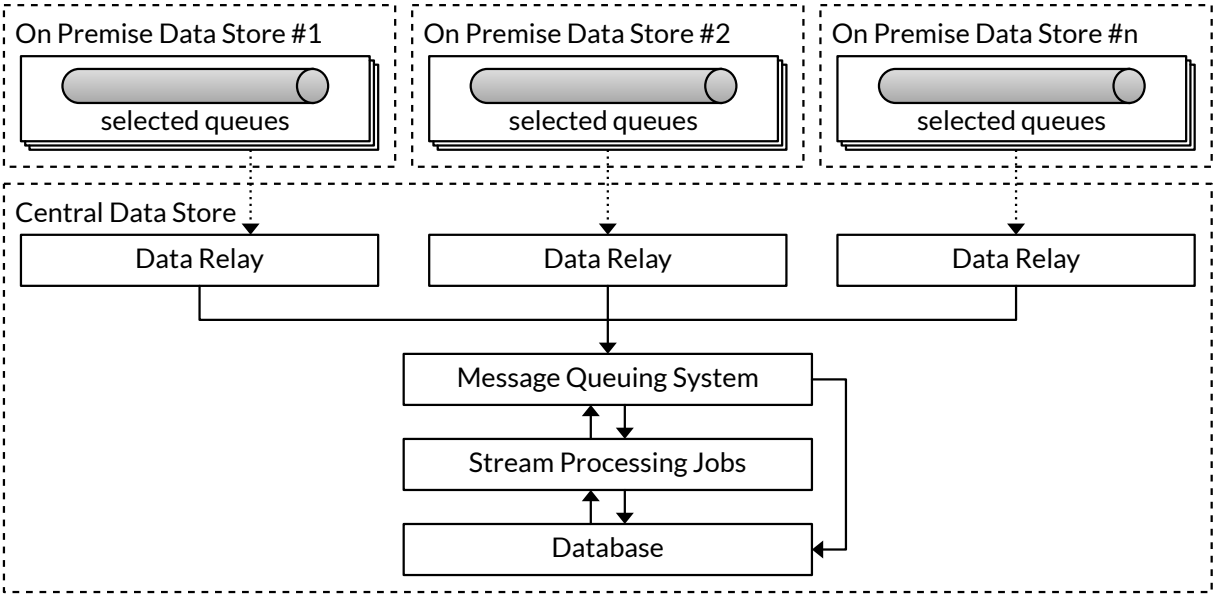

Figure 2.2.: Concept for the Central Data Store

While the RMP addresses important use cases in a full scale IoT network deployment in the real world, it was decided against building directly on it. Instead, a flexible concept is introduced that can be adapted to a large number of use cases.

2.2. The Proposed Concept

The concept proposed in this section lends itself some core principles of existing IoT solutions by Google, IBM and Microsoft. Each server cluster, on premise deployments and the data centre, is running a message queuing system at its core. For analysing data as

early as possible when it comes in, a stream processing framework is employed. Further, selected data queues are stored in a database for later analysis and data exploration. Device data is directly sent to message queues on premise. Data from multiple on premise data stores is collected at the central data store by transmitting data from selected topics of the on premise message queuing systems to corresponding message queues of the central data store. Fig. 2.1 shows the proposed concept for the on premise data store. Fig. 2.2 visualises the concept for the central data store, including the connection between the two kinds of data stores.

These three components are the same on both the on premise deployments and the data centre, to make development, deployment and maintenance easier. The architecture is flexible enough to allow for using the same stream processing jobs on both the data centre and the edge clusters, making it possible to move certain processing steps from the edge to the centre or vice versa.

The main difference of the on premise and central data stores is where the data processed by the systems originates from. For the on premise data stores, the data source are the individual devices. The exact device interface is not part of this concept, but the usage of message queues allows to connect devices communicating through virtually any protocol to the system. All that is required to incorporate a new device message protocol is to add a producer that reads device data and injects it into the correct queues.

The data centre gets all its data from the servers on premise. For this, a data relay is utilised that acts as a consumer for the message queuing systems on premise and as a producer for the message queuing system of the central data store.

2.2.1. Message Queuing System

Message queuing systems, as the name suggests, provide queues for messages. Messages can be appended to queues by producers. Consumers read messages from the queues and process them. Queues are uniquely identified by names, which are used to write to and read messages from the queues. For example, a device can act as a producer, sending data in defined intervals to a `sensor-data` queue, while a consumer reads the data from the queue and sends it to a database.

The use of queues decouples consumers from producers. As long as producers write data in the same format to a given queue, the consumers reading from that queue are capable of handling the data and do not have to care about where the data originally came from. This provides the required flexibility to handle all kinds of data. Decoupling producers from consumers makes their development easier, since teams can work on them independently, as long as they agree on a common message format.

2.2.2. Stream Processing Framework

For performing data analysis on incoming data as it arrives, a stream processing framework is introduced. Stream processing frameworks are designed to perform actions on streams of data, such as converting them to another format and filtering or accumulating the streamed data. Streams can be obtained from message queues and the resulting streams of data can again be sent to another queue, making stream processing frameworks a natural fit for interacting with message queuing services. Should a stream processing job not be able to handle all data as soon as it arrives, the queue fills up, acting as a message buffer and allowing the processing job to catch back up when the rate of incoming messages drops. This allows the system to handle periods of higher than usual rate of incoming data without much hassle, albeit at the cost of increased latency during the peak.

Stream processing also allows to break up complex operations into multiple steps, further simplifying the development and maintenance of processing jobs. When messages are handled on their own and without a shared state, parallelising stream processing jobs to increase their throughput comes down to running multiple instances of the same job and distributing incoming messages to the individual worker nodes.

2.2.3. Database

Since simply reacting to data is not enough, a database is added to store data for longer periods of time. This allows to explore historical data and possibly identify emerging trends and make predictions about future data. Data from selected message queues is continuously read by a consumer that writes received messages to the database. Thanks to message queuing, stream processing jobs do not have to take care of what data has to be written to the database and the database connection itself is neatly abstracted away. If at some point it is decided, that data of other queues should be sent to the database as well, it is enough to start up another database injector or tell an already running injector to read an additional queue.

2.2.4. Devices

Device data can be sent to the system using any producer that puts correctly formatted messages into the correct message queues. Again, thanks to the message queue abstraction, the stream processing jobs do not need to know where the data originated from. This allows to build adapters for a multitude of device communication protocols. Such a

adapter would listen for device messages, possibly transform them and then send them to a matching message queue.

The example implemented in chapter 3, Message-Based Implementation Using Apache Kafka will send data directly to a message queue, not requiring any adapter at all. This might not be feasible for devices with little memory or without an Ethernet interface, so for a larger environment in a real world, adapters would have to be built to get data from such devices into the system.

2.2.5. Data Relay

For transmitting data from the on premise deployments to the central data store, a data relay is introduced. It acts as a consumer for on premise message queuing system and as a producer for the message queuing system of the central data store. The data relay itself runs on the central data store, making the on premise deployments completely independent systems capable of running without needing to know about the central data store. The data relay of the central data store only shows up as another consumer to the message queuing systems on premise, allowing the data relay to copy data without any other job having to be explicitly developed to enable data replication.

3. Message-Based Implementation Using Apache Kafka

In order to come up with the actual implementation of the concept, an example use case is chosen around which the final prototypical implementation is built. Implementation components are then selected based on the data that needs to be processed and stream processing jobs, data relays and database adapters are built around that infrastructure.

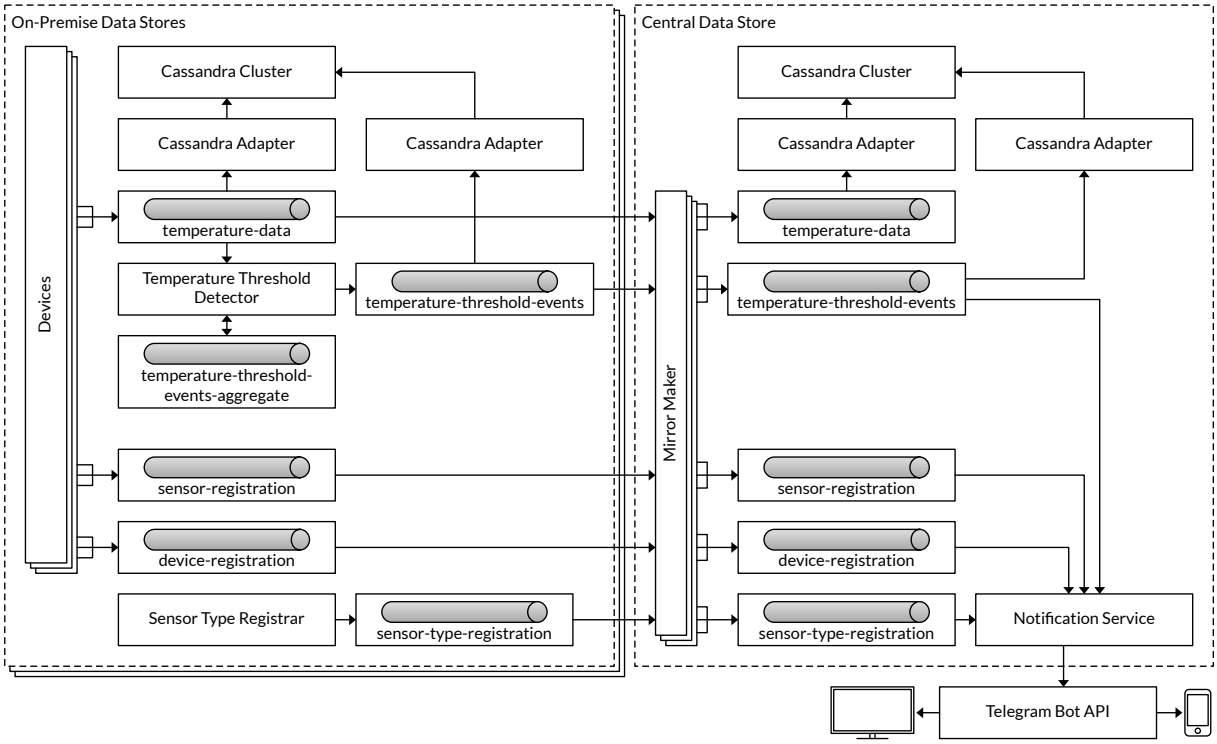

Figure 3.1.: Queues and jobs of the example use-case

3.1. Example Use Case

Temperature data is collected by devices and sent to the servers on premise. The temperature data is analysed for the exceedance of pre-defined thresholds. When a sensor starts exceeding the threshold, a message is generated, containing the sensor's current temperature and the timestamp when the data was measured. When the temperature value drops below the threshold, another event is generated, stating that the sensor no longer exceeds the threshold, along with statistics on how long the threshold was exceeded and what the average and maximum temperature were during that period. The threshold events are relayed to the central data store, where a stream processing job reads the messages and combines the events with registration information of the corresponding sensor and device. From this information, a notification message is built, which is then sent to Telegram conversation through the Telegram Bot API.

3.2. Selection of Implementation Components

The concept introduced in chapter 2, Concept for a Distributed Data Store is intentionally vague in order to allow choosing the individual implementation components depending on the exact use case at hand. There are numerous established message queuing systems available, each having distinct advantages and drawbacks compared to other solutions. The same holds for stream processing frameworks and databases alike. Especially with databases, it is trivial to add a second database that better fits some use case better than the first one, since it only requires building another database injector that takes data off a queue and inserts into the database.

3.2.1. Message Queuing Systems

At the heart of the concept lies the message queuing system. It is what all other systems connect to and the where all communication will run through, making it the only system in the concept that is not easily replaced with a different one. Therefore, the message queuing system must be chosen with greater care than the other systems.

For this thesis, two message queuing solutions were considered: RabbitMQ and Apache Kafka.

RabbitMQ

RabbitMQ implements multiple messaging protocols for writing data to and receiving data from the system. It was originally built to support the Advanced Message Queuing Protocol (AMQP)[1] standard in versions 0.8, 0.9 and 0.9.1. Through plugins, support can be added for the Simple Text Oriented Messaging Protocol (STOMP)[2], MQ Telemetry Transport (MQTT)[3], AMQP version 1.0 and even HTTP [Piv17d].

RabbitMQ uses Message Queues and exchanges to store and distribute data among consumers. Producers can either publish data directly to a queue or send it to a exchange. A queue simply holds the message until it is read and acknowledged by a client. After a successful read, the message is then popped off the queue. This has an important implication: if multiple clients subscribe to the same queue, messages in that queue are each only sent to exactly one of the clients before being popped off the queue. While this means that data in the queue can easily be distributed among worker nodes to spread out the work load, it makes it impossible for multiple types of workers to receive all messages published through a queue [Piv17e; Piv17a].

This is were exchanges come into play. When sending a message to a exchange instead of a queue, the exchange decides what to do with the message. Depending on the type of exchange, they support various delivery patterns:

Using a `fanout` exchange, data can appended to multiple queues. Producers can simply publish to a named `fanout` exchange. Clients are responsible for creating a queue and binding its input to the exchange. The exchange will then append all received messages to all queues bound to it. This way, multiple types of consumers can access all data sent to a single exchange [Piv17a].

A `direct` exchange is also available. It makes use of the possibility to assign one or more `binding keys` to queues and `routing keys` to messages. Basically, this implements a publish/subscribe pattern. Consumers declare a queue just as in the `fanout` case. Then, the queue can be bound to the exchange, passing in an additional `routing_key`. When receiving a message, the exchange then checks the message's `routing key` and appends the message to all bound queues bearing an identical `binding key` [Piv17b].

The `topic` exchange builds on top the `direct` exchange and introduces the concept of "topics". It does so by extending the `routing key` concept and allowing to specify multiple sub-keys, separated by dots (`.`). When binding to a `topic` exchange a whole key (e.g. `a.b.c`) can be given. But where the `topic` exchange really shines is the ability to

[1] http://www.amqp.org/
[2] http://stomp.github.io/
[3] http://mqtt.org/

specify wildcards within the binding key to subscribe to `message keys` that fit parts of the `binding key`. For example, a queue bound using the key `a.b.*` would receive messages with the keys `a.b.a`, `a.b.b` and `a.b.c`, but not `b.a.a`. Here, the star (`*`) accepts any sub-key at its place. Another wildcard, the hash character (`#`), can be used to substitute any number of sub-keys, equal to or greater than zero [Piv17c].

Apache Kafka

Kafka differs from RabbitMQ in a number of ways: Messages passing through Kafka each bear a key, a value and a timestamp. Queues are called "topics" in Kafka and further broken down into partitions. Producers send data directly to topic partitions and consumers read straight from them. Partitions are distributed among Kafka nodes (called "brokers") and can be replicated among them. For each partition, Kafka makes use of a ZooKeeper cluster for electing a partition leader, to which all messages directed at the partition are sent. The partition leader then writes the data to disk and replicates it to the "follower" nodes carrying the partition replicas.

Producer applications are responsible for choosing partitions for all sent messages. Per default, the Kafka client library applies a hash function to the message's key to determine the target partition, but a custom partitioning function can be used as well. The default behaviour ensures that messages with the same key always are routed to the same partition. It may be necessary to supply a custom partitioning function if all messages passing through the topic hold the same key, as they would otherwise all be routed to the same partition, leaving all others running empty.

Since consumers read at least one partition and cannot specify any filtering rules for obtaining only certain messages sent to the partition, they have to consume the whole partition. This makes the number of partitions of a topic also an upper limit for running consumers in parallel: there cannot be more consumers than the topic has partitions or the others would run idle.

In contrast to RabbitMQ, Kafka moves the responsibility to keep track of consumed messages to the consumer itself. This frees the Kafka cluster from having to keep track of the position of every single consumer. It also means that Kafka does not know when a message is read by all consumers subscribed to the topic partition. Kafka assigns each record of a partition a unique, sequentially increasing offset. Consumers keep track of their offset within the topic. This allows consumers to re-read older entries, read the whole topic from its beginning or skip new records. The official Java Kafka client library uses ZooKeeper to persistently store consumer offsets, so consumers can continue were they left off after crashing.

Per default, Kafka holds all records for a week before deleting them. However, this setting can be changed per topic, allowing Kafka to store all data ever received on a topic or only store a short time window of data. In combination with the persistent storage of records, this makes Kafka very failure-resistent, as the whole cluster can crash without any data that was successfully written to the cluster being lost [Apa17d].

The example implementation uses Apache Kafka as its messaging system. The deciding advantage over RabbitMQ was its capability to persistently store all data without discarding it as soon as it is read. While this does not remove the requirement of a dedicated database to store the sensor data, tasks like device registration can all be handled without the need for another storage solution, which is taken advantage of in sec. 3.6.2 for getting device metadata for a given sensor ID. This comes at the cost of a more complicated deployment process, since Kafka requires a running ZooKeeper cluster. Also the Kafka and ZooKeeper instances cannot be started with the exact same configuration, but have to be each configured individually. This is further addressed in chapter 4.

3.2.2. Stream Processing Framework

To process streams of data passing through the Kafka topics, two stream processing frameworks were investigated.

Apache Storm

Apache storm makes use of topologies consisting of “spouts”, streams and “bolts”. Spouts act as data sources for streams of tuples, reading data from an external source. In the example this source would be an Apache Kafka topic. Bolts do the actual processing work by applying operations on the incoming tuple streams. They can declare any number of output streams, including zero. The Storm topology describes what bolts receive which streams as their input.

Bolts can be run in parallel, running multiple tasks at once. Tasks are distributed among multiple worker nodes, which each can run multiple bolt task threads.

Since a Storm stream would receive *all* records sent to a specific Kafka topic, Storm supports the notion of stream groupings. A stream grouping defines how the stream’s data should be partitioned before sending data to the individual bolt tasks. Storm features eight grouping strategies out of the box and it is possible to add custom ones. Among those strategies are shuffle grouping (randomly distribute tuples among bolt tasks) and fields grouping (partition the tuples by given fields of the tuple) [Apa15a].

Storm streams therefore share a lot of the properties of Kafka topics, but implement them in a slightly different way. Apache Storm also requires setting up a separate cluster of Storm instances, requiring the addition of yet another application cluster to the deployment. It too requires the usage of ZooKeeper for coordinating the cluster [Apa15b].

Kafka Streams

Since version 0.10, the Apache Kafka client libraries include the Kafka Streams API. It offers a simple way of writing stream processing applications, taking advantage of the way Kafka topics work.

Similar to Apache Storm, topologies are at the core of the Streams API. They are built with Java using a simple Domain Specific Language (DSL). The whole topology can be built using this DSL. A Kafka topic is selected as the input source, which acts as the job's input stream. The stream can then be transformed using operations like `map` and `filter`. Streams resulting from stream operations like `map` can be stored in Kafka topics, allowing for intermediate results to be persistently recorded. Multiple streams can be joined and the API even supports transforming streams into tables for easy querying of values given a key. This allows to join streams with tables, making it possible to implement powerful `reduce` operations [Jay16]. This is explained in further detail in sec. 3.6.

Kafka Streams act as simple consumers and producers for the Kafka cluster, making them easy to integrate in an existing infrastructure. They also do not require another cluster for managing the individual application instances. In fact Kafka does not support any automatic deployment of such Stream jobs at all, leaving it up to the system administrator to find a way to distribute the instances among the cluster nodes. While this may seem to add an additional burden for deploying the services, it actually makes deployment easier, since the tools used for deploying the Kafka and ZooKeeper clusters can just as easily be used to deploy and spread the individual Stream jobs, as seen in chapter 4.

In the end, to keep things simple and in order not to add yet another dependency to the system, the Kafka Streams API was chosen in favour of Apache Storm for its simplicity tight integration with Kafka. This of course does not allow stream processing jobs written with the Streams API to be ported to other messaging systems, making solutions less portable should the stream processing system ever need to be changed.

3.2.3. Database

Looking at databases, three potential candidates covering different groups of database types were looked at.

PostgreSQL

PostgreSQL is “the world’s most advanced Open Source database” [Pos17f]. It is a object-relational database system and uses the Structured Query Language (SQL) to describe tables and their relations and to perform queries on the tables to insert, modify and delete database entries [Pos17a].

Each entry in the table is uniquely identified by a primary key. Primary keys can also consist of multiple fields of the row. For the example use case, the primary key would be made up out of the timestamp of the sensor recording and the recording sensor’s ID.

There are client libraries available in many programming languages, allowing to access the database from clients written in Java, C++ and Python, just to name a few [Pos17d].

However, seeing that the Internet of Things is a ever changing network of objects, the actual data a sensor sends might change over time, for example when a software update is applied to a device. Since tables rely on a fixed list of columns, it can be difficult to store IoT data in these kinds of databases, since adding more columns might not always be viable.

Relational databases therefore do not naturally fit the use case, although it should be noted that PostgreSQL added support for basic JSON operations on fields of rows since version 9.2 [Pos17e]. Over time, more and more JSON features were added, with the most recent version 10.0 supporting searching and modifying JSON object hierarchies through special operands [Pos17c]. This could be used to store dynamically changing sensor data in the database, but will probably still fall short when compared to NoSQL solutions that were built to support this from the ground up.

Apache Cassandra

Apache Cassandra is a non-relational, NoSQL database originally developed at Facebook. It differs from a classic SQL database in a lot of ways. For example, Cassandra is built to handle data coming in at a high rate as well as unstructured data, while adding support for read and write scalability. This comes at the cost of giving up ACID transactions or automatic table joins, but often these tradeoffs are worth their costs, especially when

it comes to applications with high throughput. Given the high rate of input achievable with Cassandra, it is an ideal match for consuming large amounts of data generated by IoT devices. Data stored in Cassandra can be queried using the Cassandra Query Language (CQL), which is similar to SQL but was adapted to work with the capabilities of Cassandra's non-relational approach.

Cassandra is capable of automatically distributing data across a cluster, making data replication easy and allowing for reading data from and writing data to any node of the cluster. This also makes deploying Cassandra easy, since it requires very little configuration [Sat17].

Cassandra stores its data in tables. They are "also able to provide very fast row inserts and column level reads" [Sat17]. Individual rows of tables are uniquely identified by their primary key. In the example use case, this is a combination of both the sensor ID and timestamp of the recording. Tables are grouped together in a "Keyspace", which is essentially the equivalent of a database in PostgreSQL. Replication factors and replication strategies are defined at the keyspace level [Sat17].

Getting started with Apache Cassandra is very easy, thanks to its easy deployment, automated replication and the familiar CQL syntax. These were also the key factors in the decision to choose Cassandra over other databases.

3.3. Programming Kafka Clients

Apache Kafka comes with a set of Clients that allow applications to read and write messages to Kafka topics. The official Kafka clients are written in Java, but implementations for other programming languages maintained by the community are available as well [Apa17h].

While the example is implemented in Java only, the messages passed through the Kafka topics will not use any Java-specific format so that clients written in other languages can be added as well, without having to modify any existing source code.

3.3.1. Build System

For building the clients, managing dependencies and the build process, Apache Maven [Apa17k] was used. It was preferred over alternatives like Gradle, which is used to build Kafka itself [Apa17g], since the examples provided by Confluent Inc. use Maven too, and it is well supported in IntelliJ IDEA, the IDE used to develop the example clients.

To make development and subsequently the deployment of Kafka clients easier, this thesis is accompanied by three Maven archetypes which assist in creating the basic project structure for Kafka producers, consumers and streams. The archetypes can be found in the GitHub repository accompanying the thesis[4].

For making them available on a local system and information on how to generate projects from them, read the `README.md` files found in each of the archetype projects.

3.3.2. Kafka Serialisers, Deserialisers and Serdes

Kafka stores all messages in a binary format. In order to obtain readable data out of this, Kafka clients require the use of serialisers and deserialisers. These provide functions to get objects of type `T` from `byte`-Arrays and `byte`-Arrays from objects of a type `T`, respectively. Usually a `Serializer<T>` and a `Deserializer<T>` are combined to form a `Serde<T>`, which can then be used when both a serialiser and deserialiser for the same type are required.

Kafka offers Serdes for the builtin Java types `Long`, `Integer`, `Short`, `Float`, `Double`, `String`, `ByteBuffer` and `byte[]`. Since the data used in the example use case contains objects holding multiple values, the default types will not suffice.

While it would be possible to make the data objects implement Java's `Serializable` interface and use that to serialise the objects, the serialised format is Java-specific and therefore would not easily work in combination with clients developed in other languages. To work around this issue, Apache Avro™is used throughout the example.

3.3.3. Apache Avro™

Apache Avro™is a system for serialising data. It relies on schemas to serialise and deserialise data to/from a compact binary format. While this requires the schema to be available both at serialisation and deserialisation time, less type information has to be encoded in the serialised data, since it resides in the accompanying schema. The schemas themselves are written in a human readable way using JSON [Apa17a].

The schemas only allow for a few primitive and complex types, making them easily portable between programming languages. The primitive types include boolean values, 32 and 64 bit signed integers and floating point numbers, strings and byte arrays, as

[4]https://github.com/IPVS-DDS/maven-kafka-client-archetypes

well as the null-value (i.e. no value). For combining them into more complex structures, records (the equivalent to Java objects), enums and arrays are supported [Apa17c].

While not required, it is possible to automatically generate Java classes from Avro schemas using the `avro-maven-plugin` provided by the Apache Foundation [Apa17b]. The Confluent distribution of Kafka also includes a schema registry licenced under the Apache Licence. Basically, the schema registry offers a REST interface for accessing and storing Avro schemas [Apa17a]. It will be used in the implementation, since Confluent offers libraries simplifying the creation Serdes for objects described by Avro schemas when used in combination with a schema registry.

For an example schema which is used in this example, see listing A.1. The `namespace` attribute in the schema doubles as the package name of the generated Java class, whereas the `name` of the schema is used as the class name. `doc` attributes are only there for documentation and are reflected in the auto-generated Javadoc comments for the corresponding object attributes.

3.4. Kafka Producers

Kafka makes it pretty easy to write applications for writing data to topics. The only thing necessary is to create a `KafkaProducer<K, V>` and to initialise is with the URLs to the Kafka instances, a ID unique to all clients of the same type and the Serdes used for the keys and values of the messages.

After that, messages can simply be sent to topics by calling the `producer.send(record)` method, passing in a `ProducerRecord`, which holds information on the output topic as well as the record's key and value.

3.4.1. Device, Sensor and Sensor Type Registration

Registration information for devices, sensors and sensor types is stored in Kafka topics named `device-registration`, `sensor-registration` and `sensor-type-registration`, respectively. The stored messages contain metadata like the names and descriptions of the corresponding elements. Device, sensor and sensor type IDs are used as message keys.

The exact schemas used in the example are found in listing A.1 (device registration), listing A.2 (sensor registration) and listing A.3 (sensor type registration).

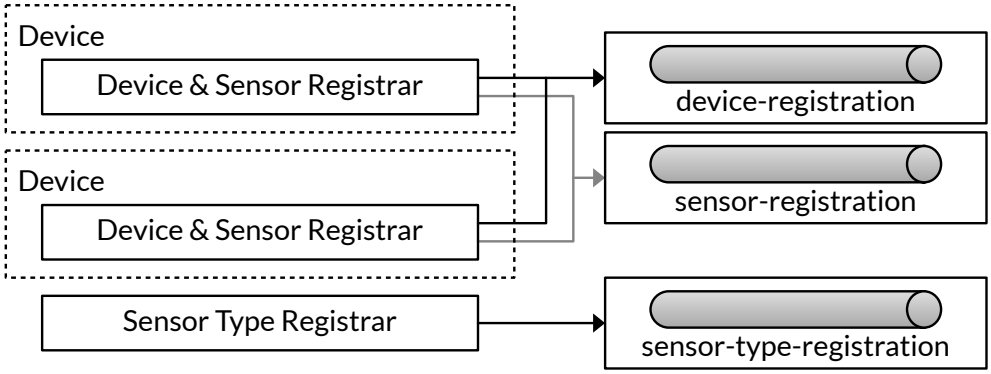

Figure 3.2.: Message queues and Kafka clients responsible for device registration

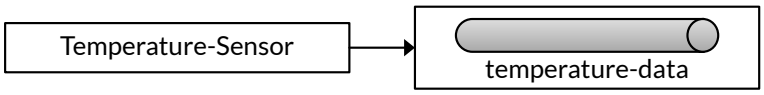

Figure 3.3.: Temperature sensor message flow

To simplify things, the registration of devices, sensors and sensor types is all handled by a single application, the object registrar[5]. It can be controlled using a number of parameters. The exact arguments to control the object registrar are documented in the project's `README.md` file, as well as the command line interface itself, where they are displayed when the application is launched without any parameters.

The object registrar then takes care of parsing the command line parameters, deciding which type of message has to be created (`DeviceRegistration`, `SensorRegistration` or `SensorTypeRegistration`), what values should be set for the messages and to which topic the messages should be sent (`device-registration`, `sensor-registration` or `sensor-type-registration`, respectively).

3.4.2. Device Simulation

The simulated device sensor acts as a Kafka producer, periodically generating `TemperatureData` messages, just as a real sensor would. Its message flow can be seen in fig. 3.3, its source code is available on GitHub[6]. Instead of measuring actual data, it generates data using the formula in eq. 3.1.

$$T_{sim}(t) = T_{base} + e^{(t_0 - t)/30000} * (T_{target} + T_{base}) + random(-1, 1) * noiseAmount \quad (3.1)$$

[5]Source code available at https://github.com/IPVS-DDS/object-registrar
[6]https://github.com/IPVS-DDS/temperature-sensor

The T_{base}, T_{target} and *noiseAmount* values can be configured through command line parameters passed to the `TemperatureSensor` Java application. The time t is the current Unix timestamp in milliseconds, t_0 is the Unix timestamp at the start of the program. Additionally, random temperature spikes are added to the temperature data, simulating devices running hotter than usual.

`TemperatureData` messages store the Unix timestamp at which the value was measured (or generated), the measured temperature and the unit the data was measured in. This can be one of `"C"` (degrees Celsius), `"K"` (Kelvin) or `"F"` (degrees Fahrenheit). The unit can also be set through the command line and defaults to `"C"`.

All messages are sent to the `temperature-data` topic of the Kafka cluster that was specified in the command line (defaulting to `localhost:9092`). A Schema Registry is used to create Serdes for the `TemperatureData`. Its URL can be given as the second parameter and defaults to `localhost:8081`.

3.5. Kafka Consumers

Kafka consumers are organised by Kafka in “consumer groups”. As the name suggests, consumer groups are used to organise consumers into groups. The Kafka cluster uses consumer groups to automatically balance the load on the individual instances of consumers within each group. Per group, Kafka assigns each of the topic’s partitions to one of the group members. If a consumer instance goes down, Kafka reassigns partitions that were previously assigned to that consumer to other consumers of the same group [Apa17e].

This architecture sets an upper limit to the number of consumers a topic can handle: since each partition is only assigned to a single consumer per group, there can not be more consumers active than there are partitions in the topic. Additional consumers would run idle.

3.5.1. Mirror Maker

For transferring data from topics on premise to the corresponding topics in the data centre, the Mirror Maker packaged with Kafka is used. The tool takes a configuration file for the input Kafka clusters and a configuration file for the target cluster, along with whitelist of topics to mirror from the source cluster to the target cluster.

The Mirror Maker as seen in fig. 3.4 is a relatively simple program: It creates a Kafka consumer for the input configuration, subscribing to all topics matching the whitelist. It

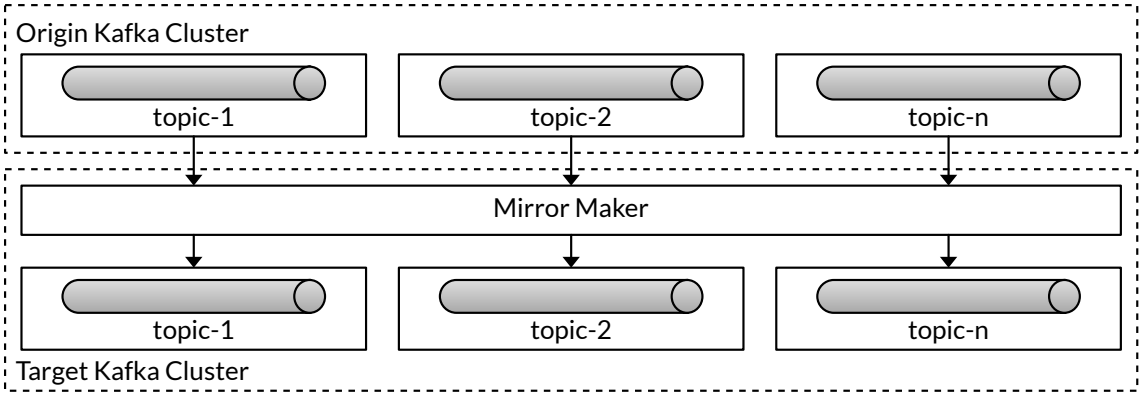

Figure 3.4.: Mirror maker message flow

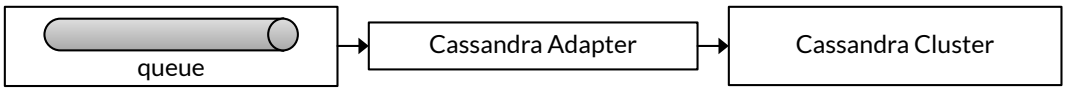

Figure 3.5.: Cassandra Adapter Message Flow

then spawns a Kafka producer for the target cluster. When a message is received on one of the input topics of the input cluster, the same message with the same key is then sent by the producer to the topic with the same name on the target cluster [Apa17f; Con17; Apa17l].

It should be noted that Confluent Inc offers a similar tool, called Kafka Connect Replicator, that can be used to replicate data of topics from multiple Kafka clusters to a target cluster, keeping the number of partitions and replicas of the source topics in sync with the target cluster as well as running transformations on the input data before it is written to the target cluster. Output topics can also have a different name than their input counterparts and it will detect and mirror whitelisted topics that were created after the Replicator process was first started. However, it is released under a proprietary licence and therefore not available without a support subscription [Con17; Jas17].

3.5.2. Cassandra Adapter

To inject data into the Cassandra Cluster, a simple consumer was developed. It takes data from either the `temperature-data` or `temperature-threshold-events` queues and sends an `INSERT` query to Cassandra for every record received as seen in fig. 3.5. Additionally to inserting all fields present in the original Avro messages, another field is added, carrying the difference between the current timestamp and the timestamp the message is carrying. This will later allow to perform rudimentary benchmarking of message transmission latencies from the devices generating the data to the adapter receiving

them and inserting them in the database. This is explained more elaborately in chapter 5. Its source code is available on GitHub[7].

3.6. Kafka Streams

The Kafka streams API further abstracts the consumer and producer APIs, providing a Domain Specific Language (DSL) for processing records.

Using the `StreamsBuilder` class, a stream topology can be created. The stream topology is a directed graph. Each edge of the graph is a "stream", which are used to pass data from one node of the topology to the next. Nodes act as "stream processors", reading data from zero or more incoming streams, possibly applying a transformation to them and writing data to zero or more outgoing streams. Stream processors without input streams are called "source processors", nodes without output streams are called "sink processors". Source processors read data from Kafka topics and produce input streams (acting as Kafka consumers), while sink processors write data from their input streams to Kafka topics (acting as Kafka producers). Stream processor nodes with ≥ 1 input and output nodes can split or join streams, or apply transformations to them. The transformations are well known from functional programming and include the operations `map`, `filter`, `reduce` and `aggregate` (also known as `fold` in the functional programming world) [Apa17j].

The `map` function can be used to perform an operation on all received records and returns a new stream, holding the results of the operation. Using the `filter` function, a predicate can be used to determine whether a certain record should be passed on further down the topology or if it should be filtered out. The most interesting operation is the `aggregate` function.

A simple example for the use of stream aggregation is counting how many records were received based on their keys (see fig. 3.6). To count the number of messages received bearing the same key, first there has to be a table mapping keys to the number of times they were observed. When a record is received, its key is looked up in the table. If the key could not be found, the value defaults to zero. Then, the value is incremented by one and gets written back to the table.

In Kafka Streams, stream aggregation makes use of the duality of streams and tables. Given a table of key-value pairs, every update to the table can be represented as a record in a stream, having its key set to the updated row's key and the value to the new value of

[7] https://github.com/IPVS-DDS/cassandra-adapter

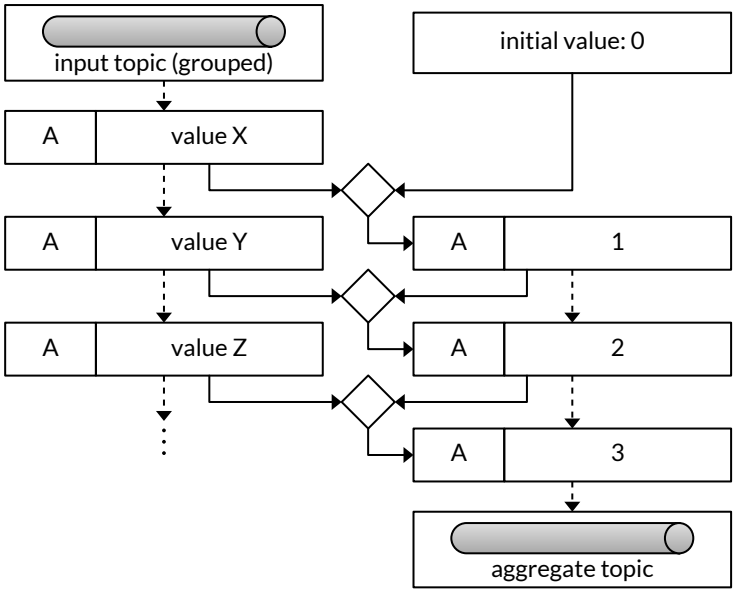

Figure 3.6.: Message aggregation using Kafka Streams

the row. The other direction of the relation is equally simple: given a stream of key-value pairs, a table can be built by simply overwriting the value of the row with the pair's key to the pair's value.

Kafka Streams have built-in support for this through the `KTable<K, V>` class. Basically, they read all entries of an entire topic, building up a key-value store that can be easily queried. They are especially important for aggregating streams in a fault tolerant way, since aggregation results are continuously written to the topic backing the `KTable` (often referred to as the table's "changelog"), making it possible to restore the stream's state, should the stream application crash.

The Kafka Streams client requires input streams to be grouped, to ensure proper partitioning of the data. This can easily be achieved by calling `groupByKey()` on the input stream or `groupBy(...)` when the input should not be grouped by the key but by some other property. Afterwards, aggregation can be performed on the grouped stream using the `.aggregate(...)` method, that takes an initial value for the aggregation and an aggregator, which provides a function that returns the new aggregation result, given the previous aggregation result and the received record. Stream aggregation is used in the example to generate events when a temperature threshold is exceeded, which include information about the average and maximum temperature during the transgression period.

The result of a stream aggregation is a `KTable` holding the aggregation results per grouped stream's group. By calling `toStream()` on the `KTable`, the table's changelog can be accessed and used for further operations.

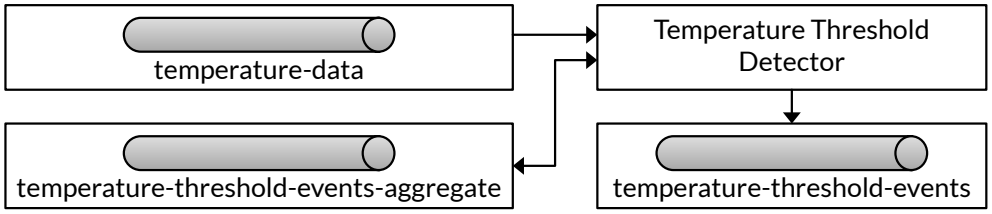

Figure 3.7.: Temperature threshold detector message flow

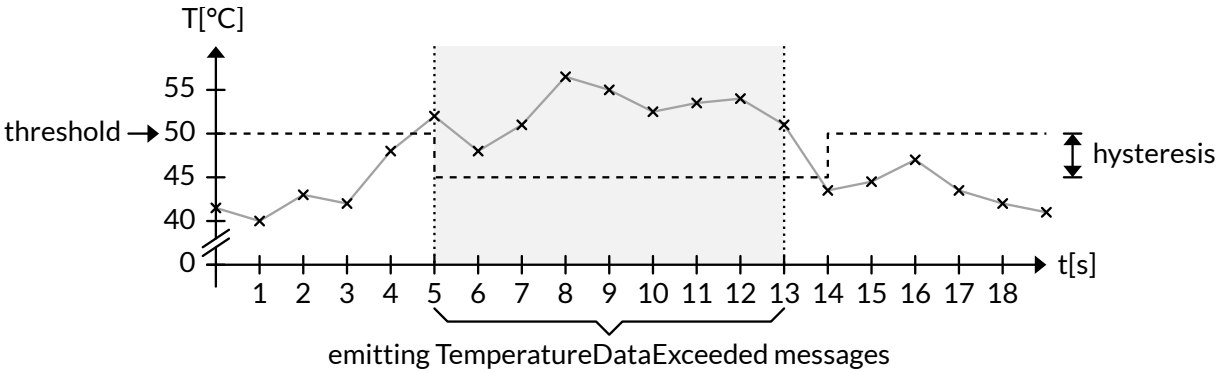

Figure 3.8.: Example of threshold messages being emitted

Additionally, `KStreams` and `KTables` (and combinations thereof) can be merged using the `join` operation. Joining a `KStream` with a `KTable` results in a new stream that emits values every time the input `KStream` receives a record and used the record's key to look up the current value for the same key in the `KTable`, returning a record with the same key and both the input value and the looked up value [Apa17i]. This will later be used to retrieve sensor and device metadata when sensor values are received.

3.6.1. Detecting Temperature Data Exceeding a given Threshold

The `TemperatureThresholdDetector`[8] reads data from the `temperature-data` topic and produces `TemperatureThresholdEvent` messages on the `temperature-threshold-events` topic when a sensor surpasses a given threshold, until the temperature drops below the threshold minus a hysteresis. For aggregating event data, the `temperature-threshold-events-topic` is used. The stream's message flow can be seen in fig. 3.7.

The `TemperatureThresholdEvents` contain information on the whole interval the temperature was over the threshold. These include the average temperature since exceeding the

[8]Source code available at https://github.com/IPVS-DDS/temperature-threshold-detector

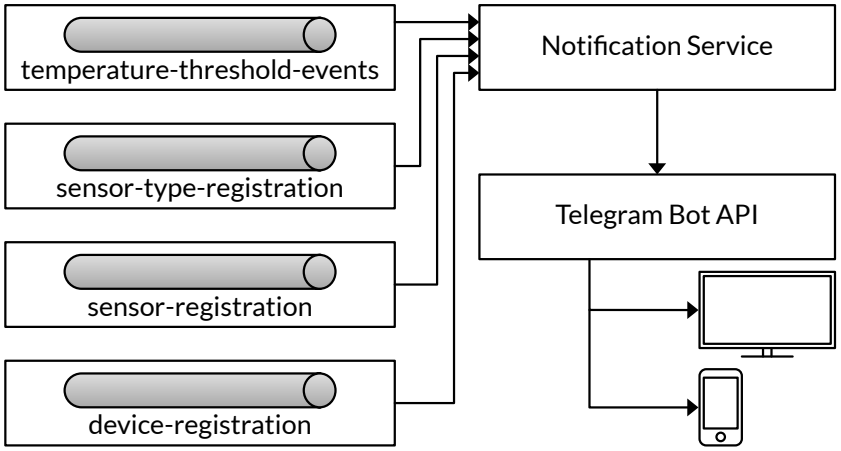

Figure 3.9.: Notification service message flow

threshold, the maximum temperature observed in the interval and how much time has passed since the interval was first exceeded.

Arriving temperature data is first converted to one of the supported temperature units, as configured through an application parameter (either `"C"`, `"F"` or `"K"`). The stream is then grouped by its key (the sensor ID) and aggregated. The aggregator checks if the temperature is currently exceeding the threshold. If the temperature is in the accepted bounds and also didn't exceed the threshold before or it previously exceeded the threshold but now dropped below *threshold* − *hysteresis*, the aggregate's `is_exceeding` value is set to false and all other values are reset. Otherwise, `is_exceeding` is set to true and the values for the timestamp of the first and most recent transgression, the latest transgressing temperature, the average and highest temperature during the transgression are set.

The resulting `KTable` is converted to a stream. Records not exceeding the threshold are discarded and the others are mapped to `TemperatureThresholdExceeded` records, removing metadata only required for the aggregation in the process. Finally, the `TemperatureThresholdExceeded` messages are sent to the `temperature-threshold-events` topic. Fig. 3.8 illustrates how the hysteresis works and for what input data `TemperatureThresholdExceeded` events are generated.

3.6.2. Triggering Messenger Notifications When Thresholds Are Exceeded

To put the `TemperatureThresholdExceeded` events to use, users can be notified of transgressing sensor values. Basically, the `NotificationService`[9] builds a message step by step. First, it is determined if the value transgressed for the first or last time. If not, the record is discarded, because the user will only be notified if the transgression state changed in order to not trigger too many notifications.

Then, the event (still holding a sensorID as its key) is joined with a `KTable` backed by the `sensor-registration` topic, yielding the sensor's name and the ID of associated device. The result is further joined with a `KTable` using the `device-registration` topic as its changelog to obtain the associated device's name and location. To make storing the device and sensor metadata more efficient and to not loose data when Kafka performs cleanups on the topic data, the cleanup policy for the registration topics is set to `compact`. This instructs Kafka to not simply delete all records older than the specified cleanup interval, but to retain at least the latest record for each key, ensuring that no registration information is lost.

Finally, the collected data is turned into a human readable message, stating the sensor and device name, the device's location and information on the transgression (including whether it exceeded the threshold for the first or last time, the total duration of the transgression and the average and maximum values during the period). The message is then sent to a Telegram chat through a Telegram bot, using the `java-telegram-bot-api`[10], passing in the bot's token and chatId as configured through the application's console parameters.

[9] Source code available at https://github.com/IPVS-DDS/notification-service
[10] https://github.com/pengrad/java-telegram-bot-api

4. Automatic Deployment of the Distributed Data Store

The example deployment for both the on premise and data centre servers consists of a cluster of Docker nodes configured as a swarm, running Apache ZooKeeper, Kafka, Schema Registry instances and the stream processing jobs described in chapter 3.

How the individual nodes and instances communicate is illustrated in fig. 4.1. There is one ZooKeeper, Kafka, schema registry and Cassandra instance per node, while stream processing jobs are automatically distributed by Docker. The ZooKeeper and Kafka instances form clusters and stream processing jobs each access the schema registry through `localhost`, requiring one instance per docker node.

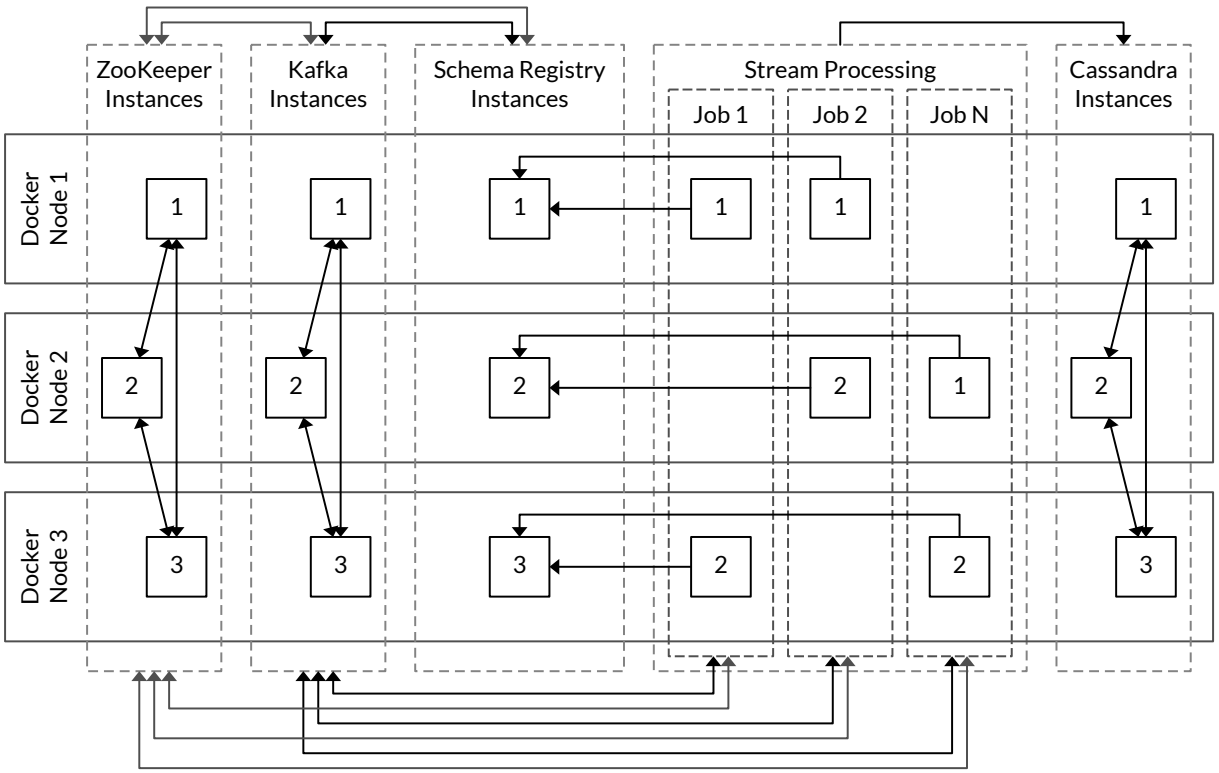

Figure 4.1.: Deployment Overview

4.1. Introduction to Docker

As written on the official Docker website, “The Docker platform is the only container platform to build, secure and manage the widest array of applications from development to production both on premises and in the cloud” [Doc17j].

Docker packages applications into containers, which contain all dependencies of an application and the application itself. Containers run isolated from each other and the operating system, which enables the containers to work on any system, regardless of host’s configuration, since the container contains with everything the application needs. In contrast to a full virtual machine, a container does not start up a whole guest operating system but runs on the active system’s kernel, making them more portable and efficient. “Containers and virtual machines have similar resource isolation and allocation benefits, but function differently because containers virtualize the operating system instead of hardware, containers are more portable and efficient” [Doc17i].

4.1.1. Installing Docker

To be able to run Docker in swarm mode, it first has to be installed on all nodes of the cluster. On Ubuntu 16.04LTS, this is achieved through running the commands in lst. 4.1.

```
1 sudo apt update
2 sudo apt install apt-transport-https ca-certificates curl software-properties-common
3 curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
4 sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb_release
      -cs) stable"
5 sudo apt update
6 sudo apt install docker-ce
```

Listing 4.1: Installing Docker [Doc17f]

For connecting to the Docker daemon, the user needs to be added to the `docker` group:

```
1 sudo usermod -a -G docker $USER
```

You’ll need to log out and back in to apply the changes.

4.1.2. Setting up a Docker Swarm

To create a swarm, the following command should be run on the chosen manager node:

```
1 docker swarm init --advertise-addr <manager-ip>
```

This initialises the swarm, assigning the current server as its manager. To add nodes to the swarm, the command displayed after running the command above has to be executed on the respective nodes.:

```
1 ssh user@node "docker swarm join --token <swarm-token> <manager-ip>:2377"
```

The join-command can also be retrieved by executing

```
1 docker swarm join-token worker
```

on one of the managers. The command for adding a node as a manager can be obtained by running

```
1 docker swarm join-token manager
```

To later promote a worker node to a manager node, one can use

```
1 docker swarm promote nodeName
```

For further information on managing Docker swarms, refer to [Doc17g].

All commands in the following sections for managing Docker services have to be run on a manager node of the swarm.

4.2. Deploying ZooKeeper and Kafka

The Docker images provided by Confluent, Inc are currently not officially supported in swarm mode. After some manual testing, the images work fine, as long as the host network is used instead of a virtual one provided by Docker.

Since ZooKeeper and Kafka instances each need a different configuration depending on the node they are running on, it is not sufficient to start them as a replicated Docker service. Instead, the deployment in this app creates one ZooKeeper and Kafka service *per node* of the cluster. To pin them to a single node each, a constraint is added restricting them to run only on a host with a given host name. This requires that each Kafka

(ZooKeeper) node of the cluster can find each other Kafka (ZooKeeper) node in the cluster by its host name. Therefore all host names have to be added to the `hosts`-file on every node, which is usually found at `/etc/hosts` on Linux [Man00].

To make deployment easier, a set of bash-scripts was created for starting and stopping ZooKeeper and Kafka services. All scripts can be found in a GitHub repository[1].

Before running the scripts, make sure to configure the host names in the `config.sh` file. Changing anything else should not usually be necessary.

The ZooKeeper instances can then be started by running `./start-ZooKeeper.sh`. After ZooKeeper is running, the Kafka cluster can be brought up by running `./start-kafka.sh`. The process may take a while when running the commands for the first time, since the Docker containers for both ZooKeeper and Kafka first need to be pulled from the official Docker registry on every node of the cluster before they are started.

For stopping the services, run `./stop-kafka.sh`, followed by `./stop-ZooKeeper.sh` to ensure that the Kafka instances all shut down correctly.

4.3. Deploying the Schema Registry

After the ZooKeeper and Kafka clusters have started and before starting the schema registry, the `_schema` topic first has to be created with the correct configuration by executing the command in lst. 4.2. The topic is special in the way that it *must* have only one partition and *must* have its cleanup policy set to `compact`, instructing Kafka to run log compaction[2] when the topic reaches its size limit.

```
1 ./kafka-topics.sh \
2   --create \
3   --topic _schemas \
4   --partitions 1 \
5   --replication-factor 3 \
6   --config cleanup.policy=compact
```

Listing 4.2: Creating the `_schema` Topic for the Schema Registry

[1] https://github.com/IPVS-DDS/deployment-scripts
[2] https://kafka.apache.org/0110/documentation/#compaction

4.4. Deploying Cassandra

Thanks to the architecture of Apache Cassandra, setting it up is extremely easy. The only configuration required is a list of IPs of where Cassandra nodes can be found. Using a Docker compose file, the cluster can be started by running lst. 4.3 on any Docker swarm manager node.

```
1 docker stack deploy --compose-file=compose/cassandra.yml cassandra
```

Listing 4.3: Deploying the Apache Cassandra Cluster

The Docker compose file assumes that the list of IPs is set as an environment variable called `CASSANDRA_IPS` when issuing the `stack deploy` command. The environment variable will automatically be set when running lst. 4.4.

```
1 . compose/config.sh
```

Listing 4.4: Sourcing Cluster Configuration Environment Variables

Hostnames of the Cassandra nodes can be specified in the `compose/config.sh` file before issuing the commands above.

4.5. Deploying Custom Kafka Clients

In order to deploy locally built Docker images to the cluster, a private Docker registry is launched.

4.5.1. Setting up a Private Docker Registry

To be able to pull the custom images, all nodes in the cluster must have access to a shared Docker registry. If a publicly available registry such as the Docker Hub[3] is available, this step can be skipped. For a private deployment though, setting up a private Docker registry can have a number of advantages, since the images pushed to the registry are not publicly available, only accessible from inside the cluster's network and self-hosted registries are completely free of charge.

[3] https://hub.docker.com

For launching the registry, refer to lst. 4.5. Additional configuration, such as attaching a persistent storage device, may be necessary for production deployments [Doc17b; Doc17a].

```
1 docker service create --name docker-registry --publish 5000:5000 registry:2.6.2
```

Listing 4.5: Creating a Docker registry service [Doc17c]

After the registry is launched, every member of the Docker swarm can access it on `localhost:5000`, thanks to Docker's mesh network [Doc17h].

4.5.2. Pushing Local Images to the Private Docker Registry

For the swarm notes to be able to run custom Docker images, the images first have to be pushed to the Docker registry. In case of using a private registry, this can be achieved by opening an SSH tunnel that makes the remote registry available on the local machine using the command in lst. 4.6. The `nodeIp` has to be replaced with the IP or host name of any of the swarm members.

```
1 ssh -N -L 5000:localhost:5000 nodeIp
```

Listing 4.6: Opening an SSH Tunnel to the Docker Registry

After the tunnel is open, the local image first has to be tagged with the registry's url and then can be pushed to the registry, using the commands in lst. 4.7. Instead of manually tagging the images, the maven `pom.xml` files of the client projects could be modified to directly name the images with the `localhost:5000/` prefix, but this requires the SSH tunnel to be open when packaging the projects.

```
1 docker tag {,localhost:5000/}imageName:version
2 # Equivalent to running
3 # docker tag imageName:version localhost:5000/imageName:version
4 docker push localhost:5000/imageName:version
```

Listing 4.7: Tagging and Pushing a Docker Image [Doc17e; Doc17d]

Alternatively, for pushing all images tagged with the private registry used in a `docker-compose`-file (as provided in the deployment repository[4]), lst. 4.8 can be run. This requires the `docker-compose` command to be installed. For installation instructions, refer to the official documentation[5].

[4] https://github.com/IPVS-DDS/deployment-scripts/tree/master/compose
[5] https://docs.docker.com/compose/install/

```
1 docker-compose push
```

Listing 4.8: Pushing a Docker Stack to the Private Registry

4.5.3. Starting Custom Kafka Clients

After the local images for all custom clients are pushed to the registries on premise and the data centre, the clients can be started. Using the provided compose-files, deploying the clients of the example is as easy as running lst. 4.9 on a swarm manager node on the on premise clusters and lst. 4.10 on a manager node of the data center form inside the deployment repository's root directory.

```
1 docker stack deploy --compose-file=compose/on-premise.yml dds
```

Listing 4.9: Deploying the Example Clients on Premise

```
1 docker stack deploy --compose-file=compose/data-centre.yml dds
```

Listing 4.10: Deploying the Example Clients to the Data Centre

Since the temperature sensor simulator is neither part of the on premise cluster nor the central data store, it has to be run individually as seen in lst. 4.11, replacing the placeholders in brackets with actual values.

```
1 docker run -it --network host --rm \
2   localhost:5000/dds/temperature-sensor:1.0-SNAPSHOT \
3   [hostname]:9092 http://localhost:8081 \
4   [sensor-name (string)] \
5   [start-temperature (double)] \
6   [target-temperature (double)] \
7   [C|K|F] \
8   [noise amount (double)] \
9   [ms between sending messages (int)]
```

Listing 4.11: Running a Temperature Sensor

4.5.4. Starting Mirror Maker instances

Kafka's Mirror Maker is responsible for copying data from topics on the local deployments to the central data store. It also copies schema data of the schema registry to the central cluster, ensuring that the same schemas are used on both the data centre and on premise,

preventing conflicts when trying to decode Avro messages. Since no ready made and up-to-date Docker container could be found that packaged the Mirror Maker in a way to be easily configured through environment variables passed to Docker, an outdated solution was forked on GitHub and adapted to use Confluent's `confluentinc/cp-kafka:3.3.0` image as base and the `kafka-mirror-maker` command that is comes pre-packaged in the container. Since the Mirror Maker requires the use of property files for configuration, it was not possible to just use the Kafka image provided by Confluent. Instead, a single layer is added to the Docker container, introducing a wrapper script that extracts configuration values from environment variables passed to the running container by Docker.

The full source code can be found in its GitHub repository[6]. For building the Mirror Maker container and pushing it to the central data store's Docker registry, simply run lst. 4.12 from inside the project's root directory.

```
1 docker build -t localhost:5000/dds/mirror-maker .
2 docker push localhost:5000/dds/mirror-maker
```

Listing 4.12: Building and Pushing the Mirror Maker

To finally run the Mirror Maker as a service on the central data store, run lst. 4.13 after sourcing the configuration file as seen in lst. 4.4.

```
1 docker stack deploy --compose-file=compose/mirror-maker.yml mirror-maker
```

Listing 4.13: Deploying the Mirror Maker

To make deployment of the whole implementation even easier, two scripts are provided in the GitHub repository[7]: `init-op-cluster.sh` and `init-dc-cluster.sh`, which are executing all steps required (apart from the Docker installation, the configuration of host names and setup of the Docker registry including pushing the required images to the server) in order to set up the complete on premise cluster or data centre cluster, respectively.

[6] https://github.com/IPVS-DDS/docker-kafka-mirrormaker
[7] https://github.com/IPVS-DDS/deployment-scripts

5. Evaluation of the Distributed Data Store

The example implementation was deployed to two clusters hosted on servers provided by the University of Stuttgart. The actual servers used were virtual machines hosted with OpenStack[1]. Due to quota constrains, the servers were restricted to instances of the `m1.medium` category, featuring two VCPUs and 4GB of RAM each. This makes for a rather humble deployment, but is enough to show that the proposed concept is indeed feasible to be used for transmitting and collecting IoT data both on premise and at a central location.

For performing the actual benchmarking, temperature sensor application instances were ran in parallel on a desktop computer (Intel Core i7 6770K, 16GB DDR4 RAM). During initial tests it quickly became apparent that the desktop computer on its own was capable of producing more events than the "on premise" cluster of three VMs could handle. Therefore, the sensor data producers were ran on the computer and not the cluster, to have more resources to spare for performing the actual task of the deployment. However, this added additional latency of about 9.5ms on average (round trip time obtained using the `ping` command, divided by two), since all data had to pass through the internet and the university's VPN. The client to server latency was subtracted from the final results.

Further, latency was only able to be measured for the time taken between generating the original data until generating the database query, since the time from executing the query until the data is available in the database was not able to be reliably recorded. Because Cassandra only guarantees *eventual* consistency, it is hard to actually make predictions about the time it takes to completely persist the data. Therefore, this time was not considered in the performed benchmarks.

Additionally, the latency for the complete message path could only be measured by recording the timestamp when sending the data and recording a second timestamp when sending the data to the database. Because it was not possible to properly synchronise all the system clocks involved in the process, the recorded latencies are potentially skewed.

[1] https://www.openstack.org/

5.1. Benchmarking Results

The benchmark results show that the implemented architecture works well for smaller deployments up to about 10.000 messages per second with latencies of about 4ms until the data arrives at the edge database and 67ms until the data makes it to the central database. Exceeding this limit, message latencies start to grow rapidly. It should be noted that the messages still continue to pile up in the Kafka topics, suggesting that the actual bottleneck in the system is the database adapter. Due to time constraints, it was not possible to investigate this issue any further or to collect more detailed results.

CPU utilisation during the tests hovered around 90% on all nodes.

6. Conclusion and Outlook

The thesis proposed a concept for analysing and store IoT data distributed among multiple edge clusters and a cluster located at the centre of the network, collecting data from all the clusters on premise. At its heart, the concept carries a message queuing system which responsible for connecting data sources, processing jobs and databases. The same message queuing system is used to transmit selected device data from the data stores on premise to the central data store.

The overall concept is held intentionally broad, in order to allow for use case specific implementations, following the same overall guidelines. An example use case was introduced and a concrete solution was implemented, making use of Apache Kafka as its messaging system and Apache Cassandra for persistently storing the processed data, where it can be easily accessed for further analysis. A number of Kafka clients were developed to demonstrate the capabilities of the Architecture, from ingesting data into the system, over analysing it as it arrives, to sending messages to uses through an external messenger. Deployments scripts and hints are provided, which aid in setting up a similar system on other clusters with minimal required configuration.

The message latencies of the implementation were evaluated, revealing that on a small cluster, the system is capable of handling about 10.000 messages per second without breaking down. This is may not be enough for a larger real world application, but such a deployment would probably sport more capable servers, making the architecture viable to be the backbone for most if not all data processing needs.

The findings in chapter 5, Evaluation of the Distributed Data Store suggest that the source of the rapid buildup of latency for messages to arrive at the database may lie in the Cassandra adapter, which takes the messages from Kafka and inserts them into Database. This issue would have to be further investigated before making a final decision if the implementation is feasible for a more realistic use case.

6.1. Future Work

In its current state, the example implementation can not directly be deployed to a production environment. Most importantly, it lacks support for message encryption. When transmitting device messages over the internet, either a VPN would have to be used to connect the clusters with each other or security features supported by Kafka would have to be employed. These include Transport Layer Security (TLS) and authentication for Kafka and ZooKeeper communication [Ism17].

Additionally persistent storage volumes would need to be attached to the Docker deployments, or all application data is lost forever when the ZooKeeper, Kafka or Cassandra instances are stopped.

The implemented architecture could further be extended by connecting it to the Resource Management Platform. Instead of in topics, device and sensor registration information could be stored and accessed through the RMP. Adapters would have to be written that automatically subscribe to newly added sensor data queues and send the sensor data to the correct topic for the corresponding sensor type.

Furthermore, the data of local RMPs would need to be made available to one central RMP located at the central data store. This could be realised by introducing the notion of RMP hierarchies. Local RMPs could send device registration changes through a specific Kafka topic to the central RMP, where they are read and applied to the central state.

Other storage solutions could also be investigated. Druid[1], for example, is specially geared towards storing and helping to analyse time series data, but is fairly complex to set up. Another potential storage system for and analysing IoT data could be Apache Hadoop[2], which implements a MapReduce approach, allowing to process huge datasets in parallel.

[1] http://druid.io
[2] https://hadoop.apache.org/

A. Appendix

A.1. Avro Schemas

A.1.1. Device Registration

```
{
  "namespace": "de.unistuttgart.ipvs.dds.avro",
  "type": "record",
  "name": "DeviceRegistration",
  "doc": "A message sent to register devices or update device information. Use the device's ID as message
      key.",
  "fields": [
    {
      "name": "name",
      "type": "string",
      "doc": "The display name of the device."
    },
    {
      "name": "description",
      "type": "string",
      "default": "",
      "doc": "A more detailed description of the device and its purpose."
    },
    {
      "name": "location",
      "type": "string",
      "default": "",
      "doc": "A description of where the device is located at."
    }
  ]
}
```

Listing A.1: Device Registration Avro Schema, 'device-registration.avsc'

A.1.2. Sensor Registration

```
{
  "namespace": "de.unistuttgart.ipvs.dds.avro",
  "type": "record",
  "name": "SensorRegistration",
  "doc": "A message sent to register sensors or update sensor information. Use the sensor's ID as message
      key.",
  "fields": [
    {
      "name": "name",
      "type": "string",
      "doc": "The display name of the sensor."
    },
    {
      "name": "description",
      "type": "string",
      "default": "",
      "doc": "A more detailed description of the sensor and its purpose."
    },
    {
      "name": "device_id",
      "type": "string",
      "doc": "The id of the device the sensor belongs to."
    },
    {
      "name": "sensor_type_id",
      "type": "string",
      "doc": "The id of the sensor's type."
    }
  ]
}
```

Listing A.2: Sensor Registration Avro Schema, 'sensor-registration.avsc'

A.1.3. Sensor Type Registration

```
{
  "namespace": "de.unistuttgart.ipvs.dds.avro",
  "type": "record",
  "name": "SensorTypeRegistration",
  "doc": "A message sent to register sensor types or update sensor type information. Use the sensor type's
      ID as message key.",
  "fields": [
    {
      "name": "name",
      "type": "string",
      "doc": "The display name of the sensor type."
```

```
    },
    {
      "name": "description",
      "type": "string",
      "default": "",
      "doc": "A more detailed description of the sensor type and its purpose."
    },
    {
      "name": "unit",
      "type": "string",
      "default": "",
      "doc": "The unit the sensor is measuring data in."
    }
  ]
}
```

Listing A.3: Sensor Type Registration Avro Schema, 'sensor-type-registration.avsc'

A.1.4. Temperature Data

```
{
  "namespace": "de.unistuttgart.ipvs.dds.avro",
  "type": "record",
  "name": "TemperatureData",
  "doc": "Message sent by sensors recording temperature data.",
  "fields": [
    {
      "name": "sensor_id",
      "type": "string",
      "doc": "The unique ID of the sensor sending the data."
    },
    {
      "name": "timestamp",
      "type": "long",
      "default": 0,
      "doc": "The unix timestamp at which the data was recorded."
    },
    {
      "name": "temperature",
      "type": "double",
      "doc": "The value of the temperature data."
    },
    {
      "name": "unit",
      "type": {
        "name": "TemperatureUnit",
        "type": "enum",
```

```
28        "symbols": [ "C", "F", "K" ]
29      },
30      "doc": "Whether the temperature was recorded in degrees celsius, degrees Fahrenheit or kelvin."
31    }
32  ]
33 }
```

Listing A.4: Temperature Data Avro Schema, 'temperature-data.avsc'

A.1.5. Temperature Threshold Events

```
1  {
2    "namespace": "de.unistuttgart.ipvs.dds.avro",
3    "type": "record",
4    "name": "TemperatureThresholdExceeded",
5    "doc": "Message sent when a the TemperatureThreshold stream processing job detects that a temperature
         value exceeds a certain threshold.",
6    "fields": [
7      {
8        "name": "timestamp",
9        "type": "long",
10       "default": 0,
11       "doc": "The unix timestamp at which the threshold was last exceeded."
12     },
13     {
14       "name": "temperature_threshold",
15       "type": "double",
16       "doc": "The threshold that was exceeded."
17     },
18     {
19       "name": "average_transgression",
20       "type": "double",
21       "doc": "The average temperature since exceeding the threshold."
22     },
23     {
24       "name": "max_transgression",
25       "type": "double",
26       "doc": "The maximum temperature since exceeding the threshold."
27     },
28     {
29       "name": "exceeded_for_ms",
30       "type": "long",
31       "doc": "The time in ms since the threshold first was exceeded."
32     }
33   ]
34 }
```

Listing A.5: Temperature Threshold Exceeded Avro Schema, 'temperature-threshold-exceeded.avsc'

A.1.6. Temperature Threshold Aggregation

```
1  {
2    "namespace": "de.unistuttgart.ipvs.dds.avro",
3    "type": "record",
4    "name": "TemperatureThresholdAggregate",
5    "doc": "Message sent when a the TemperatureThreshold stream processing job detects that a temperature
         value exceeds a certain threshold.",
6    "fields": [
7      {
8        "name": "is_exceeding",
9        "type": "boolean",
10       "doc": "Whether the value currently exceeds the threshold or not."
11     },
12     {
13       "name": "is_first",
14       "type": "boolean",
15       "doc": "Whether this is the first aggregate since a switch from not exceeding to exceeding or vice
             versa."
16     },
17     {
18       "name": "first_transgression",
19       "type": "long",
20       "doc": "The unix timestamp at which the threshold was first exceeded."
21     },
22     {
23       "name": "latest_transgression",
24       "type": "long",
25       "doc": "The unix timestamp at which the threshold was last exceeded."
26     },
27     {
28       "name": "latest_transgression_temperature",
29       "type": "double",
30       "doc": "The temperature at the latest transgression"
31     },
32     {
33       "name": "average_transgression_temperature",
34       "type": "double",
35       "doc": "The average temperature since exceeding the threshold."
36     },
37     {
38       "name": "max_transgression_temperature",
```

```
39      "type": "double",
40      "doc": "The maximum temperature since exceeding the threshold."
41    }
42  ]
43 }
```

Listing A.6: Temperature Threshold Aggregate Avro Schema, 'temperature-threshold-aggregate.avsc'

Bibliography

[Apa15a] Apache Software Foundation. *Apache Storm - Concepts*. 2015. URL: http://storm.apache.org/releases/1.1.1/Concepts.html (cit. on p. 25).

[Apa15b] Apache Software Foundation. *Apache Storm - Setting up a Storm Cluster*. 2015. URL: http://storm.apache.org/releases/1.1.1/Setting-up-a-Storm-cluster.html (cit. on p. 26).

[Apa17a] Apache Software Foundation. *Apache Avro™ 1.8.2 Documentation*. 2017. URL: http://avro.apache.org/docs/1.8.2/ (cit. on pp. 29, 30).

[Apa17b] Apache Software Foundation. *Apache Avro™ 1.8.2 Getting Started (Java)*. 2017. URL: https://avro.apache.org/docs/1.8.2/gettingstartedjava.html (cit. on p. 30).

[Apa17c] Apache Software Foundation. *Apache Avro™ 1.8.2 Specification*. 2017. URL: http://avro.apache.org/docs/1.8.2/spec.html (cit. on p. 30).

[Apa17d] Apache Software Foundation. *Apache Kafka - Documentation*. 2017. URL: https://kafka.apache.org/documentation.html (cit. on p. 25).

[Apa17e] Apache Software Foundation. *Apache Kafka - Documentation - Consumers*. 2017. URL: https://kafka.apache.org/documentation.html#intro_consumers (cit. on p. 32).

[Apa17f] Apache Software Foundation. *Apache Kafka - Documentation - Mirroring data between clusters*. 2017. URL: https://kafka.apache.org/documentation.html#basic_ops_mirror_maker (cit. on p. 33).

[Apa17g] Apache Software Foundation. *apache/kafka: Mirror of Apache Kafka*. 2017. URL: https://github.com/apache/kafka (cit. on p. 28).

[Apa17h] Apache Software Foundation. *Clients - Apache Kafka - Apache Software Foundation*. 2017. URL: https://cwiki.apache.org/confluence/display/KAFKA/Clients (cit. on p. 28).

[Apa17i] Apache Software Foundation. *Developer Guide for Kafka Streams API*. 2017. URL: https://kafka.apache.org/0110/documentation/streams/developer-guide (cit. on p. 36).

[Apa17j] Apache Software Foundation. *Kafka Streams API - Core Concepts*. 2017. URL: https://kafka.apache.org/0110/documentation/streams/core-concepts (cit. on p. 34).

[Apa17k] Apache Software Foundation. *Maven - Introduction*. 2017. URL: https://maven.apache.org/what-is-maven.html (cit. on p. 28).

[Apa17l] Apache Software Foundation. *Source Code of MirrorMaker.scala*. 2017. URL: https://github.com/apache/kafka/blob/0.11.0/core/src/main/scala/kafka/tools/MirrorMaker.scala (cit. on p. 33).

[Con17] Confluent Inc. *Apache Kafka's Mirror Maker*. 2017. URL: https://docs.confluent.io/current/multi-dc/mirrormaker.html (cit. on p. 33).

[Doc17a] Docker, Inc. *Configuring a registry | Docker Documentation*. 2017. URL: https://docs.docker.com/registry/configuration/ (cit. on p. 44).

[Doc17b] Docker, Inc. *Deploy a registry server | Docker Documentation*. 2017. URL: https://docs.docker.com/registry/deploying/ (cit. on p. 44).

[Doc17c] Docker, Inc. *Deploy a stack to a swarm | Docker Documentation*. 2017. URL: https://docs.docker.com/engine/swarm/stack-deploy/ (cit. on p. 44).

[Doc17d] Docker, Inc. *docker push | Docker Documentation*. 2017. URL: https://docs.docker.com/engine/reference/commandline/push/ (cit. on p. 44).

[Doc17e] Docker, Inc. *docker tag | Docker Documentation*. 2017. URL: https://docs.docker.com/engine/reference/commandline/tag/ (cit. on p. 44).

[Doc17f] Docker, Inc. *Get Docker CE for Ubuntu*. 2017. URL: https://docs.docker.com/engine/installation/linux/docker-ce/ubuntu/#install-using-the-repository (cit. on p. 40).

[Doc17g] Docker, Inc. *Swam mode overview | Docker Documentation*. 2017. URL: https://docs.docker.com/engine/swarm/ (cit. on p. 41).

[Doc17h] Docker, Inc. *Use swarm mode routing mesh | Docker Documentation*. 2017. URL: https://docs.docker.com/engine/ingress/ (cit. on p. 44).

[Doc17i] Docker, Inc. *What is a Container | Docker*. 2017. URL: https://www.docker.com/what-container (cit. on p. 40).

[Doc17j] Docker, Inc. *What is Docker?* 2017. URL: https://www.docker.com/what-docker (cit. on p. 40).

[DSA17] DSA Initiative. *Open Source IoT Platform & Toolkit*. 2017. URL: http://iot-dsa.org/get-started/how-dsa-works (cit. on pp. 14, 15).

[HWBM16a] P. Hirmer, M. Wieland, U. Breitenbücher, B. Mitschang. “Automated Sensor Registration, Binding and Sensor Data Provisioning.” In: *CAiSE Forum*. 2016, pp. 81–88 (cit. on pp. 12, 15).

[HWBM16b] P. Hirmer, M. Wieland, U. Breitenbücher, B. Mitschang. “Dynamic ontology-based sensor binding.” In: *East European Conference on Advances in Databases and Information Systems*. Springer. 2016, pp. 323–337 (cit. on pp. 12, 15).

[Ism17] Ismael Juma, Confluent Inc. *Apache Kafka Security 101 - Confluent*. 2017. URL: https://www.confluent.io/blog/apache-kafka-security-authorization-authentication-encryption/ (cit. on p. 50).

[Jas17] Jason Gustafson; Confluent Inc. *Kafka Connect Replicator*. 2017. URL: https://docs.confluent.io/current/connect/connect-replicator/docs/connect_replicator.html (cit. on p. 33).

[Jay16] Jay Kreps, Confluent Inc. *Introducing Kafka Streams: Stream Processing Made Simple - Confluent*. 2016. URL: https://www.confluent.io/blog/introducing-kafka-streams-stream-processing-made-simple/ (cit. on p. 26).

[Man00] Manoj Srivastava. *hosts(5) - Linux manual page*. 2000. URL: http://man7.org/linux/man-pages/man5/hosts.5.html (cit. on p. 42).

[Mar17] MariaDB. *Multi-source Replication - MariaDB Knowledge Base*. 2017. URL: https://mariadb.com/kb/en/library/multi-source-replication/ (cit. on p. 14).

[Piv17a] Pivotal Software, Inc. *RabbitMQ - RabbitMQ Tutorial - Publish/Subscribe*. 2017. URL: https://www.rabbitmq.com/tutorials/tutorial-three-python.html (cit. on p. 23).

[Piv17b] Pivotal Software, Inc. *RabbitMQ - RabbitMQ Tutorial - Routing*. 2017. URL: https://www.rabbitmq.com/tutorials/tutorial-four-python.html (cit. on p. 23).

[Piv17c] Pivotal Software, Inc. *RabbitMQ - RabbitMQ Tutorial - Topics*. 2017. URL: https://www.rabbitmq.com/tutorials/tutorial-five-python.html (cit. on p. 24).

[Piv17d] Pivotal Software, Inc. *RabbitMQ - Which protocols does RabbitMQ support?* 2017. URL: https://www.rabbitmq.com/protocols.html (cit. on p. 23).

[Piv17e] Pivotal Software, Inc. *RabbitMQ - Which protocols does RabbitMQ support?* 2017. URL: https://www.rabbitmq.com/tutorials/tutorial-two-python.html (cit. on p. 23).

[Pos17a] PostgreSQL Global Development Group. *PostgreSQL: About*. 2017. URL: https://www.postgresql.org/about/ (cit. on p. 27).

[Pos17b] PostgreSQL Global Development Group. *PostgreSQL: Documentation: 10: 26.1. Comparison of Different Solutions*. 2017. URL: https://www.postgresql.org/docs/current/static/different-replication-solutions.html (cit. on p. 14).

[Pos17c] PostgreSQL Global Development Group. *PostgreSQL: Documentation: 10: 9.15. JSON Functions and Operators*. 2017. URL: https://www.postgresql.org/docs/10.0/static/functions-json.html (cit. on p. 27).

[Pos17d] PostgreSQL Global Development Group. *PostgreSQL: Documentation: 10: H.1. Client Interfaces*. 2017. URL: https://www.postgresql.org/docs/current/static/external-interfaces.html (cit. on p. 27).

[Pos17e] PostgreSQL Global Development Group. *PostgreSQL: Documentation: 9.2: JSON Functions*. 2017. URL: https://www.postgresql.org/docs/9.2/static/functions-json.html (cit. on p. 27).

[Pos17f] PostgreSQL Global Development Group. *PostgreSQL: The world's most advanced open source database*. 2017. URL: https://www.postgresql.org/ (cit. on p. 27).

[Sat17] SataStax. *A Brief Introduction to Apache Cassandra*. 2017. URL: https://academy.datastax.com/resources/brief-introduction-apache-cassandra (cit. on p. 28).

All links were last followed on September 19, 2017.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

Ort, Datum, Unterschrift

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift