Technische Universität München
Department of Informatics
Chair for Network Architectures and Services

Privacy-Friendly Energy Monitoring

Master's Thesis in Informatics

accomplished at
Chair for Network Architectures and Services
Department of Informatics
Technische Universität München

by
Benedikt Peter

January 2015

Technische Universität München
Department of Informatics
Chair for Network Architectures and Services

Privacy-Friendly Energy Monitoring

—

Privacy-freundliches Energie-Monitoring

Master's Thesis

accomplished at
Chair for Network Architectures and Services
Department of Informatics
Technische Universität München

by
Benedikt Peter

Supervisor:	Prof. Dr.-Ing. Georg Carle
Advisors:	Dr. Holger Kinkelin
	Marcel von Maltitz, M.Sc.
Submission date:	01/15/2015

I confirm that this Master's Thesis is my own work and I have documented all sources and material used.

Abstract:

With the establishment of Smart Buildings and the possibility to measure the energy consumption of an entire building in real-time, automatic energy monitoring on a large scale has become an opportunity. For example, by monitoring the energy of individual rooms, possible power eaters can be identified and the consumption can be optimized accordingly.

However, when collecting measurement values typical for buildings, like energy consumption, this data may contain information that could be used to observe the behaviour of persons living or working in that building. Violating the privacy of subjects in such a way does not only present a major threat to the individual's desire for privacy, but is also against established law in many countries, like, for example, Germany. Thus, when designing an architecture for data processing, the preservation of the individual's privacy has to be a major goal.

This work is aimed at designing such an architecture for privacy-friendly collecting, processing and storing of data. To do this, encryption and storage schemes are designed that allow the reduction and hiding of privacy-critical information from plain view and the enforcement of access restrictions in a way preventing any unauthorized person to ever gain access to information violating the individual's privacy. The design and implementation considerations of the implemented *Privacy-Preserving Post-Processing and Storage* (P4S) architecture are demonstrated in detail and with special focus on the features implementing the privacy preservation.

The evaluation of the implementation shows that the architecture does well when hiding individual information, assumed a properly configured and secured environment is provided. Moreover, additional tests for correctness, stability and performance show that the architecture is ready-to-use in a real-world environment.

Kurzfassung:

Aus der Etablierung von Smart Buildings folgt eine Reihe von Möglichkeiten zur Messung und Auswertung von Größen wie dem Energieverbrauch in Echtzeit. Beispielsweise kann der Energieverbrauch ganzer Gebäude oder größerer Umgebungen automatisch erfasst werden. Durch die Überwachung von einzelnen Räumen kann etwa Energieverschwendung entdeckt und darauf reagiert werden.

Durch diese feinkörnigen Messungen können jedoch Daten anfallen, die einen detaillierten Aufschluss über das Verhalten von individuellen Personen geben, die in einem solchen Smart Building leben oder arbeiten. Derartige Überwachung stellt nicht nur eine Bedrohung für die individuellen Personen und ihr Bedürfnis nach Privatsphäre dar, sondern verstößt zudem in Deutschland und vielen anderen Ländern auch gegen geltendes Recht. Daher muss bei der Entwicklung einer Architektur zur Speicherung solcher Messdaten besonderes Augenmerk auf die Achtung der Privatsphäre gelegt werden.

Diese Arbeit behandelt den Entwurf und die Implementierung einer solchen Architektur für das *privacy*-freundliche Sammeln, Verarbeiten und Speichern von Messdaten. Dazu werden Schemata zur Verschlüsselung und Speicherung entworfen, die es ermöglichen, *privacy*-kritische Informationen in den Datenströmen gezielt zu verbergen, und zu verhindern, dass (unberechtigte) Personen Einblick in die feinkörnigen Daten erhalten. Die Ideen hinter dem Entwurf und der Implementierung dieser *Privacy-Preserving Post-Processing and Storage* (P4S) Architektur werden dazu detailliert dargelegt.

Die Evaluierung dieser Implementierung zeigt, dass die Architektur gut dafür geeignet ist, persönliche Informationen zu verbergen. Voraussetzung hierfür ist jedoch, dass die Architektur entsprechend konfiguriert ist und in einer sicheren Umgebung ausgeführt wird. Zusätzliche Tests bezüglich Korrektheit, Stabilität und Performance zeigen, dass einem Einsatz in einer wirklichen Umgebung nichts mehr im Wege steht.

Contents

1. Introduction

With climate change being a major topic and the *Energiewende* (energy turnaround) being realized in Germany, further efforts have to be taken on saving energy. But because there usually is no direct feedback for measures taken and only the invoice at the end of the accounting period gives a coarse impression on the energy spent or saved, for many people energy consumption remains an abstract matter and, therefore, often nothing is actually done to improve energy efficiency.

To provide individuals with immediate feedback and, hence, being able to animate them for economical energy consumption, real-time measurements of the energy consumption have to be taken. For example, this can happen in large office buildings with many employees. By motivating the subjects to reconsider e.g. ventilation, air conditioning and lighting, a considerable amount of energy could be saved. By applying gamification ideas, it could indeed give even rise to an improvement of the general atmosphere in the concerned companies.

However, measuring data in real-time and then relating this data with individuals can pose a major threat to the privacy of the individual person. From the link between identity and location in a building, movement profiles can be derived. The person in charge of the staff could, for example, try to measure the amount of time employees spend on smoking, drinking coffee, etc., if the energy consumption in office rooms drops significantly due to monitors switching to power saving mode. Possibilities of this kind could not only transfer the atmosphere in the company to a very paranoid mood, but are also strictly prohibited by law in many countries like Germany.

To still be able to provide, for example, employees with immediate feedback, strict measures have to be taken in order to protect the privacy of the individual subjects. It must be ensured rigorously that only persons with justified reason have access to privacy-imperiling data, for example an energy manager who has to contractually agree to strict confidentiality, and that no other person can derive anything inappropriate from stored data.

As an important aspect it can be noted that in most situations the actually measured real-time data is not required to be in this fine-grained state. Instead, often an aggregated form is quite sufficient that, for example, summarizes the overall consumption of one whole day. Thus, in an actual environment it could be decided on purpose how the data is processed: The employee herself may exclusively get access to the real-time data while other persons in charge only can see the aggregated, defused form.

To be able to apply this strategy in a real-world environment, an architecture has to be designed and implemented that provides all the functionality required to process and store measurement data in a *privacy-friendly* manner. Because this work has been done as a part of the IDEM project (see [RiCB14] and [idem] as well as [KMPK+14] for details) at the Chair for Network Architectures and Services at TU München, the implementation is tested and evaluated using an already installed real-time measurement system at that chair. However, the architecture is not specially designed for that exact situation, but with as much flexibility in mind as possible.

1.1 Scientific questions

While the development of this architecture aims at providing an application that can actually be used in a real-world situation, a number of scientific questions have to be answered as well.

Which situations regarding energy monitoring do exist in which the privacy of subjects is in danger?

In times of Big Data, the collecting of massive amounts of data is omnipresent and so are applications that are designed to do exactly this collection. Products like the deZem logger used for retrieving actual data values in the evaluation of this work already collect consumption values and post-process them in a cloud-like server application. Thus, it has to be investigated at first, in which situations such classical data collection endangers the privacy of subjects.

Which properties allow privacy-friendly data monitoring?

In order to grant *privacy friendliness*, a set of properties has to be defined that actually describe the characteristics of architectures called *privacy-friendly*. By using these properties it should be possible to design a privacy-friendly architecture and to validate this property.

How can an architecture for flexible data processing and storage provide these properties?

The specified properties have to be applied to the architecture. It has to be verified, which traditional design patterns can be used in an architecture and what has to be changed or derived in order to be *privacy-friendly*. To provide additional value, the approaches taken should be described as universally as possible.

1.2 Outline

Chapter 1: Introduction

Chapter 1 introduces the motivation behind this work and enumerates the scientific questions addressed.

Chapter 2: Background

Chapter 2 is focused on providing the background knowledge needed to comprehend the following chapters to the reader. Thereto an introduction is given to concepts regarding basic privacy preservation in software development, energy monitoring and using encryption methods to achieve confidentiality.

Chapter 3: Related Work

In chapter 3 an overview is given of methods used for privacy preservation in various contexts. These concepts are then compared to the situation this work is based on. The goal is to decide whether those concepts apply to the situation described or not.

Chapter 4: Analysis

In chapter 4 the answer to the first two questions is investigated. To do this, at first situations are analyzed in which privacy has to be protected. From these situations then properties are derived that a *privacy-friendly* architecture has to fulfill. Finally, the chapter translates these properties to actual requirements the architecture has to accomplish. Additionally, the chapter enumerates some other requirements that have to be fulfilled in order to provide the functionality needed to process and store measured data.

Chapter 5: Design

The requirements described in the chapter before have to be applied to an architecture design. Chapter 5 is focused on actually designing an architecture capable of providing the required functionality as well as guaranteeing privacy-friendly data processing and storage.

Chapter 6: Implementation

The chapter 6 gives an overview of the implementation of the architecture described in the chapter before. The focus is laid on the server application, because most of the important, that is, *privacy-friendly* functionality is located in the server rather than in the client code.

Chapter 7: Evaluation

In chapter 7 the designed and implemented architecture is evaluated regarding the properties and requirements described before. Additionally, performance tests verifying the operational capability of the application are discussed.

Chapter 8: Interpretation

In chapter 8 the preliminary findings are summarized and applied to the scientific questions stated above. It is verified whether a satisfying answer has been found to every question.

Chapter 9: Conclusion

Finally, chapter 9 gives a final summary of this work's results and provides an outlook on future progression regarding energy monitoring and privacy preservation.

2. Background

Before proper answers to the questions defined in the preceding chapter can be found, some background knowledge has to be provided to the reader that is required in the following chapters.

2.1 Energy monitoring

The monitoring of the energy consumption in buildings, facilities or whole company structures is already in wide use today and various standards like for example ISO 50001 (see [ISO11]) exist that define how monitoring can be done. Different approaches in hardware and software exist to continuously take measurements of all kinds of different values.

To measure the real-time consumption, measuring probes have to be deployed to the energy supply infrastructure. For example, the energy loggers offered by the German company deZem GmbH are installed near the fuse panel (see [deze]). Inductive clips are attached to the actual supply lines and measure the power.

While everybody is talking about the so-called Smart Meters, the energy monitoring devices discussed in this work generally are different: A classical Smart Meter is deployed and operated by the energy provider company and measures the energy consumed before the feed-in (see, for example, [ErTs12]). In contrast, the devices characterized here are attached after the feed-in point and, therefore, usually have advanced insight into the distributed energy consumption inside the monitored domain.

Most current systems (like the deZem system described in [deze]) use a centralized approach. That means, the values measured by local probes are transmitted to a central processing device that executes post-processing and stores the value in a database. Usually, a Graphical User Interface exists that can retrieve stored sets of data and visualize them. As a consequence of these mostly centralized systems, data that may be privacy-critical often is not immediately aggregated, but processed in the fine-grained state it has been generated in.

Especially cooperation between multiple devices in a distributed system still seems to be rather rare. That means, if multiple devices are installed in one domain, e.g. a building, usually these devices only collect data in their respective subdomain, but do not exchange data points.

2.2 Privacy

Privacy not only is a valid concern of every individual person, but also is commonly protected and enforced by laws and standards. Thus, companies and building operators have to strictly verify any attempt to collect and evaluate any measured data that may threaten the individual subjects' privacy.

For example, in Germany the *Bundesdatenschutzgesetz* (see [bund09]) requires public authorities as well as companies to comply with a number of restraints: Among other things, when person-related information is collected automatically, the subjects have to be informed, may always opt-out and may also request the deletion of collected and stored data.

While the privacy aspect is not the main focus point of standards like ISO 50001, those standards usually explicitly mention the importance of compliance with national and international laws. For example, in ISO 50001 (see [ISO11]) this compliance is specially considered in the PLAN as well as in the CHECK phases. Additionally, persons in charge and in-house points of contact have to be constituted. Regarding privacy, for example, an energy manager could be constituted, who is responsible for the verification of the internal structure.

To design and implement an architecture for automatic data collecting, processing and storing in this context, an application, of course, has to strictly comply with these legal requirements, too. Additionally, to not to lose the subjects' trust in, for example, the company, processing privacy-imperiling data should always take the subjects' well-being into account, no matter what the law would actually allow.

As a means to design such architectures, the eight strategies for "privacy friendly" design defined and described by Hoepman in [Hoep12] provide a good base. These strategies as well as the original definitions are listed in table 2.1.

In his paper Hoepman takes into consideration "existing privacy principles and data protection laws" to enable applications to be "privacy by design" (see [Hoep12]), for example, the OECD guidelines as well as the European Data Protection Directive. While the first four strategies, **MINIMISE**, **HIDE**, **SEPARATE** and **AGGREGATE** affect the context of the application itself, the last four, **INFORM**, **CONTROL**, **ENFORCE** and **DEMONSTRATE** rather concern the managing and operating process that is also required when deploying a full-scale measurement and processing architecture.

2.3 Ciphertext-Policy Attribute-Based Encryption

Generally, crypto systems are divided into two sub classes: Symmetric and asymmetric crypto systems.

While symmetric crypto systems use the same secret key for encryption and decryption, asymmetric systems usually possess two different keys, one private and one public key. The public key is used for encrypting messages and the private key for decrypting messages that have been encrypted with the corresponding public key. Therefore, in contrast to symmetric systems, in an asymmetric system the one encrypting messages does not need to be able to decrypt messages. This comes in handy when designing architectures with components that only may write sets of data, but not read them.

Traditional asymmetric crypto methods like RSA (see [RiSA78]) and DSA (see [KeDi13]) require to supply every user of the system with her own key-pair, i.e. every user possesses a private key as well as an own public key. Hence, the entity responsible for encrypting

MINIMISE	"The amount of personal data that is processed should be restricted to the minimal amount possible."
HIDE	"Any personal data, and their inter-relationships, should be hidden from plain view."
SEPARATE	"Personal data should be processed in a distributed fashion, in separate compartments whenever possible."
AGGREGATE	"Personal data should be processed at the highest level of aggregation and with the least possible detail in which it is (still) useful."
INFORM	"Data subjects should be adequately informed whenever personal data is processed."
CONTROL	"Data subjects should be provided agency over the processing of their personal data."
ENFORCE	"A privacy policy compatible with legal requirements should be in place and should be enforced."
DEMONSTRATE	"[...] A data controller [should] be able to demonstrate compliance with the privacy policy and any applicable legal requirements."

Table 2.1: This table lists the eight strategies for designing privacy-friendly applications stated by Hoepman in [Hoep12].

data must have access to all public keys and also has to know the mappings between keys and identities.

Like other key-pair-based crypto systems, Ciphertext-Policy Attribute-Based Encryption (CP-ABE, see [BeSW07]) utilizes private keys that are deployed to the users, so they can exclusively decrypt encrypted information. However, instead of using public keys to determine the one being able to decrypt a message, CP-ABE makes use of a concept called *attributes*: When generating a new private key for a user, a set of attributes is weaved into this key. These attributes determine the capability to decrypt messages. When encrypting a message, a policy string is used instead of a conventional public key. This policy string specifies, which attributes must be present in a private key in order to be able to decrypt the message.

For example, if a message is encrypted using the policy "**employee *or* energymanager**", the owner of a private key with the attributes **employee** and **management** is able to decrypt the message, because the attribute **employee** matches the policy. By contrast, a private key with other attributes than the two mentioned in the policy would not be able to be used to decrypt the message.

Besides the checking of attributes set, the policy language supports recursive trees using the keywords ***and*** and ***or***. Additionally, numeric constants can be defined as attributes and then inquired by inequations in the policy language (see [BeSW07]).

Like other asymmetric crypto systems, CP-ABE is usually used in a hybrid way (see for example the OpenPGP message format described in [CDFS+07]). The mathematical methods regarding the actual asymmetric encryption are very slow or simply not fitting for encrypting and decrypting arbitrary messages (both cases are true for CP-ABE). Thus, the asymmetric functions are used only to encrypt and decrypt an element of constant size, e.g. a random group element. This element is then used as a key for an arbitrary, fast, symmetric procedure like, for example, AES.

To encrypt a message (see figure 2.1), a random sequence of bytes is generated. These bytes are used as a key for AES to encrypt the actual message. Then the key is asymmetrically encrypted using the specific crypto system and the public key of the designated recipient. Both resulting sequences of bytes are then concatenated such that a combined ciphertext is formed, that can be transferred or stored.

When the recipient receives the combined ciphertext (see figure 2.2), at first she uses her private key to decrypt the first part into the sequence of bytes used as symmetric key. She then decrypts the actual message using this key.

Therefore, a ciphertext derived by an asymmetric method usually consists of two parts: The asymmetrically encrypted key and the message symmetrically encrypted using that key.

It is worthwhile noting that both the policies and the attributes advect in a mathematical sense into the mathematical group elements that are saved as keys or ciphertext (see [BeSW07]). Therefore, CP-ABE really enforces access policies cryptographically (in contrast to programmatic rules that are obeyed). This makes the crypto system particularly worthwhile for cases where access to information is to be restricted based on a role concept. As a consequence, CP-ABE seems to be well-suited to be used as an underlying solution for a privacy-friendly storage system.

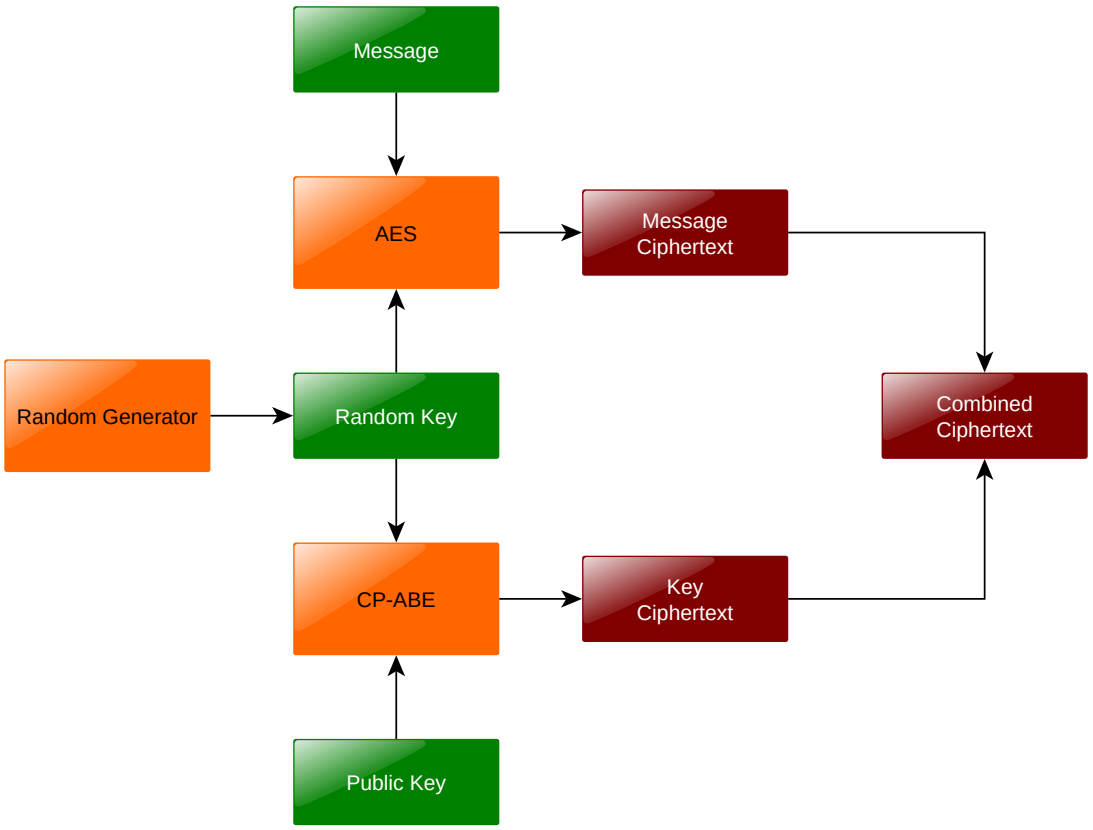

Figure 2.1: This diagram demonstrates how CP-ABE applies hybrid encryption in order to process arbitrary messages.

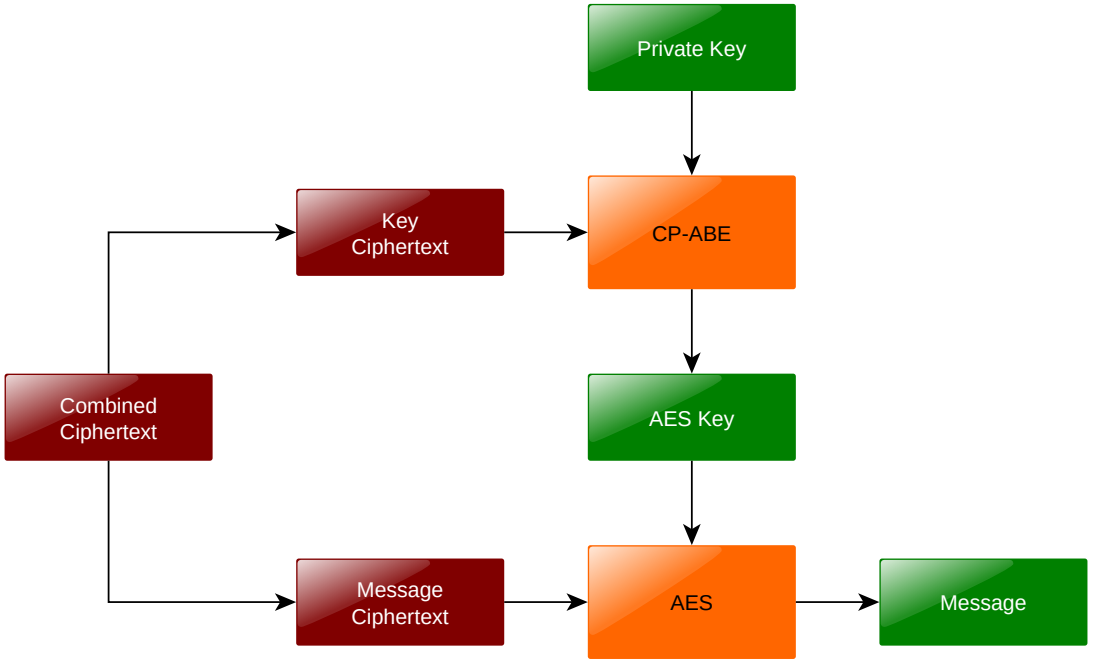

Figure 2.2: This diagram demonstrates how CP-ABE applies hybrid decryption in order to restore arbitrary plaintext messages.

3. Related Work

This chapter is focused on giving an overview of other works that feature one or more of the concepts and problems discussed in this work. All of these works are described shortly and then compared to the subjects discussed in this work.

In their publication *Security for Future Networks: D6.1 - Cryptographic Requirements* (see [AMKL+13]) the authors Abdalla, Kreutz, Lyubashevsky, Malichevskyy and Santos describe a wide range of problems and solutions for dealing with security in the so called *future networks*, i.e. "environments based on virtual networks and clouds". Many of these considerations also apply or are directly related to the protection of the privacy and have some bearing on the concepts described in this work.

One part is focused on methods for secure data outsourcing and computation when working with untrusted providers. Functional Encryption and Fully-Homomorphic Encryption (FHE) are proposed as possible solutions to this problem, even though it is stated that "current FHE schemes remain mostly of theoretical interest and are still far from being practical". However, because this work assumes the provider of the architecture to be a trusted party, these solutions do not match to the design described in this work.

Encryption is also mentioned as a means to provide confidentiality and privacy. The publication even mentions Attribute-based Encryption (ABE) methods like CP-ABE used in this work as well as Identity-based (IPE) approaches.

The authors Yang, Zhong and Wright describe in their paper *Privacy-preserving Queries on Encrypted Data* (see [YaZW06]) methods for storing encrypted, privacy-critical data in a database as well as running queries in order to retrieve already stored sets of data. Therefore, the paper addresses one of the major questions when encrypting databases also asked in this work: How can an encrypted database be searched for specific entries, if as much as possible of the information is hidden from the view of the entity executing queries?

The model designed in the paper uses a frontend entity that translates queries into a form applicable to retrieve the correct results from a database server as well as decrypts the results, such that the issuer of the query receives the decrypted version of the queried data set. While the number of query types supported is limited, the authors show that their solution may perform well on a large set of data.

In comparison to the paper, this work uses explicit plain text indices for retrieving specific documents from the database. By doing that, the step of translating queries can be omitted. The drawback of this approach is, of course, that no advanced queries can be executed like in the approach discussed in that paper. In this work, this is mitigated by the client GUI caching results and, therefore, running queries on the set of decrypted data values.

In their paper "k-anonymity: A Model for Protecting Privacy" (see [Swee02]), the author Sweeney describes the property k-anonymity as a means to characterize whether a set of data meets the requirement that "the information for each person contained in the release cannot be distinguished from at least k-1 individuals whose information also appears in the release". She considers this a good solution in situations, where a data set has to be published, but no privacy-critical information may be retrievable from the published set.

The paper "L-diversity: Privacy Beyond K-anonymity" (see [MKGV07]), written by Kifer, Machanavajjhala, Gehrke and Venkitasubramaniam provides detailed information about attacks on k-anonymity and then describe a more powerful property called l-diversity. This property is characterized in a formal way as well as evaluated against practical needs.

While k-anonymity and l-diversity aim at protecting the individual subject's privacy, the resulting data sets would only be suitable for coarse-grained analysis in the context of energy monitoring. For individual feedback, the access to personalized data must still be possible.

The publication "Achieving energy efficiency through behaviour change" (see [BaGM13]), produced by the European Environment Agency and written by Barbu, Griffiths and Morton consolidates a number of studies about behavioural change in energy consumption. Among other things, the text focuses on the relationship between consumer behaviour and consumption practices (over time). Feedback on energy consumption is described such that "without an appropriate frame of reference, consumers cannot know whether their consumption is excessive".

The results of the publication show that at least some influence on the energy consumption can be shown, but that no adequate quantification exists so far. It is stated that this mainly results from the lack of reliable long-term studies, so it can be expected that further investigation in that direction can show more suitable results. These results would be of great interest when associated with the architecture resulting from this work.

The authors Erkin and Tsudik cover in their paper "Private Computation of Spatial and Temporal Power Consumption with Smart Meters" (see [ErTs12]) the usage of the so-called Smart Meters for measuring and monitoring the power consumption of individual consumers. To do this, temporal and spatial aggregation is used on the individual values. Homomorphic computation and encryption is used to protect the data and preserve the privacy.

In their publication "Privacy-friendly Energy-metering via Homomorphic Encryption" (see [GaJa11]) the authors Garcia and Jacobs propose a solution for energy metering similar to that described before. Once again, homomorphic encryption is used in order to preserve the subjects' privacy. The focus is laid on fraud detection, i.e. detecting malicious nodes that deliberately provide false values, and preventing other users from receiving privacy-critical information.

In contrast to this work, both papers consider Smart Meters, which usually measure the consumption of whole buildings. Additionally, malicious nodes are excepted explicitly,

because a trusted party as operator of the measurement architecture is assumed. This applies to many other applications of secure multiparty computation like the one described in [LiPi08].

In the paper "Identity based encryption from the Weil pairing" from Boneh and Franklin (see [BoFr01]) a working identity-based encryption (IBE) scheme is proposed. IBE schemes allow the usage of an identity information as an encryption key and, therefore, manage to require only one global public key for encryption, while decryption, once again, is done using a traditional private key.

Because the encryption and storage scheme used in this work is mainly independent from the asynchronous encryption method in use, IBE could be used instead of the actually used CP-ABE without any problems. However, the explicit usage of identities as encryption entities would immediately reveal the underlying identity. By contrast, CP-ABE allows to specify any attribute as a policy for encryption, so identity and the ability to decrypt a document are effectively decoupled. This allows for a more flexible dealing with keys and identities.

4. Analysis

Because the purpose of this work is not only to give an answer to the three questions defined before, but designing and implementing an architecture reassembling the gained knowledge, this chapter is divided into two parts: In the first part, the questions are analyzed and conclusions are drawn accordingly. In the second part, a requirements analysis is performed to find the functional and non-functional requirements for the architecture to be designed.

4.1 Privacy and energy monitoring

As mentioned before, the eight privacy design strategies (see [Hoep12]) provide a reasonable base to derive requirements regarding a privacy-friendly design and to give an answer to the questions stated before.

4.1.1 Privacy-critical situations

The possible exploitability of data sets consisting of a greater number of individual measurement values depends on the type of measured values as well as the fine-graininess.

For the former, e.g. energy consumption measurements in individual rooms can cause inferences to be drawn about the behaviour of subjects working in these rooms. When the subjects switch the light on or off, or leave the computers in an idle state causing power saving functions to kick in, these changes usually are visible in a real-time energy consumption visualization. However, other measurements, like, for example, of humidity values may allow inferences to a much lesser extent, especially if these values are influenced distinctly by environmental factors that are not caused by humans. Though, even measurement factors that may seem to have no impact on the privacy should not be underrated when estimating the privacy danger potential.

For the latter, the fine-graininess surely has direct impact on the information density of the data flow and, thus, the amount of information or knowledge that can be drawn from the collected data about individual subjects. Because measurements usually are taken at discrete points in time and, thus, represent snapshots of the monitored factor paired with the respective timestamp, every value already contains a temporal aggregation of a continuous[1] time span implicitly. The more information one data value aggregates from the *actual* value, that is, the continuous, real-time variable, the less information can be

[1] Of course, unless the factor measured is discrete on its own.

derived about a specific point in time. That means, that coarse-grained flows of data are less dangerous and, thus, by artificially lowering the density, a degree of coarse-graininess can be reached that is acceptable for all parties.

Partially, this also applies to implicit spatial aggregation, that is, for example, the measurement of the combined energy consumption of multiple rooms. However, in that situation the problem is that rather than the individual privacy now the group privacy is in danger: Inferences could be drawn about the behaviour of a whole group of subjects. Because of the possible impact monitoring still could have on this kind of privacy, spatial aggregation is considered an inferior method for preserving privacy compared to temporal aggregation.

Besides both spatial and temporal density of information contained in a flow of energy consumption values, a person trying to derive information from the data flow usually has to have access to additional knowledge in order to properly generate additional information. For example, if she does not know from which room a specific set of data has been measured, no information can be linked to an individual person. Because of that, it has to be taken into consideration, what knowledge may be accessible or made available.

4.1.2 Properties of a privacy-friendly architecture

To define the properties and requirements of a privacy-friendly architecture, it seems to be a good idea to straightly follow the eight design strategies described in [Hoep12]. These strategies have to be applied on the situations mentioned above in order to set the boundaries of how such an architecture should be designed.

The first strategy **MINIMISE** implies that only a minimal amount of data is collected, so no unnecessary data is processed. While an application usually can not decide alone what is collected, but the person in charge makes this decision, the application still has to provide the means to regulate the amount of data collected and reduce this amount when necessary. Thus, the architecture should implement a flexible filter system to drop information that is not needed. For example, data values could be automatically anonymized or pseudonymized. Additionally, the application could implement the data collection in an opt-in kind of way. That means, data is not collected unless it is explicitly stated to do so. That way, no data is collected *accidentally*, e.g. when the system has been misconfigured. Moreover, it would be more obvious, what kind of data is collected, especially for people verifying the correctness of a configuration.

The second strategy **HIDE** is meant "to achieve unlinkability and unobservability" (see [Hoep12]). As mentioned in the paper, encryption traditionally provides a solution for this problem. By encrypting the data and making sure the secret required for decryption is only known by authorized persons, the information is protected from unauthorized access. While encryption usually is only used for transport confidentiality, to protect the persistently stored data in the database these values should also be encrypted. To protect these encrypted data values from decryption by an unauthorized person, an encryption scheme is required where the ability to encrypt and decrypt is separated into two pieces of knowledge: The server holds the piece required for encryption, while only the authorized persons have access to the other piece of knowledge. This scheme is provided e.g. by traditional methods like DSA (see [KeDi13]) and RSA (see [RiSA78]) as well as CP-ABE (see [BeSW07] and section 2.3). To be able to properly encrypt all of the privacy-imperiling data, an encryption scheme has to be designed that makes sure, after encryption only harmless data is still visible.

As a third strategy Hoepman states to **SEPARATE**, i.e. process the data "in a distributed fashion" (see [Hoep12]). While a classical approach for such an architecture would contain a central server instance that executes all processing and storing tasks, this always

increases the risk of a massive data leakage if this server gets compromised. But on the other hand, by distributing the architecture into many small instances of such a server, information flows between the individual instances would no longer be possible. Therefore, a compromise has to be found between these two concepts. In particular, that means the individual local parts of a distributed architecture do have to be able to cooperate by exchanging information. In order to minimize the amount of unneeded data in every node, the methods required for the strategies **MINIMISE** and **AGGREGATE** could be applied. For example, a hierarchy could be formed with a tier 2 layer processing real-time data and a tier 1 layer for central accounting calculations. The accounting devices usually do not have to have information in real-time, so the tier 2 devices could send an aggregated version, e.g. a summation of each month, to the tier 1 devices.

The fourth strategy **AGGREGATE** is used to decrease the information density in a flow of data by grouping the information and, thus, removing parts of the fine-graininess. Because reducing the amount of data, in particular if the density is higher than required, is an important aspect in making an architecture privacy-friendly, a comprehensive aggregation functionality has to be provided. As the amount and kind of aggregation possible heavily depends on the environment to be monitored, the architecture should enable the operator to flexibly configure the aggregation functionality to solve as many situations as possible. For example, in the hierarchical environment described in the paragraph before, the operator should be able to decide precisely which device is provided with which level of detail and on which attributes the aggregation should take place (e.g. over time spans, rooms, companies, etc.).

As many laws regarding privacy state that authorities have to **INFORM** the subjects "whenever personal data is processed" (see [Hoep12]), this strategy may not be neglected when deploying an application collecting and processing privacy-critical data. Certainly the application itself has no means to directly inform the subjects and the operator of the architecture must remain in charge for doing this. However, the application can assist the operator by providing information that can be handed to the subjects in order for them to verify how data regarding them is processed. In particular, by making all data regarding one subject visible to her, the amount of data collected is obvious to her. Because usually laws and standards include opt-out possibilities, subjects may always decide to allow or deny access to privacy-critical data.

As stated in [Hoep12] the **CONTROL** strategy gives the subjects "the right to view, update and even ask the deletion of personal data collected about her" as well as the ability to control on higher scale what information might be collected and what not, and, therefore, goes one step further than the **INFORM** strategy. For the architecture, this means in particular that an opt-out possibility must exist, and that a subject is no longer concerned by any collection of privacy-critical information if she decides to do so. Because the architecture is required to be able to be used for accounting issues, automatic, user-initiated deletion of data records may seem to be not feasible. However the architecture should provide at least the operator with the means to delete sets of information. That way a subject may always request the deletion of data records. While this solution may not be the best, it makes a compromise, which should be applicable in most situations.

While the legal part of the **ENFORCE** strategy as always has to be fulfilled by the company or the architecture operator, enforcement of this privacy policy can be assisted by implementing access control (as described in [Hoep12]). While the encryption approach mentioned above already provides access control on a cryptographic level, an additional, traditional access control system using e.g. username and password as credentials can be used at server side to prevent any unauthorized access.

The last strategy **DEMONSTRATE** requires the operator to "be able to demonstrate compliance with the privacy policy and any applicable legal requirements" (see [Hoep12]). Therefore, the application itself has to provide means enabling the operator to prove this compliance. For example, information about the processing storage schemes should be available in a format that can easily be validated by third-parties. This may also turn out to be helpful when certifying application environments.

4.2 Requirements analysis

Besides the requirements that can be derived from the properties stated above, the architecture has to fulfill a number of rather normal requirements that are required for processing and storing data values as well as accessing them.

This section is focused on these functional and non-functional requirements.

4.2.1 Functional requirements

Because the architecture should be able to take data values and process them as well as allow subjects to access the persistently stored values afterwards, the functionality of the architecture can be divided in two parts.

4.2.1.1 Processing

To be able to process data values from measurement devices or other sources, the architecture requires a network endpoint that can accept those values. Along with the value itself, additional information like the sensor identifier, the unit and a timestamp have to be specified. The architecture should be able to take raw and processed values from measurement devices like the deZem logger and convert them to a general format if needed. The endpoint interface has to support taking data values from various sources, e.g. clients written by oneself or a deZem logger device.

Once values have been received, the architecture should be able to process these values in a way required by the processing stage itself, as well as by privacy-concerning means. This includes aggregation, filtering, dispatching[2], calculations and modification of meta data information among other things. By providing these, an operator should be able to adjust the architecture in order to answer a wide range of different situations.

Once the data values have been processed, they have to be stored persistently, for example in an external database. As described above, an encryption scheme could be used to store these values in a privacy-friendly manner.

4.2.1.2 Accessing

When accessing data, subjects should be able to issue queries on the stored data sets. While the stored data values may have been encrypted, the subjects must be able to get the data they are interested in, by at least specifying a time interval. The data values resulting from these queries have to be transferred in the same encrypted format they have been stored, so the subject can decrypt them locally.

[2] In this context, dispatching means to decide between multiple different processing paths to proceed the processing of one data value.

4.2.2 Non-functional requirements

Of course, the privacy friendliness is the most important non-functional requirement. But in order to provide an architecture ready-to-use for production, other requirements have to be defined and fulfilled as well.

To provide security features like confidentiality, special care must be taken when designing and implementing security-critical functionality in the architecture.

To be able to process and store even a large number of data values per second, the architecture has to be scalable on a high degree. As much work as possible should be parallelized, so the architecture can utilize the hardware effectively.

Finally, the architecture has to be flexible in configuration and deployment, so a wide range of situations can be covered.

5. Design

In the analysis, a set of requirements that must be met by a privacy-friendly architecture was identified. Besides these special demands, the architecture must be able to act like a standard architecture that provides a specified service to users and processes on the network.

Therefore, this chapter not only takes into account the privacy-friendly design, but also the design of an architecture ready to be used day-to-day by users. Also, the main focus of this chapter lies on the server application.

5.1 Server architecture

As from now, the term *server architecture* identifies the part of the architecture that is responsible for data processing and storage on the one hand and data access (by users or processes) on the other hand.

Typical distributed applications can be categorized in multiple ways. One of them is the distinction between classical client-server-architectures and decentralized architectures.

This section aims to provide a comparison between these approaches while considering advantages and disadvantages for a privacy-friendly architecture.

5.1.1 Centralized approach

A classical centralized client-server approach like the solution provided by deZem (see [deze] for details) features one central application that provides services to one or many remote client applications invoking service methods on the server. Using these service methods, the clients, for example, could in this case query the server for measured data or add new data points to be processed and stored.

Figure 5.1 shows the basic structure of such an architecture. The central server instance is responsible for storing and processing the actual data. Two types of clients connect to the server: *Input clients*, for example a measuring device, add data points to be processed. *Access clients*, on the other side, query the server for already stored sets of data, be it a client application that provides a graphical analysis, or a program that further processes these data sets. To restrict the access to data sets to certain users, an ACL system or the like has to be used.

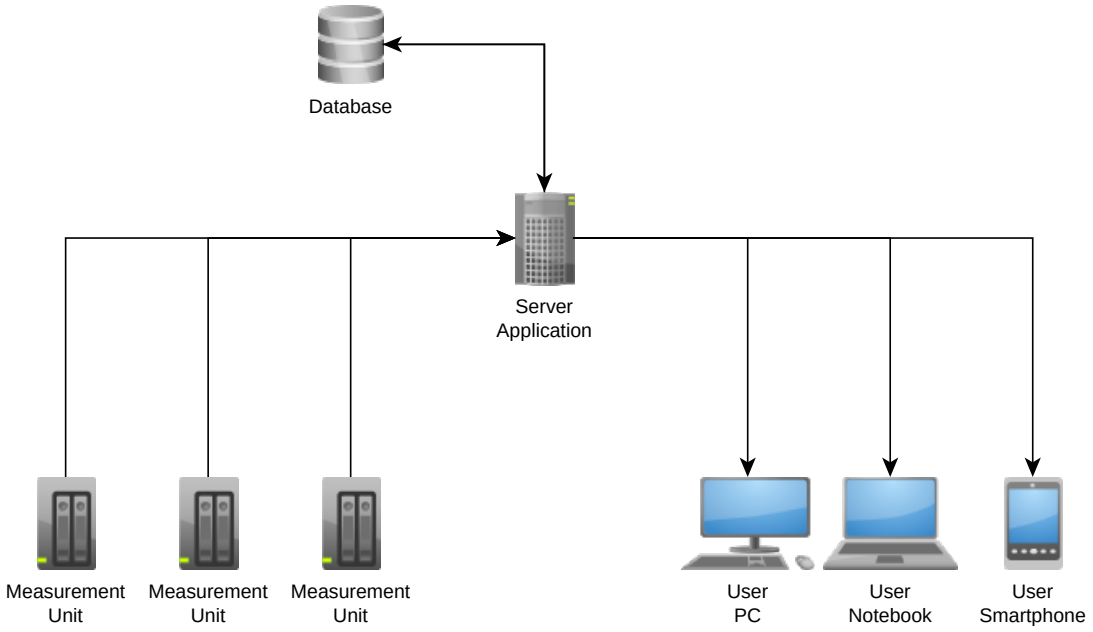

Figure 5.1: An example of a centralized architectural structure. One or multiple measurement devices provide sensor data that is sent to the server application. The server processes the data and stores the results in a database. If users (or processes) want to access data, they connect to the server and execute queries.

The drawback of this approach is the risk that arises when too much privacy-endangering data is stored at the same location. If that one server gets compromised, an attacker instantly could gain access to all the data that is stored there.

While there are still possibilities to circumvent this problem, a pure centralistic approach always has the restriction that every single data point eventually passes this one server application, so the application depicts a single-point-of-failure, both in security and privacy as well as redundancy terms.

5.1.2 Decentralized approach

Although the term *decentralized* may awake associations with peer-to-peer systems, in this case the term refers to an approach where multiple independent server instances exist that do not interact with one another. In figure 5.2 one can see an example setup with three server instances, each with its own database server. Every server instance processes measurements from (in this case) one or more measurement units. When a user wants to access stored data sets, she connects to the server holding that specific data set and queries for it. To be able to do that, the client has to know on which node the data is stored. Even so, this approach could be extended with an indexing service to provide clients with that information.

The general idea of this approach is the partitioning of the processing domain into self-sufficient parts. By doing this, the impact of one compromised node is minimized, because, in the worst case, only the part of the data can be obtained by an attacker that has been processed and stored on this node. While this possibility still means a major threat for the user's privacy, the risk is mitigated slightly at least. Although, room for improvement still exists.

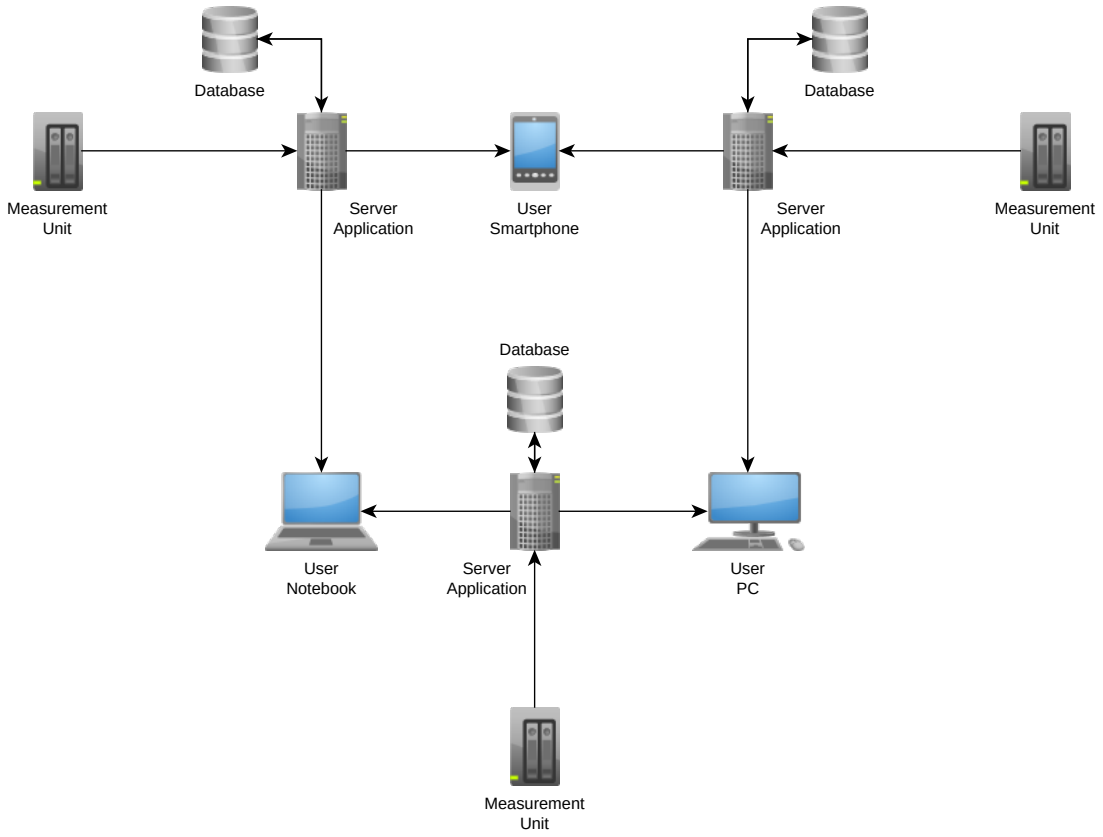

Figure 5.2: An example of a decentralized architectural structure. Multiple measurement units provide data points to a number of independent server applications. Clients connect to one or more servers and query for data. As long as there is no directory service, the clients have to know, on which server a specific data set is stored, to retrieve it.

Another problem of this approach is the fixedness in configuration possibilities. For example, to provide the individual users with their personal consumption data on the one hand and the building manager with the aggregated energy consumption of an entire company on the other hand, the separation into independent domains is a non-trivial task: Either both processing domains are combined on one server node and, thus, again exposed to a single attack, or one of these processing steps is relinquished in favor of the other.

5.1.3 Hybrid approach

Both of the approaches mentioned in the sections above were rather basic approaches and therefore not really fitting to provide a substantial solution to the problem of privacy preservation. Because of that, a reality-related approach rather uses a combination of these ideas described above.

Different streams of data usually have an affinity to an actual location, e.g. the energy consumption in a specific room, floor or building. While consumption information may be required in devices not located near that location, these entities usually are content with coarse-grained, aggregated information.

Because of that, those streams can be separated into different domains. Here, a domain depicts a group of data streams that belongs together. Once any such groups have been identified, the processing can be decentralized using these domains.

For example, let there be an office building subleased to a number of companies (see figure 5.3). The building owner operates an energy measurement system that can fine-grainedly measure the consumption in all office rooms. While the energy manager of the building itself usually is only interested in getting aggregated consumption values (e.g. for a total month) for accounting, the energy manager of a specific company may want to optimize the efficiency of the company by monitoring individual rooms.

This opens the possibility to separate both processes into domains possibly running on different hardware devices: The actual sensors send the fine-grained data points to a local device that stores and processes the streams of data for e.g. an individual company. An aggregated form of these data points is then forwarded to a central device that can be used for accounting.

Besides an improvement of the overall scalability, this separability lays the ground for a concept that could be called *privacy-by-configuration*: Information (i.e. streams of data) are configured to only be forwarded to a place where it is needed and in a way that it is needed.

In the example above, the building energy manager does neither need to have access to fine-grained information regarding individual rooms nor is she interested in the names of employees.

While this allows to straighten out the overall processing and therefore avoids single processing devices to become too desirable for attackers, it still does not prevent data leakage in the case of a compromised server or an overly inquisitive employee. To also provide this feature, the architecture must therefore implement another layer of privacy preservation.

5.1.4 Privacy using public-key cryptography

While classical access control systems are able to manage access restrictions by e.g. using a set of rules, these rules tend to be overridable. That means, once the rule is disabled, access may be granted to subjects that were not able to do so before. This problem persists as long as the processed data is permanently stored in an unencrypted form. However, by encrypting the data using a secret that only the intended subjects do possess, no other subject can decrypt and, therefore, access the data, even if the whole system, including its database, is compromised.

Because the server system has to be able to encrypt data, but must not be able to decrypt it, an asymmetric encryption system provides an excellent solution to this requirement. The server only possesses one or more public keys, while the corresponding private keys are kept secret by their holders.

5.2 Data processing

Because the basic idea of the architecture is to assure that the processing is privacy-friendly itself on the one hand and its outcome non-privacy-violating on the other hand, the process of handling and storing data must be the main focus of the design process.

The following sections describe in detail the decisions made in constructing this architecture.

5.2.1 Data model

The data model used by this architecture is centered around the concept of data points. A data point represents the value state of a subject at a specific point in time. Because values are measured within discrete intervals, a presumption must be made for the period

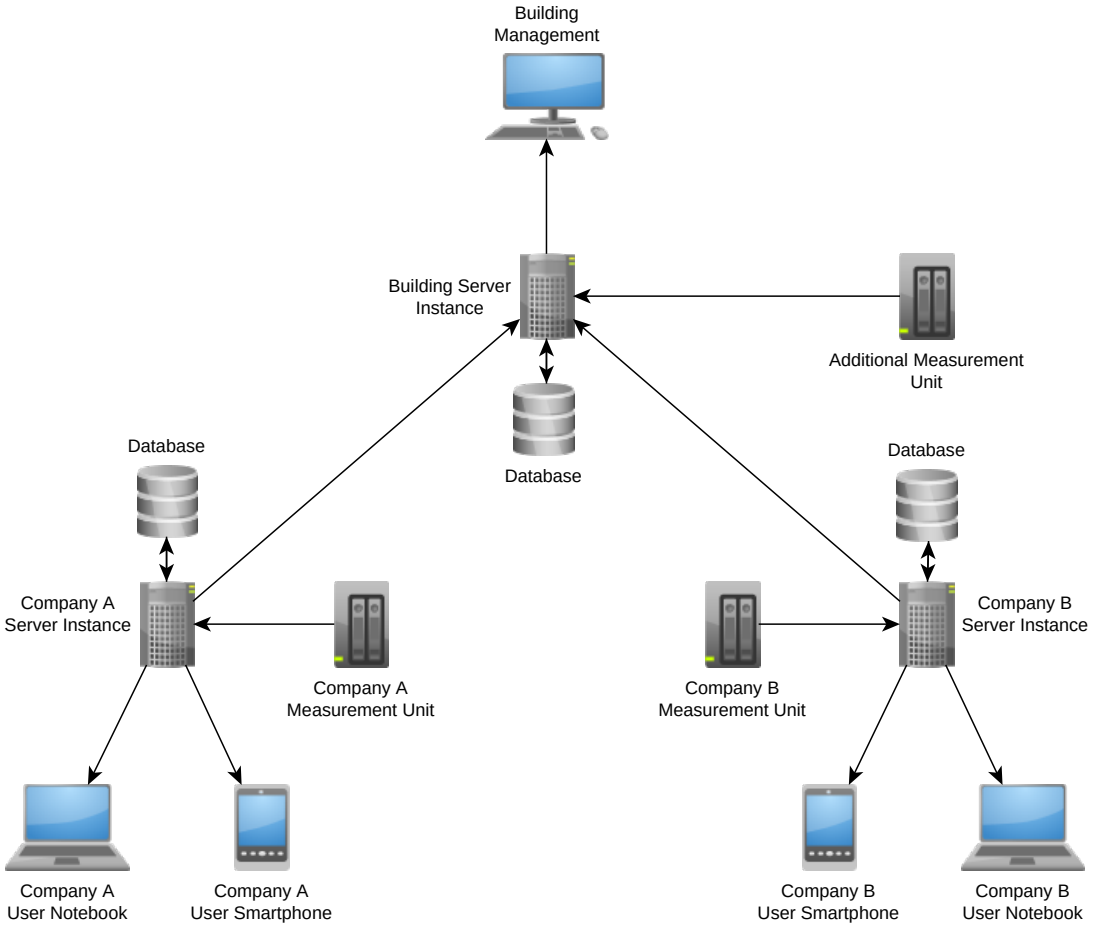

Figure 5.3: An example of the hybrid architectural structure. Measurements are processed and stored by devices near the measurement location. Users having access to fine-grained data streams can directly access them using these devices. Views on data that are relevant on a more global level are forwarded to another server instance that collects these data streams from multiple devices below. While this and other examples depict hierarchical architecture, this approach is not limited to such structures.

between measurements. Because the measurement devices used for this work issued new data values whenever a change was observed, this data model is designed to assume a data value to last until the next value is measured.

Each data point contains the timestamp it has been measured at in UTC. The storage of the measurement value itself requires some special consideration, because measurement devices often provide a number format that not only consists of a single floating point number, but rather a combination of values that has to be normalized or manipulated otherwise to yield an absolute, comparable number.

As a consequence, in this data model a dynamic definition of the concept of a value is used: The actual data that is stored as value depends on the value type and is picked at run-time. To be combinable with data values of other types, every value type must comply to a number of requirements: It must support being converted to an unambiguous basic floating point value and support mathematical basic operations like addition, division etc. with other values of arbitrary type.

Besides these two basic information entities, a data point holds the unit of the value, the name of the flow (see 5.2.4.2) and a set of meta data information (see figure 5.4).

The meta data is individually determined for each data point and contains information about the acquisition and context of the measured value. While some meta data is directly generated by the sensor itself, other meta data must be derived by using external knowledge.

As one can see in the figure mentioned above, every piece of meta data consists of one part describing the type of information and one part representing the information itself. They are called *class* and *tag* respectively. Multiple tags can have the same class and no order exists for tags of one specific class. Using this model for storing meta data, relationships like *measurement affects person A, B, ...* can easily be expressed.

Except the flow field partially, there is no real relationship between individual data points. While this means an overhead in storage space at first, it also allows supporting dynamic switching of entities in the context between data points. For example, if the energy consumption of a room should be allocated to the overall consumption of employees using that room and there is knowledge on when the individual persons are present in that room, the meta data class **person** can just be switched between two data points.

Because the model is meant to be implemented in a way supporting serialization and deserialization for permanent storage and transmission over a network connection, client and server can both use the same implementation of the model.

5.2.2 Storage model

A traditional architecture would simply take the data point documents described in the last section, serialize them and write them to a database or another conventional permanent storage.

However, because the data points contain mostly information that is highly precarious in terms of privacy preservation, leakage of concerning bits of information must be prevented.

To achieve this, a storage scheme is used that makes sure an attacker, who can obtain access to the permanent storage, is not able to extract any useful information. The storage scheme is designed around *Ciphertext-Policy Attribute-Based Encryption* (CP-ABE), the asymmetric crypto system described in section 2.3.

As a first step in designing a storage model, it appears that, in order to minimize overhead, data points should be consolidated into blocks, because CP-ABE introduces a significant

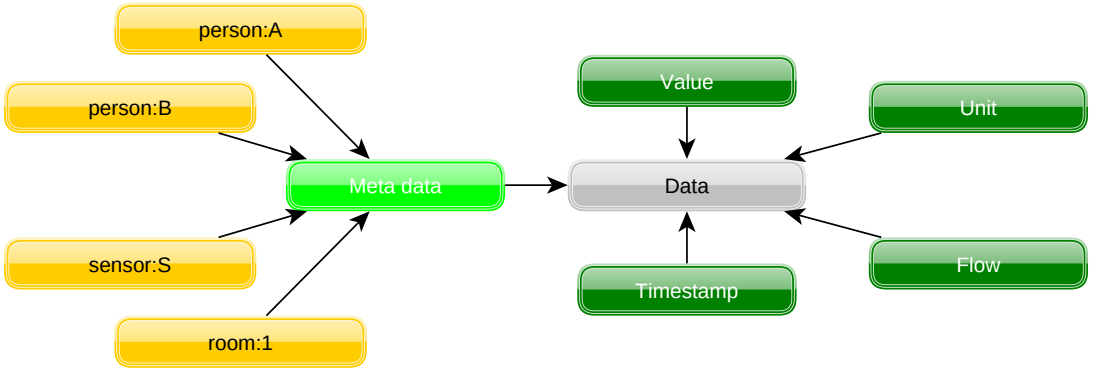

Figure 5.4: The data point entity of the used data model. Arrows denote a *belongs-to* relationship.

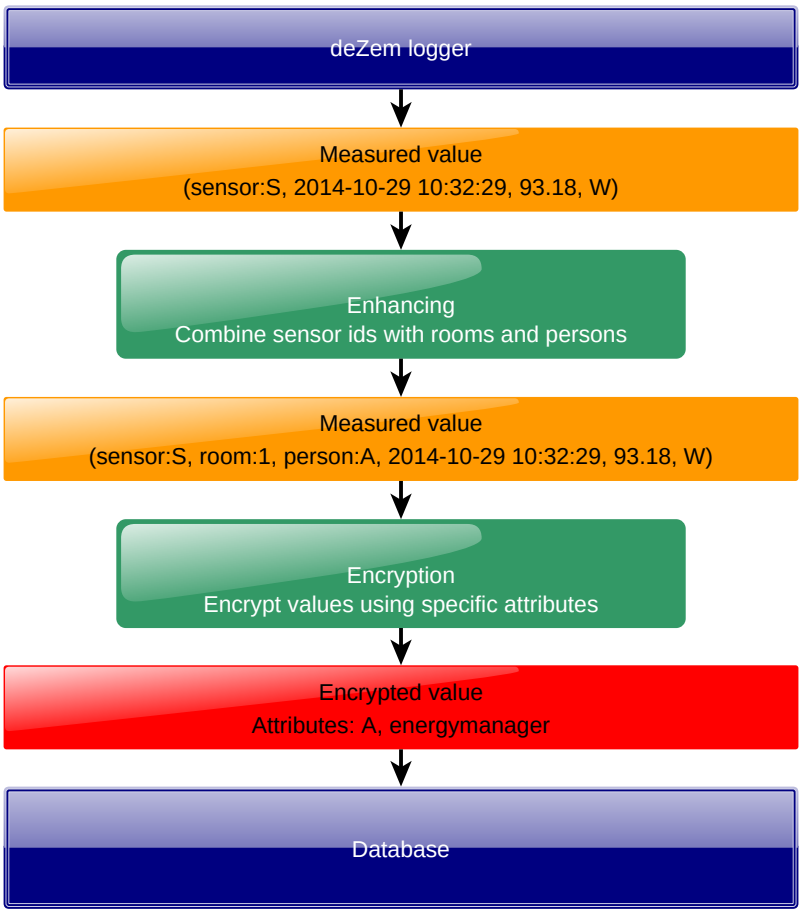

Figure 5.5: This figure describes the basic procedure of processing and storing data points. The measurement device (e.g. a deZem logger) generates a data point with some basic meta data attached to it. The architecture then derives a set of additional meta data tags like room or person. When the processing is finished, the data point is encrypted using all identities derived in the step before and then stored in a classical database.

amount of additional data that needs to be stored along with an encrypted block of payload. Since this block does not depend on the actual size of the plaintext message, this can be compensated by choosing bigger blocks of payload. As from now, the document containing an encrypted set of data points is called *block* document.

To additionally decrease the amount of storage space required, it is desirable to support shared access for data blocks that can be accessed by multiple users. While those blocks could be encrypted using a dynamically assembled policy containing identifying attributes of all target users, this approach contains a tricky pitfall: The policy used for encrypting a chunk of data is included in the ciphertext in unencrypted form. This means that an attacker could always see who can decrypt a block of data. In a system that dynamically assigns and revokes access rights to data points based on presence, that could imply the knowledge of mutual presence of persons.

To bypass this problem, the usage of access sharing by CP-ABE policy should be avoided. Instead, every user and/or attribute, respectively, that should be able to decrypt the data, must have an own document exclusively.

So as to nevertheless share the actual data among multiple identities, the storage of effective data points and data access information must be separated.

For that reason, another type of document is introduced: the *index* document (see figure 5.6). The content of an index document itself is encrypted using CP-ABE[1] and a policy containing one single attribute (that therefore is "public" in the sense of an attacker having access to the database). Of course, an index document must be created for every user/attribute that should have access to the actual data. To lessen this overhead, index documents can last for a specific amount of time until they are renewed.

On the other side, the block documents no longer have any relation to an attribute or CP-ABE in general. Instead, the content (the set of data points, that is) is encrypted using a strong symmetric cipher like AES and a randomly generated code of bytes, the *AES key*. Additionally, another random code is created and added in unencrypted form to the document as a findable (referring to database queries) field. This code is called *reference id* and is used by all users having access to the data to retrieve the encrypted data blocks (see figure 5.6).

Both the AES key and the reference id are then stored in encrypted form in the corresponding index documents. If a user is able to decrypt her index document, she can restore both codes. By using the reference id, she can retrieve the correlating data blocks and decrypt them with the AES key.

An index may have a specified life span in which all emerging data blocks are encrypted and tagged with the same pair of keys. On the other hand, one data block may belong to one or more index documents. Figure 5.7 further demonstrates these relationships.

As a last optimization, the sharing of single reference ids is removed. Instead, every index gains its own reference id. To accommodate this, the block documents in turn no longer contain one reference id, but now each of them contains a set of reference ids (see figure 5.6). This adds another barrier between the ability to connect knowledge as well as allows to selectively remove and garbage collect blocks whose indices have been deleted by the users bit by bit.

The already mentioned figure 5.6 demonstrates the final model that reassembles all of these considerations.

[1] Technically, CP-ABE is a hybrid crypto system. That means, CP-ABE itself is only used to encrypt and decrypt a secret keyword of a fixed length. This keyword, in turn, can be used by a conventional, symmetric cipher to encrypt and decrypt, respectively, the actual payload. CP-ABE uses AES as symmetric crypto system.

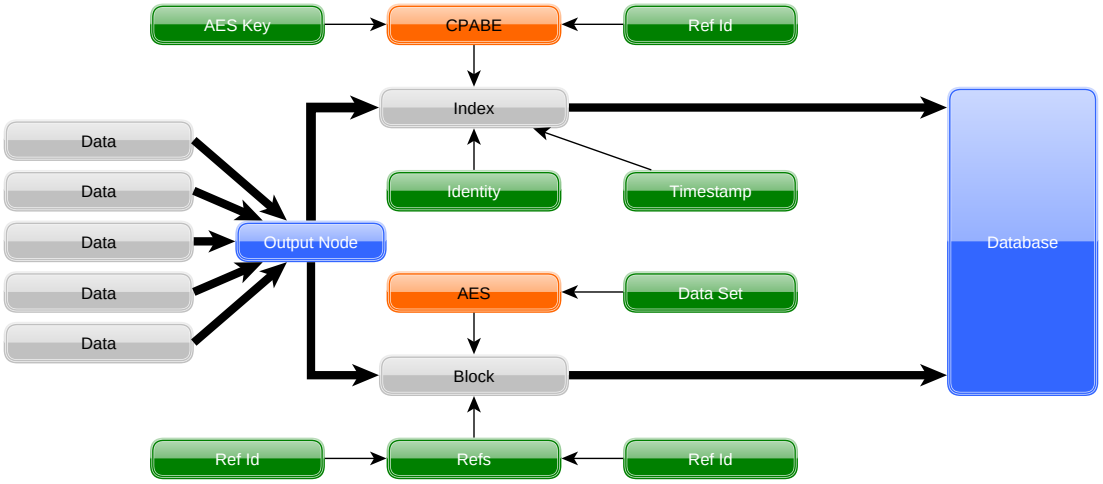

Figure 5.6: This figure demonstrates the final storage model used for privacy-friendly permanent storage. Bold arrows denote a process, while thin arrows signify a *belongs-to* relationship. The output node is responsible for serializing data points into the final marshaled form that can be written to a database, that is, a set of index and block documents.

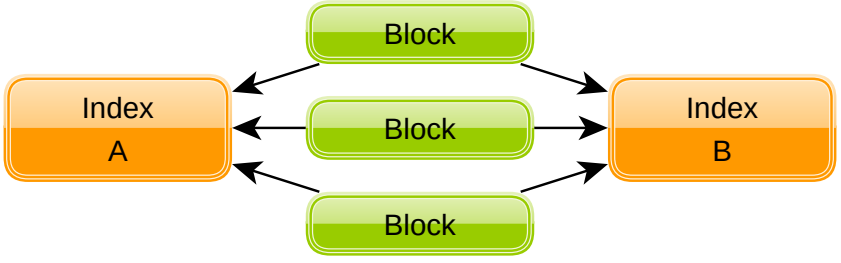

Figure 5.7: Every block document references one or more index documents with the same timestamp.

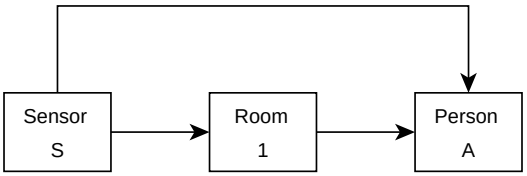

Figure 5.8: This figure shows the knowledge relationships between the three meta data classes *sensor*, *room* and *person*. The class *person* is only reached using the intermediate class *room*.

5.2.3 Enhancing and knowledge

Once a data value is measured, usually a set of simple meta data tags is added (e.g. the identifier of the sensor, the measuring device, etc.) To provide additional information that is required for assigning identities to individual data points and encrypting them accordingly, another node type must be defined that works on the meta data information, the *enhancer node*.

While the term *enhancing* seems to actually target only the augmentation of meta data information density, nodes of that type can likewise be used to reduce the density; a separation of both manipulation types would make no sense in respect of the implementation. Therefore, enhancing is meant to be referred to the act of changing the information density in the desired manner.

An advanced, dynamic enhancer usually is equipped with *knowledge*. The term knowledge targets an abstract set of information that leads to transitive relations between classes of meta data. The transitivity feature is of special importance in cases where the intermediate classes are not used in the end, but are the only way to reach the classes of interest (see figure 5.8 as an example).

For example, this set contains the knowledge that a specific sensor **S** measures the energy consumption in room **1**. It also knows that person **A** resides in room **1**. Based on that knowledge it can assume that data values from sensor **S** depict the energy consumption of person **A**.

To be able to store this knowledge in an applicable way without having understanding on the knowledge itself, a data structure is needed that can hold this set of information. A knowledge provider is responsible for augmenting this data structure, while the processing part of the architecture utilizes the collected information to tag the incoming data points. The implementation details of the used knowledge providers is covered in the next chapter.

A tree of matching rules is chosen as such a data structure suitable for easy application on meta data sets. The tree consists of **match**, **add** and **remove** nodes. Beginning with the root, which always is an empty **match** node, the tree is traversed.

A **match** node contains a set of regular expressions which are checked against the meta data set of the currently processed data point. If the expressions of a **match** node fit, the child nodes of this **match** node are evaluated. The **add** and **remove** nodes depict the respective action to be executed on the meta data. The combination of nodes of these three types allows the realization of transitive tagging relationships like those described above (see figure 5.9).

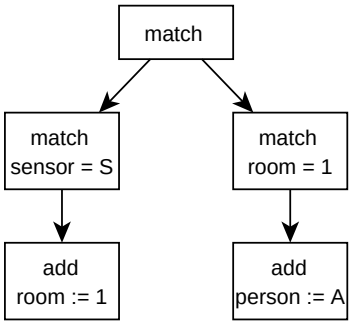

Figure 5.9: An example knowledge tree for assigning rooms and person identities to sensor ids. Applying this tree on a data point that has been tagged with sensor id **S** would also tag this point with room **1** and person **A**.

5.2.4 Flow graph

To provide a configurable and flexible processing environment, a model is required that fits the needs of defining and describing the path of a data point. This path ranges from the moment the data point is measured to the moment it is dropped or stored permanently.

There are some prerequisites for such a model: At first, it must contain entities that resemble the creation and finalization of data points, therefore the entry and exit points of the model. Between these there must be additional entities that each conduct a specified action on bypassing data points. All of these entities must then be connected in a relationship that models the forwarding of data points from one entity to another.

A very close-by representation of these prerequisites can be met by modelling the processing environment as a directed graph: Entities are expressed by nodes and the forwarding actions between entities by edges, directed from the source node to the destination node.

Entry nodes and exit nodes may only have outgoing and ingoing edges, respectively, and a graph may have one to many nodes of both types. All other nodes, however, may have an arbitrary number of edges. An edge represents a possible forwarding path, but whether a data point is forwarded along a specific edge or not is decided by the source node and, thus, depends on the processing function definition of that node.

As in forwarding graphs in computer networks, cycles in the graph must be strictly prohibited to prevent data points from looping forever and, thus, overexert the processing application.

As mentioned earlier, a set of node types has to be designed to meet all requirements of a data point processing architecture. The following sections will give an in-depth look in the definition of these nodes.

5.2.4.1 Stateless and stateful nodes

The differentiation between whether a node has a *state* or not embodies an important distinctive feature for the definition of processing nodes. In this case, the term *state* refers to whether the processing of a specific data point has further influence on the processing of subsequent data points.

For example, a node that simply counts the data points passing through would be labelled *stateful*, because the node would increase an internal counter variable every time a point

passes by. This counter variable represents the internal *state* of the node and is maintained between the processing of multiple data points.

By contrast, a node that e.g. increases the measured value of all data points by a constant value would, thus, be called *stateless*: Regardless of how many data points pass and in which order they pass by, neither the processing action nor any internal state depends on the processed data points.

5.2.4.2 Flows and flow states

Initially, individual measured data points do not have any relationship to one another other than a chronological order in time. Almost no relationship can be introduced until some background knowledge has lead to additional information (in the form of meta data) that implies such relationships.

Once such virtual relationships are established, e.g. by adding rooms or personal identities to the data points, some requirements must be met to ensure proper further processing of these data points. For example, when aggregating all values belonging to one room, values from other rooms may not be interweaved. Instead, the aggregator would have to aggregate for every room separately.

As a rule, stateless nodes usually do not have to take these dependencies into account. That is because these nodes can not distinguish between individual data points anyway.

In contrast, stateful nodes do have to differentiate between data points grouped into specified divisions. In general, the factor behind this subdivision must be configured for a processing unit in order to be able to obey these restrictions.

Usually, such a subdivision is maintained as far as the end, that is, the processing step where the data points are permanently stored. When accessing the data later, the subdivision can also be used to sort the stored data into groups belonging together. Henceforth, these subdivisions of streams of data points are called *flows*.

5.2.4.3 Entry node

Entry nodes represent the entrances of a graph. Entry nodes may only possess outgoing edges and do only forward data points along all these edges. That means, no alteration or filtering may be done.

Thus, the very simple processing function of entry nodes can be summarized in the following pseudo code:

```
1  function process(data point p)
2      for all e: edges
3          forward p on edge e;
4      end
5  end
```

As one can see from this code, no state information is used. Thus, all entry nodes are stateless.

5.2.4.4 Temporal aggregator node

A common task for data point processing is the aggregation of the real-time data over a time interval. For example, alongside the momentary energy consumption in a room, one could be interested in the aggregated amount of energy that has been consumed on a specific day.

Different mathematical aggregation functions are possible and the choice depends on the actual area of application. Thus, the node design can be split up into one generic part and one part that assembles the aggregation function itself.

Because the aggregation node must mathematically combine all data point values from the used interval, an internal state must be maintained that holds these values, until the next output value is computed and forwarded. This state is handled by the generic, function-independent part. For this, that part tracks the point in time where data points have arrived and accumulates these data points until a specified interval is over. As mentioned before, stateful nodes must make sure to preserve and separate flows from each other. To do that, an own state instance for every observed flow that passes the aggregator is created and stored:

```
1  function process(data point p)
2      get or create state for p;
3      add p to state;
4  end
```

Once an interval has passed, a resulting data value is created from the interval's state, an appropriate meta data representation is added and, finally, the new data point is forwarded along all outgoing edges:

```
1  function flush(state s)
2      if s.interval has expired
3          data point p = aggregate(s);
4
5          assemble meta data for p;
6
7          for all e: edges
8              forward p on edge e;
9          end
10     end
11 end
```

The second part of the node functionality is contained in the *aggregate()* function. Depending on the type of processed data values, a range of mathematical aggregation functions appear to be useful.

Sum and average

In some environments it is required to provide the sum or the average value of all incoming data values for a time interval. This can easily be computed by summing up all the data point values and then (in case of the average value computation) dividing this sum by the count of data points used up:

```
1  function aggregate(state s)
2      float value;
3
4      for all p: data points in s
5          value = value + value of p;
6      end
7
8      % The following line only appears in
9      % the average aggregation function:
10     value = value / number of points in s;
11
12     return data point with value;
13 end
```

Because sum and average both reduce a number of data values to one single value, the granularity and therefore the impact on the privacy may be greatly diminished.

Maximum and minimum

Storing maximum and/or minimum values instead of the real-time data may also greatly reduce the granularity of the data output. The following pseudo code illustrates the maximum case:

```
1  function aggregate(state s)
2      float value;
3
4      for all p: data points in s
5          value = max(value, value of p);
6      end
7
8      return data point with value;
9  end
```

The corresponding minimum *aggregate()* function uses *min()* instead of *max()*.

Integral over time

In many cases summing up the data points does not yield a useful result, because the physical background demands a sum over infinitesimal small slices of time. To aggregate such data value types, an integration approximation must be provided.

For example, to aggregate real-time consumption data values ($[P(t)] = [W]$) over time into energy ($W(t) = [kWh]$), an integral over the power function must be computed:

$$W(t) = \int_0^t P(\hat{t})\mathrm{d}\hat{t}$$

Because input values only appear at discrete points in time, no continuous integral can be computed. Instead, the integral is approximated by using a Riemann sum over all n data point values, weighted by the widths of the time slices:

$$W_R(t) = \sum_{i=0}^{n} P(t \hat{=} t_i)(t_i - t_{i-1})$$

This results in the following pseudo code:

```
1  function aggregate(state s)
2      float value;
3      data point last;
4
5      for all p: data points in s
6          value = value + (value of p * (timestamp of p - timestamp of last))
7      end
8
9      return data point with value and new unit;
10 end
```

As one can see in line 9, this node must take into account the possibility of a changed unit for the resulting value.

5.2.4.5 Spatial aggregator node

While the temporal aggregator node computes a combined data value from all the measured values in a time interval of a predetermined length, spatial aggregation means the combination of the real-time values for a spatial entity. This could be a room, a floor, or even a whole building. To make the spatial aggregator easily distinguishable from the temporal one, from now on it is named *combiner node*.

To generate a spatially combined value, this node has to aggregate the current values of all inputs grouped by spatial origin into one resulting value at every single point in time. Because actually the data values are supplied within discrete, variable time slots, real-time aggregation is not possible. Instead, the node must store the latest corresponding values

for all input entities and then aggregate them on-demand, once a new output value is being created. Usually, this is the case whenever one of the input devices yields a new value.

As with temporal aggregation, multiple aggregation functions are imaginable, like summation, minimum/maximum etc. Below the summation case is described; the design of other functions can be done correspondingly.

Like in the sections before, a new data value must be assigned to a state, as soon as it arrives. To differentiate between states, a meta data class must have been configured. Values from different states can then be processed without interfering with one another:

```
1  function process(data point p)
2      get or create state s for p;
3      add(s, p);
4  end
```

In contrast to temporal aggregation, a second configured meta data class is required to define the spatial domain to aggregate over.

For example, if the real-time sum of the consumption of a group of rooms is to be computed, the first class defines the group of rooms itself. Hence, for every group of rooms available, a single output flow is generated. In turn, the second class determines entities of the actual calculated sum. In this example case, this would have to be the room identification.

This second state information is contained in the outer state:

```
1  function add(state s, data point p)
2      get or create spatial domain d;
3
4      if p is newer than last value in d
5          add p to domain d;
6
7      forward(aggregate(s));
8  end
```

As one can see, an output data point is created whenever one of the inputs changes. This means that a combiner node yields a higher output frequency than the single input nodes.

At last, the actual aggregation must take place. In this case, a simple summation is used:

```
1  function aggregate(state s)
2      float value;
3
4      for all d: domains in s
5          value = value + d.value;
6      end
7
8      return data point with value;
9  end
```

5.2.4.6 Filter node

A common task in data value processing is filtering, to distinguish between wanted and unwanted data points. The filter node provides the functionality to define rules about which meta data information is required for a data point to be forwarded. Points that do not meet these requirements are dropped.

Because the rules are configured and loaded at startup of the application and after that remain static, no internal state is required. Therefore, contrary to the nodes described as yet, the filter node illustrates an example of a stateless node type.

The rules themselves are applied against the meta data information of the individual data points. For a data point to *fit* a filter, all rules must apply. Hence, the set of rules describes an **and**-relationship.

Every rule consists of two regular expressions: One for the meta data class name and one for the tag names (see section 5.2.3 for details).

Whenever a rule is about to be applied, at first the class name expression is checked against all available classes of the data point. If none of the class names matches, the data point is assumed to have failed the rule.

Otherwise, the tags of all matching classes are checked against the tag expression. If at least one class contains a tag matching the tag expression, the data point succeeds the rule and is forwarded. If not, the data point is dropped.

The following pseudo code demonstrates the behaviour of the node itself:

```
1  function process(data point p)
2      for all r: rules
3          if not matching(r, p)
4              return;
5          end
6      end
7
8      forward(p);
9  end
```

And this code shows the actual matching process:

```
1  function matching(rule r, data point p)
2      for all c: classes in p
3          if c matches r.class_expression
4              for all t: tags in c
5                  if t matches r.tag_expression
6                      return true;
7                  end
8              end
9          end
10     end
11
12     return false;
13 end
```

5.2.4.7 Dispatcher node

Another common task in the structure of a flow graph is the separating of flows into independent subflows. This separation also follows some rules that define which data point is forwarded along which edges.

The core behaviour of such a node closely resembles the behaviour of the filter node described above: Data points are checked against a set of rules and based on that a decision is made. In fact, the exactly same behaviour could be modelled by using multiple filter nodes as respective first nodes after a split up of a flow into multiple flows.

However, to simplify the modelling process, this action can be consolidated into one single node. The resulting node type is called *dispatcher node*.

In respect of the code, only few changes are required compared to the filter node:

```
1  function process(data point p)
2      for all e: edges
3          for all r: e.rules
4              if not matching(r, p)
5                  break;
6              end
7              forward(e, p);
8          end
9      end
10 end
```

The function *matching()* stays the same entirely.

5.2.4.8 Enhancer node

As described above, the term enhancing addresses the act of modifying the meta data information of data points in a desired manner.

Generally, an enhancer always does its work on the meta data set and then forwards the data point along all outgoing edges:

```
1  function process(data point p)
2      for all e: edges
3          enhance(p);
4          forward(e, p);
5      end
6  end
```

Enhancing can be done in a wide range of complexities.

Static enhancing

Static enhancing is done by using a set of match and modification rules defined at configuration time. These rules allow common, incomplex tasks like tagging every data point with one or more specified tag strings or removing such strings.

The latter may be very important regarding privacy-preservation: While some information in terms of meta data is required to determine the identities that should be able to decrypt the data points, once they have been written to a database, some of these meta data classes are no longer required later. These obsolete meta data classes could then be dropped using a well-defined rule.

This behaviour can be outlined as follows:

```
1  function enhance(data point p)
2      for all a: add_rules
3          apply a to p;
4      end
5
6      for all r: remove_rules
7          apply r to p;
8      end
9  end
```

Dynamic enhancing using knowledge

In section 5.2.3, the term *knowledge* and the design background of knowledge trees have been introduced. The *knowledge enhancer node* uses these structures to actually apply the stored knowledge to data points.

Because the actual enhancing work is implemented and done in the respective knowledge provider, this node can turn out very simple:

```
1  function enhance(data point p)
2      hand over p to knowledge provider;
3  end
```

The behaviour of the knowledge providers is described in greater detail in their respective sections.

5.2.4.9 Exit node

Exit nodes provide the functionality of releasing data points from the flow graph and processing them in a finalizing manner. This also means that an exit node may not have any outgoing edges.

To make available the functionality required by the concepts described in this chapter, two kinds of exit nodes have to be defined.

Remote exit node

In order to make flexible decentralization possible, the *remote exit node* provides the ability to forward data points along a virtual edge to a remote server's flow graph. Technically, this node uses a client implementation to connect to the remote server's interface.

```
1  function process(data point p)
2      connect to remote server s;
3      write(s, p);
4      disconnect(s);
5  end
```

Database exit node

If the processing and enhancing of a data point has completed, the data point must be written to a permanent storage. This is done by the *database exit node*.

This node implements the storage scheme described in section 5.2.2 in greater detail:

```
1  function process(data point p)
2      update identities from p;
3
4      writeToDatabase(renewIndices());
5
6      update block b;
7      add p to current block b;
8
9      if length of b > MAX
10         writeToDatabase(b);
11         clear b;
12     end
13 end
```

The function *renewIndices()* checks whether new index documents have to be generated (e.g. due to a change in covered identities or a time out) and, if so, returns a set of newly created ones. The currently active block is always referenced by all active indices, so a user can retrieve the block if and only if she is able to retrieve and decrypt the corresponding index document.

5.3 Data access

With the processed data stored in the database, there must be a possibility for users to access their data eventually.

The data access usually originates from a client application that creates a network connection to the server application in order to query for specific sets of data. Therefore, the server has to provide an interface that is remotely accessible.

This interface allows two kinds of queries: The query for index documents as well as for data blocks belonging to one specified reference id (see section 5.2.2 for details).

Figure 5.10 depicts an example of how such an access request is established. Usually, the user specifies a time interval to retrieve first. The server then delivers a list of index documents the user has access to. Once these index documents have been downloaded to the client, the client can use the user's private key to decrypt the index documents in order to obtain the reference ids as well as the AES keys for all blocks of data that are attached to the retrieved indices.

At last, the client has to query the server for all reference ids and then decrypt all blocks of data using the AES key.

Note that the server does not have any information about relationships between data values, so every kind of query (besides the specified time interval) that a typical storage server would be able to handle must be executed by the client, like querying for a specific flow.

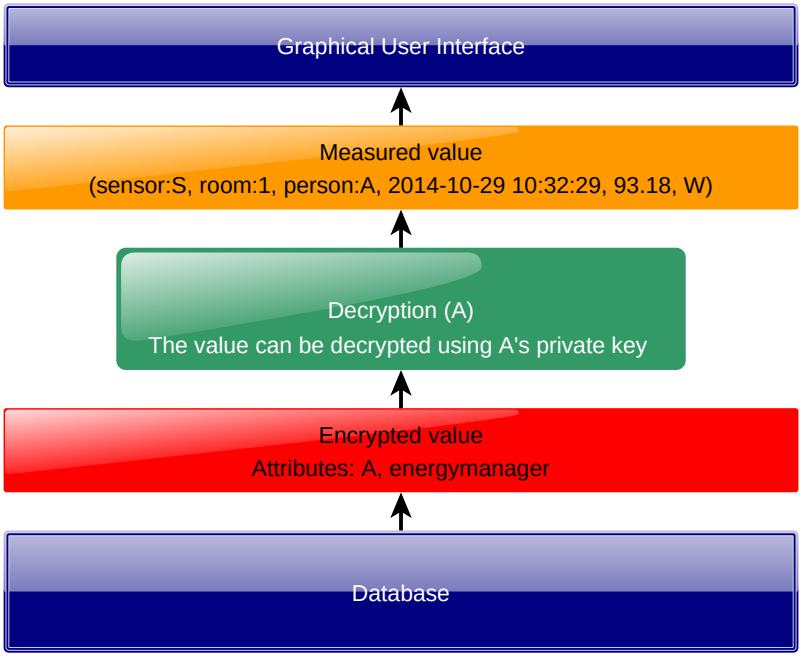

Figure 5.10: This figure shows a basic example on how a user can access processed data after it has been written to the database. Because the database itself only contains encrypted values, the user has to specify exactly, which sets of data she is interested in. Once the database provides the user with the encrypted values requested, the user can use her own private key to decrypt the data in her local machine and e.g. display the results in a graphical user interface.

6. Implementation

In the chapter above, various focus points of a privacy-friendly architecture were designed and described. Now, attention is turned to the actual implementation. The implemented architecture is called *Privacy-Preserving Post-Processing and Storage* (**P4S**). Hereafter, the server application is referred to by the term *p4s* while other modules are called by their respective names once they are introduced.

6.1 Architecture and implementation overview

Because the architecture inherits several features of a classical client-server-architecture, at least one client has to be implemented along with the server. For this architecture approach in particular, there exist two kinds of clients for which a dedicated client library and corresponding interfaces on the server have to be implemented:

The first type of client (called *p4s-client-input*) allows to send data points to a server for processing. One purpose of this library is to provide the interface between two or more server applications for decentralized processing flows. Every server that wants to forward data points to another server instance can make use of this library.

However, pure client applications may exist as well. An application that reads values from a measurement device and injects them to a server instance provides an example of the purpose of such a client.

The second type of client (called *p4s-client-access*) in turn provides an interface for data access by client applications. This client library encapsulates the whole process of retrieving index and block documents from the server and decrypting these. Ideally, when implementing an access client application, the developer should be able to query for data points in a specified time interval without taking notice of the special properties of this privacy-friendly approach.

The server itself as well as the access client require access to various functions of the CP-ABE toolkit (see section 5.2.2 for details). Because at the present moment the existing implementations of CP-ABE in C (see [BeSW07]) and Java (see [Wang12]) are lacking some of these required functions, a wrapper module called *jcpabe* has been created for this work that provides a unified interface including some utility functions.

Functionality that has to be shared between modules has also been consolidated into another extra module called *p4s-common*. This module contains the implementation of

the whole data model described in the chapter before as well as a set of globally required utility functions.

The main purpose of the *p4s-client-access* client library is to build actual client applications. As part of this work, two Graphical User Interfaces (GUI) were created that take usage of the library: A standalone desktop application (called *p4s-gui-desktop*) as well as an experimental app for Android smartphones and tablets (called *p4s-gui-android*).

Both GUI applications are based on yet another module, called *p4s-gui-common*. This module acts as a code base for applications providing data access in terms of visual representations, like charts. By using such a base module, implementing another graphical interface is reduced to porting the device-dependent code to the target platform.

At last, two small tools are part of the implemented architecture:

p4s-key-tool provides a command-line interface for managing the CP-ABE key infrastructure. This includes the creation of such an infrastructure as well as the creation of private keys.

p4s-key-user allows to execute maintenance tasks on the user repository used by the server application. The current version supports the creation and deletion of users and the modification of user permissions.

As an overview, figure 6.1 shows all of these modules as well as their mutual relationships.

6.2 Development environment overview

To be interoperable with other components of the IDEM project, Java was chosen as target platform. Therefore, all modules can easily be deployed on any system supporting Java. While the client and utility libraries are compatible with all Java versions since version 7, the server application makes use of features introduced with Java 8 and hence only works if a recent version of Java is installed. In order for the CP-ABE part to be usable, the *Java Cryptography Extension* (JCE) has to be installed, too.

All development and testing has been done on an Ubuntu 14.04 LTS system using Maven 3.2.3 and the JDK 8 from Oracle. Other operating systems should also be supported, though, because the implementation uses no specifically device- or system-dependent functionality.

All required code is contained in the Git repository belonging to this work. The repository also provides a set of utility scripts for building and deploying parts of the application (see appendix A for details).

6.3 Server

The server application is implemented as a flexible application that is designed to run in the background and process requests from remote clients.

This application itself is heavily based on the Spring framework (see [Inc.b]). Spring Boot is used for bootstrapping the application and configuring all other parts that make use of Spring modules. Spring Core provides Dependency Injection for bootstrapping the application controller objects (see 6.3.4). Spring Web manages the web server sockets and the corresponding RESTful interfaces for data input and data access. At last, Spring Security provides the security layer used for enforcing access control on all remotely accessible interfaces.

The application *p4s* supports three run modes: Standalone application mode, UNIX daemon mode and application server mode.

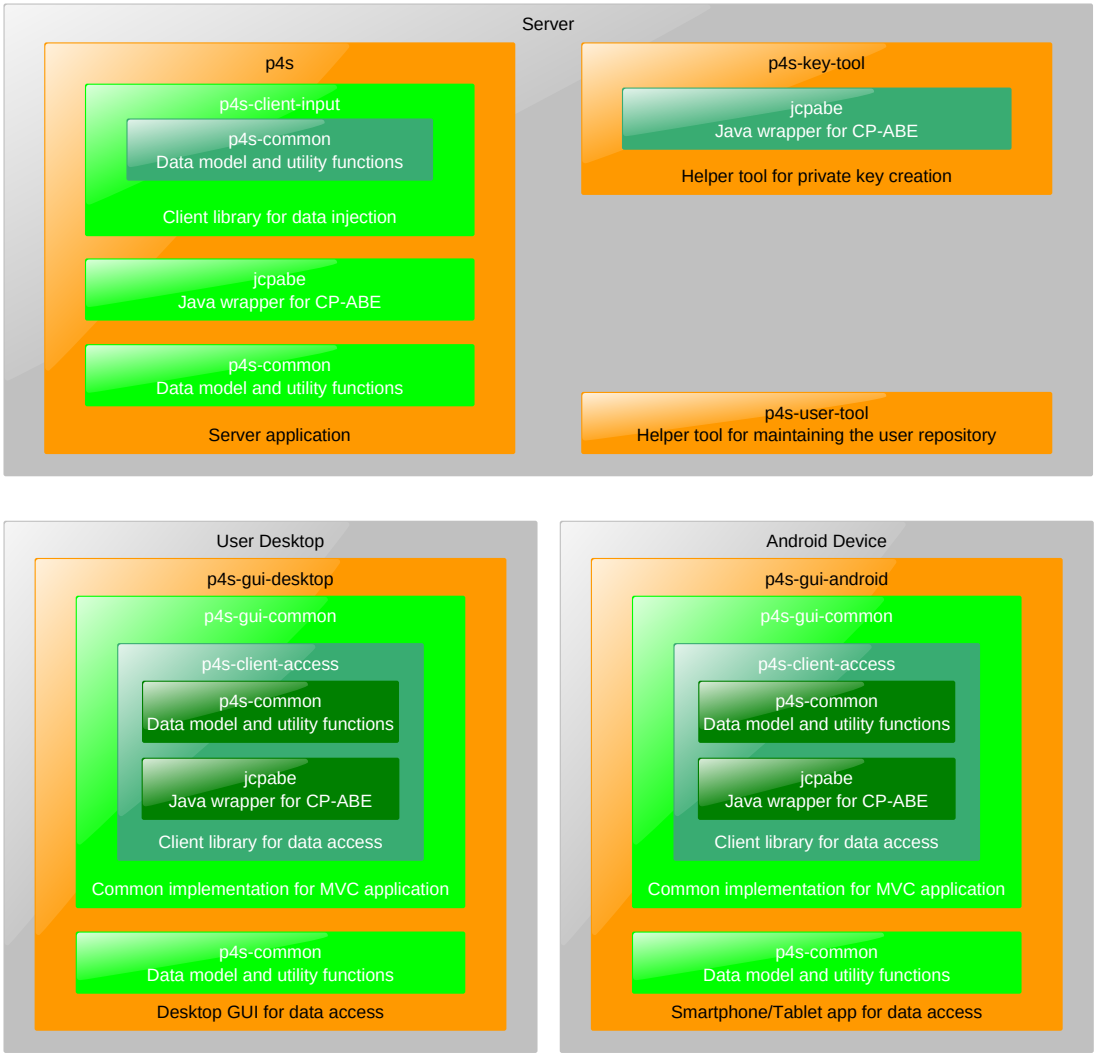

Figure 6.1: A summary of all modules developed for this work as well as their relationships.

In standalone application mode, *p4s* simply acts as a standalone console program. Once started, it listens on the configured ports and processes incoming HTTP requests. Logging output is written to standard output. This mode can be used when running *p4s* for testing or for starting it from out of a non-root user's session.

In daemon mode, the application is started as a background process. The type of process depends on the operating system. Apache Commons Daemon is used to handle the operating system dependent part of starting the application as a daemon, like setting an unprivileged user to run the executable (see [Founa]). Currently, Windows and UNIX-like platforms are supported. When running in daemon mode, the logging output is always redirected to a log file. Starting and stopping of the application is done using the utilities provided by the operating system.

Before running in application server mode, a Web Archive (WAR) file has to be created from the application packet. This file can the be deployed to an arbitrary Java application server providing support for Java servlets.

These three modes only differ in the way of bootstrapping. Once the application has been started, the functionality provided is exactly the same.

6.3.1 Flow graphs

As mentioned in the chapter before, the processing setup inside a server instance can preferably be constituted by a graph. This graph must be provided as kind of a configuration file to the server. Instead of reinventing the wheel by specifying an extra file format for this, GraphML is used.

GraphML is a XML-based markup language for specifying graphs by providing a set of nodes, as well as edges as interconnection information (see [BEHH+02] for details about the specification). Every entity (node or edge) can have a number of properties (like strings, numbers, etc.) specified. There exist a number of production-ready parser-libraries, with the result that loading and parsing GraphML files in an own application states almost no problem and is done in a few lines of code.

The structure of the processing graph can be mapped 1:1 to the GraphML format, with nodes depicting processing steps (like aggregation etc.) and edges depicting the forwarding between two such steps. While GraphML itself does not specify any properties regarding the displaying of graphs (node position, size, color, etc.), third-party editors usually specify their own namespace for saving these kind of information.

However, GraphML does not have a consistent concept of a node type[1]. Additionally, specific nodes may have to be configured individually (e.g. a time interval to aggregate over). Both of these information types are mapped to node properties. In order not to interfere with the properties of other tools (like the used graph editor), all configuration properties are equipped with the prefix **p4s**.

The GraphML configuration file is specified as one of the configuration files for *p4s*. At application startup, the file is loaded and parsed. The configuration reader iterates over all nodes and edges in the GraphML graph and generates an isomorph structure from node objects. These objects are instances of classes extending the abstract base class **Node** and contained in the Java package **de.tum.in.net.p4s.server.core.processing**. Each type of node described in the chapter before has its own implementing class.

As an example, figure 6.2 shows a visual representation of the graph file actually used at the test setup at the Chair for Network Architectures and Services at TU München. This

[1] GraphML does support embedding XML content, but most graphical editor tools do not support proper editing and saving of such files.

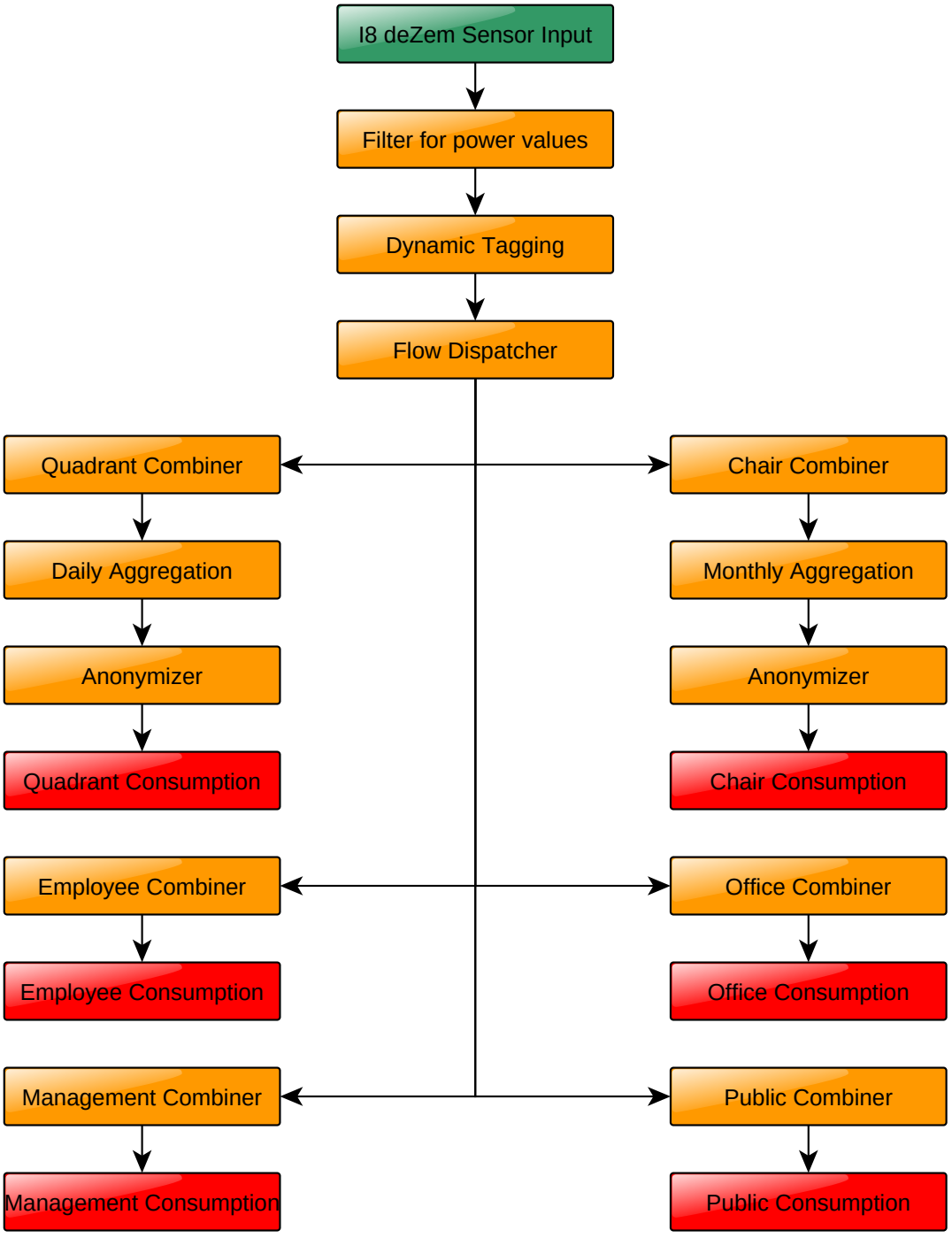

Figure 6.2: An example for a processing graph. This figure has been directly exported as a graphics file from the actual configuration GraphML file. Thus, by using GraphML, processing graphs can easily be built in an arbitrary graph editor. Entry nodes are colored green, exit nodes red and all other node types orange.

graph has been created with the yEd Graph Editor and demonstrates the great advantage of using GraphML: There is no need for either editing complex graphs in a text editor or developing an own, third-party graph editor. The graph can easily be edited in an arbitrary GraphML editor and used as a configuration file as well as exported to a graphic file for demonstration purposes (like in this case). This also serves the purpose of demonstrating the privacy-friendly quality of a system configuration, like mentioned in chapter 4. The example graph itself is described in greater detail in the next chapter.

To circumvent duplicated code, some nodes with similar functionality share an additional own abstract ancestor. The class **Aggregator** provides the management of states for different flows and time interval and contains two abstract methods. The first abstract function **process()** is called every time a new value arrives, and takes this value as well as the state it belongs to. The second abstract function **aggregate()** is called with a state as parameter, whenever the current time interval has finished. The sub classes **AverageAggregator**, **MinAggregator**, **MaxAggregator**, **SumAggregator** and **IntegralAggregator** implement these methods by using their own specific functionality. Other additional abstract sub classes of **Node** are **Exit** for graph exit nodes and **Enhancer** for all node types that enhance the meta data information of data points. Diagram 6.3 depicts an overview of the node type classes contained in the standard application package.

The base class **Node** features the two methods **getRequiredProperties()** and **getAllowedProperties()**. These methods return a set of properties that are required for a node class, or allowed, respectively. While loading the graph at boot time, it is verified that these requirements are satisfied. If an error occurs, the loading process is aborted.

Node itself has two required properties, that therefore must be provided for every node instantiated in the graph file: **p4s.class** and **p4s.id**. **p4s.class** contains the qualified name of the Java class to instantiate[2]. **p4s.id** holds an id that must be unique in the graph configuration. This id may differ from the id entity of GraphML itself. Sub classes of **Node** may specify a variation of additional properties. For details see the documentation of these classes.

While the set of implemented classes already covers a wide range of applications, there may still be situations where a more specialized node type is needed. Because node classes are instantiated by their qualified class names, own implementations of specialized node types can easily be added to the architecture and loaded by specifying their class names. To do this, the class must be contained in the Java classpath of the runtime environment at application startup.

6.3.2 Knowledge providers

As mentioned in chapter 5, the server application uses a tree structure as a representation of knowledge information. Because knowledge may originate from many different sources, the application should provide an interface for implementing different approaches and easily switching between them.

Because of this, the package **de.tum.in.net.p4s.server.core.controller** contains an interface called **KnowledgeProvider** that can be implemented by a class to add such a knowledge source to the server. The current version of P4S provides one built-in class: **XmlKnowledgeProvider** uses a direct representation of the model described in the chapter before to load a static knowledge tree from a file. While this does not provide any dynamic source, it serves its purpose as a simple solution that can be replaced once e.g. an interface to the knowledge plane has been established that is also part of the IDEM

[2] e.g. **de.tum.in.net.p4s.server.core.processing.Combiner**

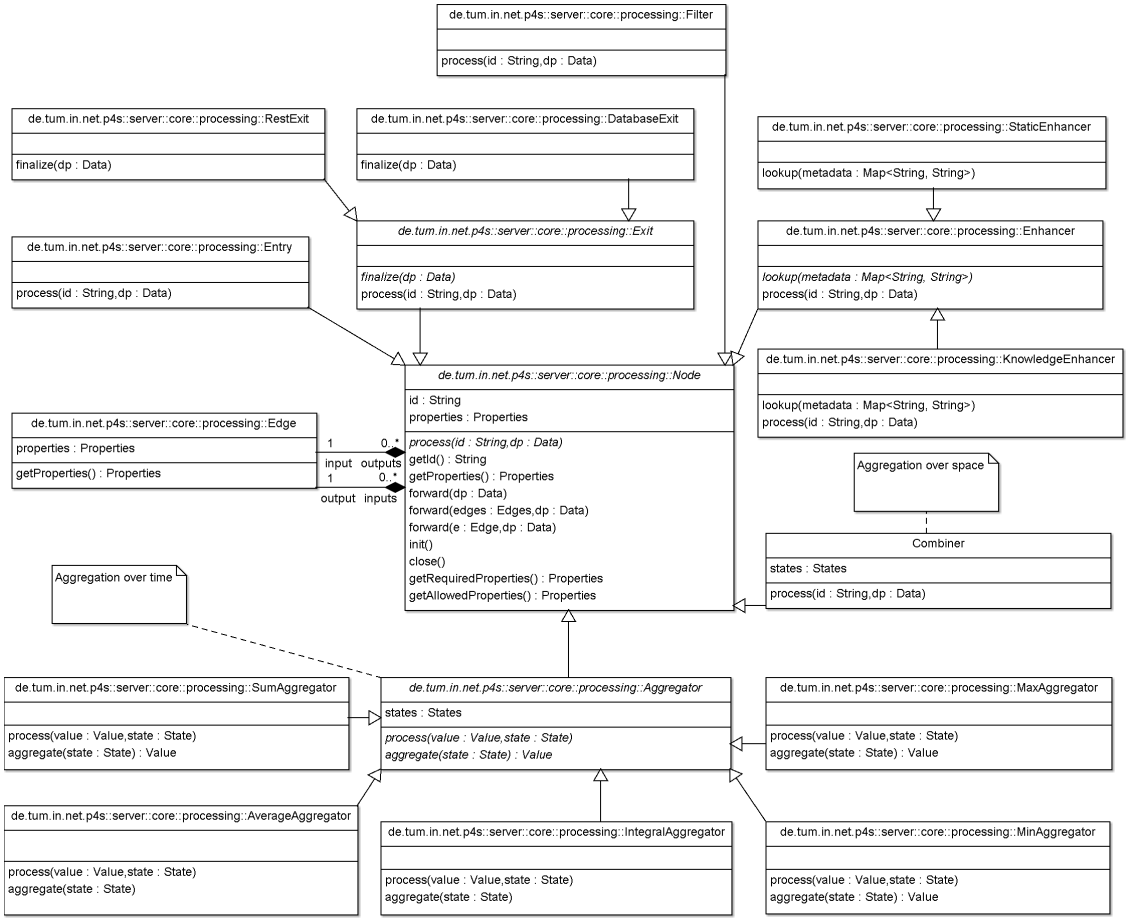

Figure 6.3: This UML diagram shows the classes and relationships contained in the Java package **de.tum.in.net.p4s.server.core.processing**. Note that some methods have been hidden in order to make the diagram more clear (for example, the **getRequiredProperties()** and **getAllowedProperties()** methods in all sub classes).

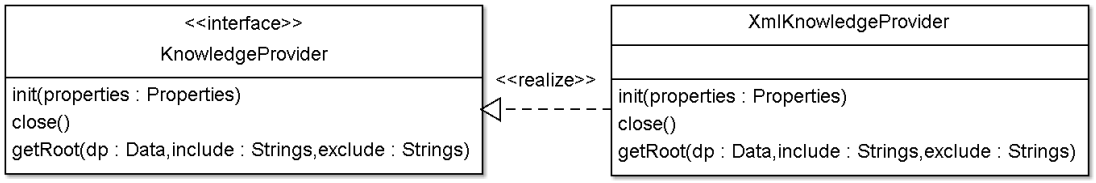

Figure 6.4: The UML diagram demonstrating the interface for providing dynamic knowledge information. The current version of P4S only supports a pseudo-dynamic provider that reads a knowledge tree from a XML file.

project. An example knowledge XML file is included in the standard distribution packets generated by the helper scripts in the Git repository (see appendix A for details).

Figure 6.4 shows an UML diagram featuring the mentioned interface as well as its built-in implementation. The implemented class to load is specified in the configuration file of the server application. At start up, this class is instantiated and provisioned with a set of properties optionally stated in the configuration file (for example, **XmlKnowledgeProvider** takes the name of the XML file containing the knowledge tree as property). For a class to be loaded, it must be on the Java classpath.

The method **getRoot()** takes a data point, an include and an exclude list and generates a knowledge tree that is applied to the data point by the knowledge controller subsequently. An implementing class may enhance the meta data indirectly with this returned knowledge tree as well as directly using the reference to the data point. The latter is of particular interest for dynamically changing properties like unit or value, because knowledge trees do not support the modification of these.

The two provided lists allow to restrict all enhancement to a specified subset of meta data classes by either blacklisting or whitelisting classes. This comes in useful when classes may not be modified in any case.

6.3.3 Database, crypto and user backend

The current version of the server architecture implementation supports CP-ABE as asymmetric crypto system and CouchDB (see [Founb] for details) as database storage backend. However, the architecture uses the concept of context interfaces to be able to load additional implementations without changing the original code.

Internally, every entity that seeks access to crypto or database functionality is provided with a context object that offers appropriate methods. This context object is of the type **DatabaseContext** or **CryptoContext**, respectively. Both are interfaces that can be implemented by third-party classes and loaded by placing these classes on the classpath of the server application and setting up the server configuration file accordingly. Thus, the server does not depend on one specific method. For example, a storage backend that saves the encrypted documents as common files on the hard disk could easily be implemented.

The access control management does not use this context concept indeed, but the way of loading third-party implementations is very similar. As contrasted to the crypto and database backend, the user backend does not have a context. Instead, one single instance of the class implementing the **RoleHandler** interface is created, that is used globally.

Detailed information about the interfaces can be found in the Javadoc documentation that is contained in the Git repository or can be generated from the source code.

6.3.4 Controllers

The main structure of the server implementation consists of a set of singleton classes called *controllers*. Each of these controller components manages a specific scope of the server and provides a public interface for other components. Figure 6.5 shows an overview of all controller components the server architecture consists of, as well as the REST interfaces for remote access.

These components can be categorized in three isolated realms (realized by using different Java packages): Common (in the diagram: grey), processing (green) and access (blue). The common part offers functionality that is used across all other components. For example, both processing and data access need to have access to the database using the **DatabaseController** instance. Hence, the other two parts both depend on the common part, but not mutually on themselves.

The class **ConfigController** holds the configuration of the loaded environment and provides default values for non-set properties. The configuration itself is read by a class entity called **ConfigReader** that is invoked at application startup.

The class **TimerController** globally manages the execution of periodic tasks by other controllers. For example, node classes can implement a method called **timer()** that is called on every timer cycle.

The class **StatisticsController** can be activated by a switch in the configuration file. Once activated, it maintains a map of counters that can be used to measure the performance of various parts of the architecture. The configuration mechanism is designed such that a disabled statistics mechanism does not have any impact on the performance.

Because these three controllers provide very common functionality, multiples of the other controllers have dependencies on them. These are not shown in figure 6.5.

The classes **RoleController**, **DatabaseController**, **CryptoController** and **KnowledgeController** all manage instances of their respective actual implementation. This means that they do not contain any code that directly provides functionality. Instead, all calls are forwarded to specialized classes implementing the offered interfaces.

The class **FlowController** contains the core part of this architecture: the management logic for processing data points. **FlowController** contains the representation of the flow graph, composed of instantiated sub classes of the abstract **Node** class. Once the graph has been read from a GraphML file and the application is ready for work, **FlowController** passes incoming data points to the specified entry nodes.

FlowController is also responsible for the lifecycle of the processing graph. While the designed initialization process makes no assertions regarding order of node initialization, the finalization process has to be ordered. That is because nodes may want to flush their internal state prior to finalization. If a stateful node would be finalized when its successors have already been closed, this state would be lost irretrievably. To do this, **FlowController** uses a simple algorithm that manages a queue of yet to be closed nodes. For every node it is checked if all predecessors have been closed. If so, the node is also closed and removed from the queue. If not, the node is once again added at the end of the queue.

At last, the class **AccessController** provides methods for accessing encrypted data sets using time intervals. The methods offer the possibility for a client to request data sets that can be read locally using a private key. The server, however, does not have this secret and, therefore, does only have a very basic view on the data.

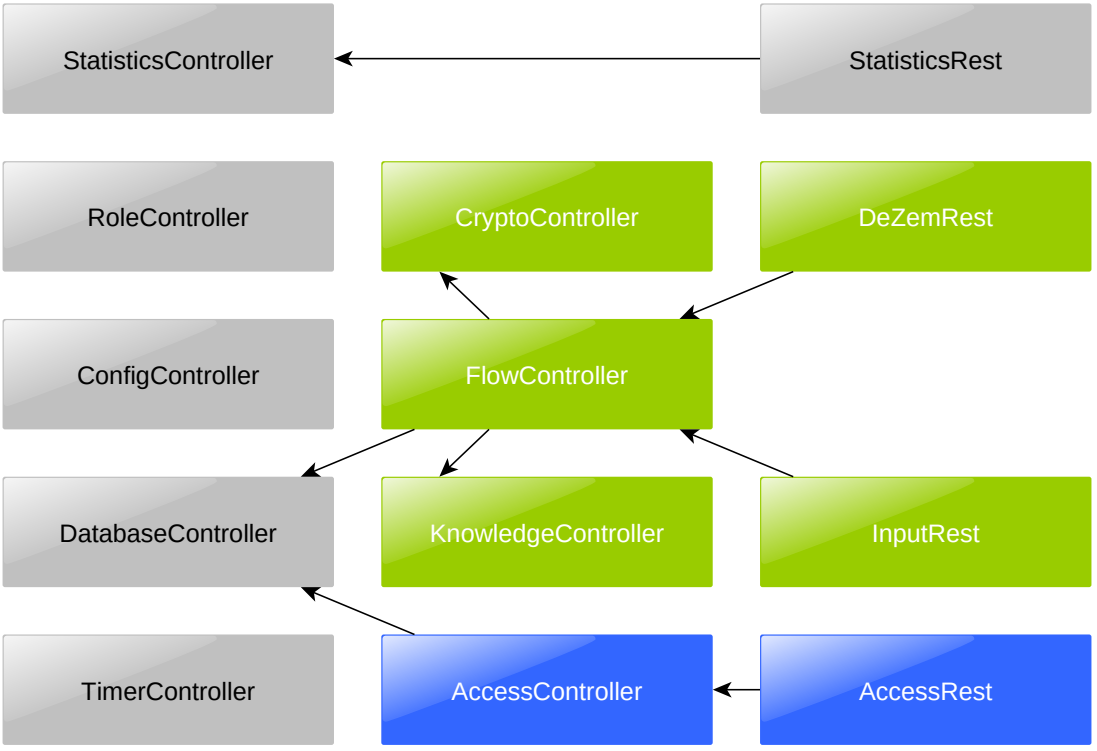

Figure 6.5: This figure shows the controllers the server application consists of. Dependency relationships (depends-on) are depicted by arrows and the realm membership is depicted by colors. Note that some dependencies have been omitted for clarity.

6.3.5 RESTful controllers

The public interface of the server application consists of a set of controllers providing a RESTful interface that can be accessed over HTTP or HTTPS. These controllers usually do not contain any functionality going beyond calling functions in other controllers. This dependency is also shown in figure 6.5.

All these controllers use JSON as transport format. The Java classes that contain the various parts of the data model are marshaled at the one side and unmarshaled at the other side. For details on the data model and the available request mappings provided by these RESTful controllers, see the Javadoc documentation.

The class **StatisticsRest** provides the information collected by the **StatisticsController**. This can be used by client programs that run benchmark tests on the server application as well as for debugging purposes. If collecting statistics information is disabled, this RESTful interface is not available. Note that user accounts do need to have a special permission switch set in order to be qualified for remotely retrieving statistics information.

The controllers **DeZemRest** and **InputRest** both accept data points from remote sources and inject them into the processing graph. While **InputRest** offers a self-defined interface that can be accessed by the input client library described later, **DeZemRest** provides an interface that is designed for receiving data points from a deZem energy logger device or another device that processes data points of this type.

At last, the **AccessRest** class provides the remote interface for data access. The methods provided are essentially the same ones as offered by **AccessController**.

6.4 Client libraries

While the remote interfaces offered by the server already provide an easy possibility for a client application to access server functionality, a pre-implemented client library further simplifies this issue by providing Java classes that can be used directly without setting up an own HTTP client.

The project contains two ready-to-use client libraries: *p4s-client-input* and *p4s-client-access*. Both client modules are meant to be used as a dependency package for developing client applications. However, both include a simple command-line client application, that runs, if the JAR file is started directly. These applications are meant for debugging and testing purposes primarily. Nevertheless, both of them are feature-complete and could even be used for scripting etc. in production environments.

6.4.1 Input client

The input client library contains the class **InputClient**, offering the method **process()** as well as some common client functionality for connecting and disconnecting. The class only supports synchronous calls.

To be able to forward data points to other server instances, the server module itself depends on *p4s-client-input*. Custom client applications could implement functionality that imitates the server behaviour by e.g. post-processing data points that have already been written to the database.

6.4.2 Access client

The class **AccessClient** offers the method **query()** that allows to query the server for data points in the specified time interval. Further querying and filtering has to be done by the client application itself. Thus, *p4s-client-access* provides a very basic means of synchronous data access. However, **AccessClient** supports callbacks for status reports, so asynchronicity can be added by the surrounding implementation code. *p4s-client-access* is used by *p4s-gui-common* that is described in the next section.

6.5 Graphical User Interfaces

While the access client library already contains a small command-line interface, a typical user might prefer a graphical application that is able to display a visual representation of the data points, like, for example, a diagram plot.

This project contains a module called *p4s-gui-common* that offers the basic functionality for retrieving, caching and managing sets of data. However, it does not contain any code that actually displays anything. Instead, it provides interfaces that can then be implemented to assemble a complete application. This allows to quickly build an application for a target platform without worrying about the target independent code. Currently there are two supported targets: The standalone desktop application *p4s-gui-desktop* for Windows, Linux and Mac OS X and the Android app *p4s-gui-android*.

p4s-gui-common provides Java interfaces that have to be implemented by applications: **MainView**, **ConnectionView**, **PasswordView** and **ErrorView**. These interfaces represent the parts of the graphical user interface for data display, choosing a server, entering authentication information and displaying error messages respectively. The implementations of these interfaces have to call methods in the controller objects whenever the user executes an action. The controller objects of *p4s-gui-common* then execute these tasks asynchronously and inform about updates via a callback.

Because the access client library does not implement any caching or queueing mechanism, *p4s-gui-common* adds an in-memory cache that supports querying for flows and meta data tags besides time ranges.

6.5.1 Desktop

The module *p4s-gui-desktop* allows to display plots of flows within arbitrary time intervals. This application is implemented using the SWT library for the GUI elements. This restricts the use of the application to platform targets supported by this library, but provides a native look and feel on every supported platform.

The module is deployed as an executable JAR file containing all dependencies, so the application can easily be installed and executed. A private key as well as the public key of the key infrastructure must be provided at start up.

6.5.2 Android

The module *p4s-gui-android* contains an app for Android smartphones and tablets. It provides essentially the same functionality as the desktop variant, but uses the default Android widget toolkit to display the graphical user interface.

Like the desktop application, both a private key and the infrastructure's public key must be provided as files on the SD card. The app itself is built as an APK file and can easily be installed on any Android device running Android 4.x or higher.

6.6 Deployment

As an architecture that supports the distribution of components on a number of devices, special care must be taken upon making the installation process as easy as possible. Especially the automatic deployment of virtual machines is a feature that allows the rapid assembling of environments with a high number of processing entities.

To achieve this, administrators and users must be equipped with methods to set up a working environment with the least possible effort. This environment must include a well-supported operating system, preferably with Long Term Support, the Java 8 runtime, a database server and finally a reasonable default configuration.

The Git repository contains a set of script files that can assemble such environments. To cover a wide field of application, these scripts are able to create different types of deployment units: Currently packages for Debian distributions, docker images and Virtual Machine images are supported. The respective scripts share as much code as possible, so building an additional script for yet another deployment target can be done very easily.

All deployment methods are based on Ubuntu Server 14.04 LTS. This offers a very stable platform that will be supported for some more years at the present point in time. It also offers a wide range of included or installable packages, so almost all dependencies can be installed and kept up-to-date using the built-in packaging tool APT. This is a very important point for a security-critical device.

Currently, all scripts build deployment packages including all components of the architecture (except the validation and testing tools discussed in the respective chapter), but do not include any desktop environment. To deploy a desktop device for *p4s-gui-desktop*, the X server and desktop packages have to be installed manually.

The deployment scripts create a user and group pair named *p4s* to run as, as well as an init.d script for automatically starting the server application as a daemon at system boot. However, this automatic start of the daemon is disabled by default and must be activated by the user once. Additionally, a CouchDB server is installed and configured properly by creating a user called *p4s* and three databases including the required views.

As a default configuration the example files contained in the **Source** folder are copied to the deployment packages. To provide an own set of configuration files, one could either edit these files before creating a package, or edit the files once the package has been created. The approach differs slightly between the provided deployment strategies because of the difference in the ease of manipulating an already assembled package.

All methods are based on third-party tools to create the bundled packages. These tools usually are very mighty, so the provided scripts can easily be extended to provide even more functionality or do very specific tasks that are required for deployment in a situation.

6.6.1 Debian package

This script creates a DEB package file that can be installed on any recent Debian-based distribution. All other packages required are marked as dependencies and installed by APT, the Debian package manager. If installed on a desktop operating system, the desktop GUI can be used right after installation.

Besides the simple installation on an already running system platform, this deployment method also supports to easily remove the application once it is no longer required on a system. This comes in handy when using the desktop application *p4s-gui-desktop*.

6.6.2 Docker

Docker is an environment for operating system containers. These containers provide a lightweight method of separating pseudo operating systems from the host system. While this could be used for isolation out of security reasons, the real-world implementations have had some trouble with isolation outbreaks in the recent time, so classical virtual machines still seem to be the better choice when aiming for that target. However, with major cloud platforms like OpenStack providing support for Docker, it would be waste of potential not to support the platform (see [Inc.a] for details).

The repository contains a so called *Dockerfile* that can be used to assemble a Ubuntu 14.04 LTS based ready-to-use docker image. This image can then be deployed or extended further.

6.6.3 Virtual Machine images

Besides the creation of Docker images and containers, there also exists a script for building classical hard disk image files that can be used by Virtual Machines. The script uses utilities provided by the libvirt-tools (see [lTea] for details) and the QEMU environment to create a QCOW2 image file containing an Ubuntu server system with all required applications installed. This file format is used by the QEMU environment for virtual hard disks and can also be loaded by XEN-based hypervisors without further ado.

Some other virtualization platforms do not support the QCOW2 format. However, QEMU contains a set of tools that allow the conversion of QCOW2 files to many other hard disk image formats like the ones used by VMware, VirtualBox or HyperV. Because of that, QCOW2 acts as a good intermediate point for creating deployment packages for the required virtualization target in almost every situation.

7. Evaluation

The preceding chapter introduced the implementation of the architecture. Now, this chapter is focused on analyzing and evaluating this implementation. In this process the quality of the privacy-preservation is of particular interest, of course.

7.1 Evaluation environment

The evaluation was done at the Chair for Network Architectures and Services at TU München. An energy logger device from the German company deZem was installed at the chair, so the architecture could actually have access to real-world, real-time measurements. The logger had access to a number of supply lines. However, some rooms shared one supply line, so measurements would depict the energy consumption of multiple rooms. This was acceptable in a test environment, but in a production environment every room should have its own supply line measured. Otherwise, one always has the problem to decide whether to hide the personal information from the employees or always reveal the consumption of all employees affected by one supply line.

The server application itself was installed on a test system running Ubuntu Server 14.04 LTS. The system was equipped with an Intel Core i5 processor with 2.50 GHz and 4 GB memory. A similar system was used to run the client applications.

The server application was configured to use the processing graph shown in figure 7.1. This graph has been created for the setup at the chair using requirements defined by the supervisors of this work and aims to be fully usable in the use case at the chair.

The graph takes measurements from the one logger device, processes them and finally forwards them to various exit nodes. As a first step in the processing chain, data points that do not contain power values are dropped[1].

Then the current knowledge tree is applied. A static XML file was used as knowledge provider. This XML file contained mappings from the sensor ids to rooms, from rooms to persons and from persons to identities. Additionally, rooms were tagged with one of the predicates *office*, *management* or *public* that was used to specify the information privacy level per room. After that, the newly added meta data can be used to dispatch the data points into one or more categories for further processing.

[1] The deZem logger generates power, energy and phase angle measurement values.

If the room type *office* predicate is set, the data points are forwarded to the employee chain. There, the consumption of the individual employees is grouped (combination over the *person* meta data class). The results are forwarded to the employee database exit and stored in the database. A similar output chain exists for offices (predicate *office* as well), for public rooms (predicate *public*, e.g. the kitchen) and management rooms (predicate *management* e.g. server rooms). All of these output chains only process real-time data, so no aggregation over time is done.

The other two chains use aggregation over time to provide measurements over a specified time interval. Both use an integral aggregator to yield a summarizing value in kWh. For the individual quadrants[2] the energy is aggregated on a daily base, so one day is chosen as the interval for the aggregator. The chair chain summarizes the energy consumption of all rooms of the chair and aggregates over a whole month. An Anonymizer[3] is used to make sure no person-related information remains in the resulting data points.

It has to be noted that the configuration described here does not contain multiple machines, so it does not correspond to the *ideal* solution with multiple domains of varying information density. However, because the setup only contained one measurement device, another tier of processing would not have much influence. Nevertheless, one could always forward the two time aggregated flows to another processing device to add a new tier.

7.2 Code validation

At first, it must be verified that every component of the architecture behaves in exactly the way it is supposed to behave. Besides running the final product and testing whether it reacts sanely, Unit Tests usually are the appropriate means to achieve that goal. A Unit Test is used for testing whether single units of an application work correctly by e.g. feeding specific input and verifying that the output is as expected.

In the development of this architecture, Unit Tests have been written for almost all modules. Only the two platform-specific GUI modules *p4s-gui-desktop* and *p4s-gui-android* do not have any unit tests attached[4]. By using Maven as build automation tool, Unit Tests are automatically run on every build of the project.

The main focus was on making sure that the processing part works as expected. To do this, a Unit Test was written for every type of processing node. These Unit Tests create a simple processing graph consisting of the node under test as well as special testing nodes as needed (e.g. a special exit node that keeps track of any data points passing it).

According to the EclEmma plugin for Eclipse, all modules achieve a high test coverage (above 90 percent). Additionally, there exist Integration Tests besides the classical Unit Tests that verify whether the combination of multiple individual units behaves as expected.

At last, some additional tools like FindBugs and Checkstyle were used to automatically analyze the code and find potential error sources.

7.3 Performance

As an architecture that has to process real-time measurement data, it is of crucial importance that the incoming data points are processed in a way such that the available processing capacity provided by the device's hardware is utilized properly. Note that this

[2] A set of rooms arranged next to each other.

[3] A static enhancer node that removes all of the specified meta data classes.

[4] However, the shared code in *p4s-gui-common* is tested.

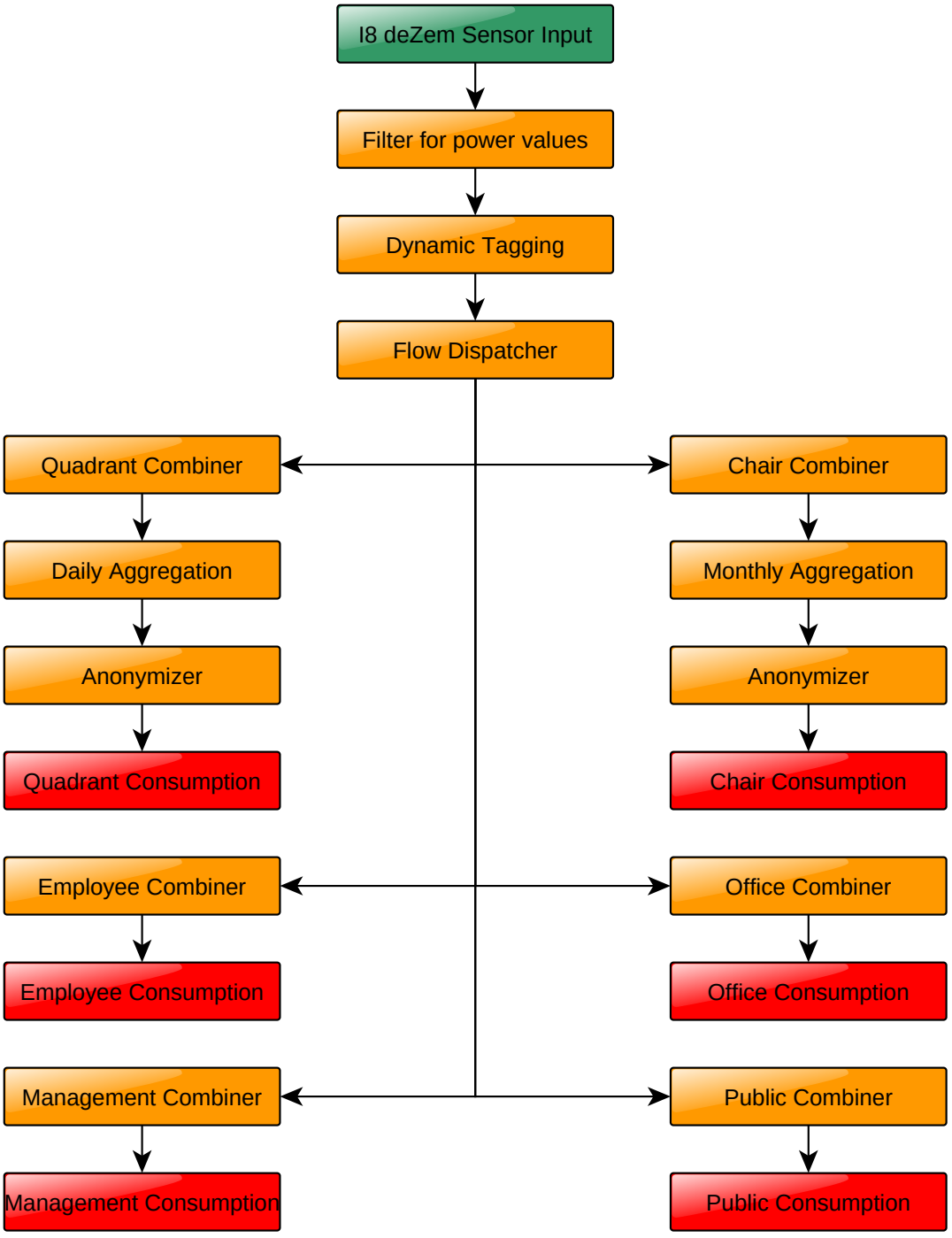

Figure 7.1: The processing graph used for the test setup at the Chair for Network Architectures and Services at TU München. Entry nodes are colored in green, exit nodes in red and all other nodes in orange.

almost only applies to the server architecture, because the client traditionally does only one action at a time.

That means that the server architecture in particular must be parallelizable on a very high degree. As an additional requirement this also implies that a load that is higher than the maximum load the device can handle does not break the system. The ideal behaviour allows to linearly reach a load point where the number of processed data points per time unit stagnates. When increasing the load further, this processing indicator may not begin to drop, because this would allow to e.g. execute Denial of Service (DOS) attacks by exhausting the server's processing capacities.

7.3.1 Requirements considerations

Because the original requirement for the server application stated that it should be able to handle "millions of measurement values per hour" when necessary, the performance tests were conducted with a goal of ~1000 data points per second in mind.

To be able to conduct performance benchmarks on the server application, a simple benchmarking tool called *p4s-burner* was written. The program takes the length of a block of data and the number of blocks per thread and second as parameters and gradually spawns new threads. The input client library is used to send these blocks to a running server application. Simultaneously *p4s-burner* continuously retrieves the current map of statistics information from the server and writes these statistics to a data file that can e.g. be plotted using Gnuplot.

Because it had been expected that the performance would be very dependent on the used block size, additional consideration had to be done to choose an appropriate number. As almost all entities that would send data points to other instances are using buffers, a medium block size of 40 data points per block was considered a good choice. Therefore, all tests were conducted using 40 as the block size. Because the number of blocks per thread and second merely has only influence on the pace of the load gain, it was set to a static value of 1, such that the measurements for a specific number of threads was based on a wide range of values (considering one statistics sample per second).

On the server application, the processing graph mentioned above was used in most of the measurement passes. Because a typical processing environment can behave in an unintended way if the incoming data points do not have the expected meta data information like sensor ids etc. attached, *p4s-burner* supports to specify a special XML file that contains a scheme for randomly creating meta data that fits to the expected domain. For example, one can specify a meta data class *sensor* and a set of ids from which one is chosen for every data point randomly. By supporting this, the behaviour of a proper environment can be mimicked.

7.3.2 Benchmarking results

To fine-tune the application, a number of benchmark runs were conducted. The measurement results presented in this chapter are based on a final long-term execution lasting several days. By running the benchmark this long, it additionally provided the proof that the application remains stable even under a load that is significantly higher than the load the hardware would support. It appeared that the high load had no impact on the stability of the server application. After shutting down the benchmark, the server application simply processed all data points remaining in the queue and then went idle.

Figure 7.2 shows the average number of data points the server application actually has handled during a measurement interval (roughly one minute). The plot exhibits a very noticeable phenomen: The processing throughput nearly linearly increases up to a plateau

(approximately at 5000 data points per second). From that moment on the performance stays the same for some hours. Then it begins to increase once again linearly until it finally reaches its maximum at approximately 22500 data points per second. It is assumed that this phenomen is caused by the Java HotSpot optimization. The Java VM can generate machine-code for the running platform on-demand if a piece of code is executed many times. For example, the optimization of the CP-ABE Java code could be the cause for this phenomen, because examinations with VisualVM revealed that this code is one of the most expensive calls in the processing functionality.

During the test, *p4s-burner* also monitored the size of the processing queue (see figure 7.3). As expected, the queue stays at a low size while the server can process all incoming data points without any problems, because new data points are taken away from the queue faster than the client can provide new data points. As soon as some turning point is reached, the queue size rapidly begins to increase and then stays at the maximum for the remaining time of the test run.

Tests with other block sizes as well as other graphs showed that the former has a significant influence on the performance while the latter has not.

Because the server application itself only processes individual data points, the influence of the block size is based almost only on the connection between client and server. For every input request, an own HTTP request must be created and sent. Therefore, smaller blocks decrease the performance and bigger blocks in turn increase performance, because the overhead is mitigated. The performance evaluation showed that a block size of 40 data points leads to a good overall performance. Because of that, all entities that act as input clients should consolidate data points in a buffer and only send such blocks of data. The REST exit node of P4S already behaves this way.

On the other hand, influence of the size or complexity of the processing graph can be explained by the characteristics which a typical processing graph possesses. It is surely possible to construct graphs that do have a significant influence on the performance, but these graphs often are not very useful. In fact, most of the nodes are very simple to process and only few exist that really can have an impact on the performance. But these node types (like e.g. the knowledge enhancer) are usually not included many times in a processing graph. As long as the set of available node types stays at a very low number and the graph complexity remains at a level comparable to the graph used for evaluation in this chapter, the graph structure itself should not cause any performance problems.

In conclusion, these benchmarks have shown that the server architecture meets the requirements even at the first performance plateau. Considering that the architecture is designed to be distributed among multiple machines, performance seems to be not a problem at building measurement environments with this architecture at all.

7.4 Privacy friendliness

In the analysis chapter 4 several requirements and goals were defined that a privacy-friendly architecture must obey in order to be privacy-friendly. These points were primarily based on the eight *privacy-by-design* strategies nominated in [Hoep12]: MINIMISE, HIDE, SEPARATE, AGGREGATE, INFORM, CONTROL, ENFORCE and DEMONSTRATE. But as stated in chapter 4, in order to design a very flexible architecture some of these requirements are better fulfilled in the configuration than in the architecture itself. Hence this principle has been called *privacy-by-configuration*.

At first, besides the risks to protect against, there should be matters defined that an architecture can not preserve. While in the preceding chapters there often have been

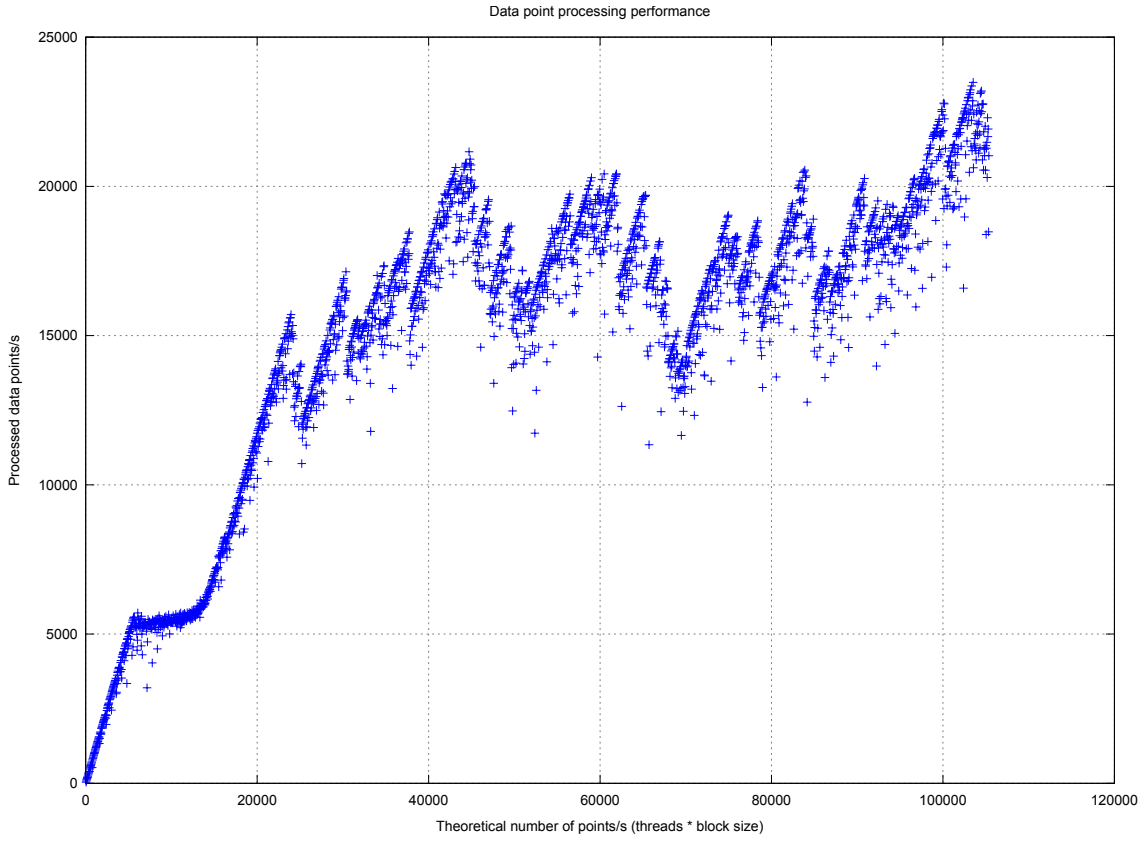

Figure 7.2: The result of the long-term performance measurement run. The x-axis shows the number of data points that the client tries to send and the y-axis shows the number of data points the server actually processes. If the server queue is full, the server blocks incoming input requests, so the difference between these two values does not mean that any data points have been dropped. Instead, the benchmark threads have to wait until there is space in the server queue once again. The number of points/s actually sent at the client roughly equals the number of points received per second.

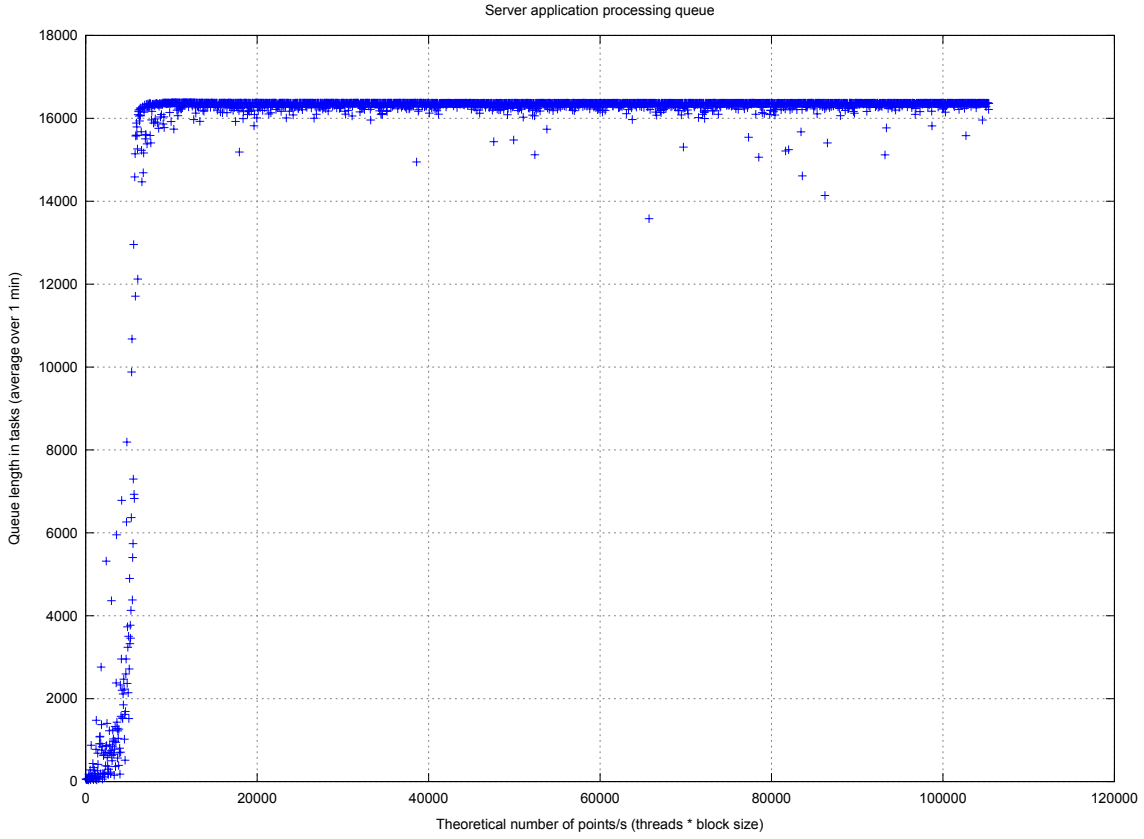

Figure 7.3: The size of the processing queue at the server application. The queue size is capped at 16384 tasks. When the queue exceeds the limit, all input interfaces are blocked until there is space available once again. Every task equals one input point to process.

company employees being called malicious (like e.g. a employer who wants to spy on his employees), the architecture can only protect the employees from being spied on by single entities. Because the company's buildings or rooms are owned and controlled by the company itself, a privacy-friendly architecture can achieve no protection if the spying is done on behalf of the company and its decision makers. Likewise, it can not shield the employees from malicious persons like an attacker having physical or remote access to the supply equipment, supply lines or controller systems, or an attacker using social engineering to acquire access to the system or to gain knowledge of any kind.

However, it is decisive that in these situations the privacy of subjects would be threatened regardless of whether a privacy-friendly architecture would be in use or not. Thus, this represents a problem that such an architecture is not able to solve. Because of that, it also is not obliged to solve it. Instead, other measures have to be taken that are addressed below.

On account of this, the remainder of this chapter is focused on how the designed and implemented architecture preserves the privacy of the subjects in an environment where it can be assumed that no one with malicious intents can tamper with the measurement and processing equipment. As a greater goal it is verified that by using the architecture the privacy of the subjects is at least not put in a position worse than the situation where no measurement architecture is used at all.

7.4.1 Processing and storage

The processing and persistence functionality is the most privacy-critical part of the architecture design. Other parts surely are able to have an impact on privacy preservation, but these considerations that have to be taken into account naturally matter on every architecture regardless whether it is designed privacy-friendly or not.

Hereafter, the design and implementation of the processing part is verified and compared to the eight strategies defined in [Hoep12]:

7.4.1.1 MINIMISE

While in the designed architecture the minimization of the amount of collected data itself is merely a matter of configuration, methods must be supported to actively and passively prevent the collection of unwanted information that could be abused otherwise.

Because the architecture itself does not have any knowledge, the creation and combination of potentially privacy-violating information depends on the knowledge that is provided. Knowledge that is not provided can not be used. Therefore, one can determine fine-granularily, which information may be stored. For example, if an employee decides to completely opt-out of the measurement process, no data will be collected for her if she does not appear in the knowledge tree. Hence, the knowledge tree acts as an upper bound for the amount of privacy-imperiling information collected. In the Git repository there is a simple Java tool called *p4s-knowledge-tester* that can test a static knowledge tree file by applying the knowledge to randomly generated data points.

A properly configured processing graph always makes sure that only data points are forwarded that are meant to be collected by filtering and dispatching data points. For example, the processing graph at the chair almost immediately drops all values that do not contain power (Watt) values. Therefore, unneeded information is not stored in the database.

Besides the processing itself, the storage scheme must assure that no information can be leaked from side-channels, like, for example, the documents stored in the database:

While the bulk of the information is contained in the encrypted blob, any unencrypted information may possibly be read by persons that actually should not have access to this information. Because of that, only the index documents contain such information, namely the timestamp and the identity, and just for querying reasons. But because there is indeed no visible coherence between index and block documents, this information can barely be used by an attacker.

For further minimization, it must be considered individually which pieces of information are necessary, e.g. for accounting issues, and which are not required. With accounting in mind it can be assumed that most companies and building operators will favour processing graphs that reduce the amount of data.

7.4.1.2 HIDE

[Hoep12] states that "any personal data, and their interrelationships, should be hidden from plain view" and explicitly mentions encryption to provide "confidentiality".

As briefly mentioned in the section above, visible relationships like the one between index and block documents could imply the leakage of privacy-imperiling information, if it is possible to extract information. For example, if an encrypted block of data would have the identities of person **A** and **B** visibly attached, an attacker could assume that both persons were at the same location (e.g. the same room) at the time the measurement was taken. Especially in a system that is designed to handle dynamic knowledge, it is of high importance that those relationships never are revealed.

With the encryption scheme in place, almost all information is hidden from plain view. However, the actual privacy and confidentiality certainly depends on the quality of the configuration. Only if the set up infrastructure of identities and privacy keys makes sure that e.g. only an employee and an energy manager may access information regarding her, her privacy is preserved properly. Because the implementation simply does not support to store sets of data points unencrypted, the risks of misconfiguration are lowered.

While the encryption itself already makes certain that unauthorized persons have no access to restricted pieces of information, it should be mentioned that the classical access control policies provided by the architecture implementation provide an additional layer of confidentiality. Additionally, because of that, a misconfiguration only can lead to information leakage if both layers are misconfigured in the same way.

Altogether, the role model has to be defined precisely by decision makers who also deeply consider legal and directive policies. It must be assured that no role has the access right, i.e. the attribute in its private key, to access restricted information.

7.4.1.3 SEPARATE

In [Hoep12] the goal of separation is meant as to distribute unrelated pieces of information in processing as well as in storing. Distributed and decentralized systems are mentioned as examples.

Here, too, the designed and implemented architecture can just provide the functionality necessary to make those goals possible. Whether a setup really achieves the goal, is once more a matter of configuration.

Because the architecture supports to connect server applications in such a way that a bigger, *global* processing graph emerges, distributed environments can be built. It is incumbent upon the administrator to assemble such a global graph that separates the data flows.

The best way to provide a reasonable separation depends on the requirements and properties and the environment, e.g. structure of the energy supply system of a building. A good point where separation can be applied exists in environments where systems are already distributed.

For example, in an office building with separate supply lines for resident companies there is no need to interconnect the measurement devices that process the real-time data. In every resident's energy supply line a separate logging device can be deployed along with a machine running the server application. Employees of these companies only connect to the server provided for their company. The building operator usually only wants to access aggregated data like the monthly overall consumption, so this aggregate value can be computed in this device and then forwarded to another central device to which only the building operator has access. That way, only aggregated values are forwarded to a central place while all other processing and storing takes place in a heavily distributed way.

Other points from the paper mentioned above can not be applied to this architecture. The splitting of database tables, for example, would lead to more information being derivable from the plain database view. When using encrypted documents that appear as blobs without any structure for an attacker, simply throwing all data in one table naturally generates a kind of background noise, so no inferences can be drawn from analyzing these blobs.

7.4.1.4 AGGREGATE

As aggregation over time and space provides the main concept to derive a coarse-grained version of real-time data in the designed architecture, this goal seems to be easily achieved.

However, whether aggregation is properly applied "with the least possible detail in which it is (still) useful" (see [Hoep12]) or not is decided by the quality of the configuration used once again. The one creating the processing graph has to make sure that data is only stored in the required granularity. Additionally, the meta data map usually has to be cleaned up once before the data points are encrypted.

P4S supports two types of aggregation: Aggregation over time and aggregation over entities (called combination) which in turn support multiple mathematical methods.

Aggregation over time allows the condensation of all data points belonging to one flow into one derived, synthetic data point that contains a mathematical aggregation of all the data points as one value. The amount of information density this value contains may vary and heavily depends on the number of data points that were consumed in order to compute the resulting value. For example, if on a specific flow a new value is measured only every hour, an aggregator node using one hour as interval obviously will have almost no effect on the resulting information density. Because of that, this number of data points also has to be considered regarding the used time interval to aggregate over.

On the other side, aggregation over entities or space can be used to remove the relationship between a flow of data points and identities from the view by summarizing real-time information for a set of entities. For example, if not the consumption of individual employees, but only the consumption of the whole floor is measured, the resulting values provide a rather coarse view on the of the single employees. Like at aggregation over time, the degree of the privacy preservation achieved by this may vary. If only few entities are combined, information may still be retrieved, especially when combined with external knowledge. If e.g. the consumption of two offices is combined and one of the employees resident in these offices is on vacation, the resulting value probably reassembles the exact consumption of the remaining employee. In that case, the more entities are used for combination, the better is the result in case of privacy preservation.

As mentioned above, it is incumbent of the one creating the processing graph to design a graph that provides sufficient protection of the subject's privacy. Considering the motto described in [Hoep12], it seems to be a good method to first investigate which granularity is required for operational purposes like accounting. Usually, most of these granularities may tend to be very coarse. As an additional step it should then be checked whether there still are flows that have a granularity that exceeds the appropriate amount.

Other problems and concepts stated in [Hoep12] are not applicable in this architecture design. For example, the problem that *k-anonymity* (see [Swee02]) tries to address is rather difficult to reproduce in an architecture like P4S. Because P4S primarily uses pairs of timestamps and values as data points that may (or not) contain arbitrarily structured meta data classes as additional information, classical matching of rows does not work that easily. Even if in a situation a relationship between tuples of information exists, at least no one should have the permission to access all required information at a single blow.

7.4.1.5 INFORM

Informing the subjects, that is the users of the system, is primarily a task to be done by the company's person in charge for the data collection matters. However, the software used for collecting and processing data can assist that person in charge in this process of informing.

The most important aspect of user information is the direct access to measured data. The subjects can view the measurements regarding them, but, of course, have to trust the environment operator that no additional data is collected.

The second aspect is the information about the type of measurements and the type of knowledge that is used for enhancing and tagging. While the knowledge acquisition is done individually and, thus, must be told to the subjects arbitrarily, the operator could, for example, make the processing graph file public, so subjects can review, how and what kind of data is collected.

7.4.1.6 CONTROL

As in the preceding section, the interaction between subject and system operator can, if anything, only be assisted by the architecture, and not assumed completely. Even in [Hoep12] no proper way of implementing such functionality is described.

Because the architecture was designed with accounting and other operating issues among other things in mind, letting the subjects have complete control over the collected and processed data is not possible without important data being not collected properly. For example, when providing the subjects with the ability to delete sets of data from the database, the energy consumption measurements can not reasonably be used for accounting. Therefore, it is feasible for the architecture to not provide any full-featured functionality to handle this.

Instead, it is once again the operator's responsibility to provide e.g. the company employees with the possibility to actively shape the collecting process. For example, employee organizations could be involved in finding a proper way how data of the subjects may be collected and processed.

7.4.1.7 ENFORCE

The architecture alone can not force an environment operator to properly respect and preserve the subjects' privacy. However, national laws and policies still exist. For example, in Germany the *Bundesdatenschutzgesetz* forbids the observing of the employee's behaviour

and enforces strict boundaries for data collection. This favours the collection of data in an aggregated, anonymized form with the least possible information density.

With the number of *Big Data* companies collecting various data, governments may decide to further restrict the collecting and processing of privacy-violating data. Therefore, a system that explicitly allows to reduce the information density is well prepared for this further development.

7.4.1.8 DEMONSTRATE

At last, the ability to "demonstrate compliance with the privacy policy and any applicable legal requirements" (see [Hoep12]) is closely related to the other three more policy-related strategies.

Once again, there is no real means to actually implement more than the equivalent of a helping hand in the architecture.

7.4.2 Encryption model

When designing the encryption model, some detours had to be done to circumvent problems with information being derivable unintendedly. For example, the initial draft included a timestamp field in the data block document, that later was removed, because it led to relationships between indices and blocks being derivable from plain database view.

These detours do have an impact on the experience witnessed by users. For example, for queries usually more than necessary data must be retrieved in order to properly display all relevant information. Thus, performance is sacrificed for a more privacy-friendly approach.

While the model theoretically allows for user deletion requests (although these were not implemented due to the thoughts described in section 7.4.1.6), the automatic cleaning of the database is rather difficult to implement, because a garbage collector would not have any information about whether a document is still in use or not. Thus, in a real-world environment the database would simply grow. To circumvent this, one could simply change the target database table, for example, once a year. Another solution would be to re-add the timestamp to the block documents, which would only cause a slight information leakage in case of unauthorized access to the database[5].

7.4.3 Security considerations

As a very privacy-critical application, the usual countermeasures against malicious users as well as external attackers must be applied. Additionally, some aspects deserve to be observed further.

As stated multiple times, the encryption model aims to provide confidentiality once the encryption scheme has been applied. However, before that very moment the data points still exist as unencrypted, privacy-critical objects in memory. Thus, an attacker that obtains access to a processing device may indeed not be able to read already stored data, but would instead be able to eavesdrop on any incoming data.

Overall, this may lower the impact of short-time attacks, that is, an attacker that gains access to a system but is detected and eliminated quickly. But instead, the impact of a silent long-time compromising is drastically increased. Because the immediate detection of an attacker would provide him with no benefit at all (unless, for example, when doing

[5]This applies only theoretically. In practice, the more data is processed and stored at a time, the less is the risk of unintended data leakage through comparing of timestamps. Although in the implementation created with this work no timestamps are saved along block documents, one could add this feature with only a few changes to the server architecture.

a denial of service attack), an attacker probably may do everything conceivable to remain undisturbed as long as possible. Therefore, attention has to be focused on intrusion detection and appropriate reaction in particular.

To counteract this risk, intrusion detection systems (IDS) can be used to monitor processing devices. When using a decentralized environment with multiple devices, a distributed IDS could be used that correlates alerts from multiple systems to create a global view on possible attacks. Further information to this topic is found in the respective literature.

Once an attack is detected, to prevent any leakage of privacy-critical data the system under attack should be shut down immediately. Because the server architecture does not store any unencrypted data permanently, the immediate shutdown ensures that the attacker's access to unencrypted data is denied. If the reaction happens rapidly, the attacker may only gain access to the real-time data of a few seconds. When the device has been shut down properly, the system can be inspected by the administrator and all attack gateways can be closed without any problem of time.

Because the architecture uses private keys with a rather long validity, there always is the risk that a private key is lost or compromised. In that situation, an attacker holding a valid private key (and of course the password with which the key is protected, as well as the login credentials) is able to access any data the owner of the key would have had access to. In particular no perfect forward secrecy is provided, because the encryption scheme can, in contrast to a transport security layer, not utilize a traditional key exchange protocol. However, because the server application contains a separate access control system, access for the compromised identity can always be denied completely, so an attacker without having plain view on the database is not able to gain access to data, once the account has been banned.

7.4.4 Crypto system choice

Because the encryption scheme does not have any requirements that common asymmetric encryption systems like RSA do not support, the architecture is not restricted to any special method.

While some of the additional features of CP-ABE were not used in the final version, it sure provides some advantages to other methods like RSA. Because CP-ABE only has one global public key for a key infrastructure and basically uses strings to determine whether a private key should be able to decrypt content, the encryption process on the server is simplified considerably. Additionally, the server can encrypt information for users that have not yet obtained a private key.

A clear disadvantage of the CP-ABE method is the acquisition process of private keys. Because the global master key has to be used to actually create the private key, there always remains the risk of a malicious key infrastructure operator who secretly keeps copies of all generated private keys. The person operating the key generation process, that is the person with access to the master key, always can generate arbitrary user keys, and, therefore, can access all data. In contrast, in an imaginary crypto context implementation using RSA, the user would generate the key pair on her own trusted computer and only provide the public key to the operator. To counteract this, there should be a clear personal barrier between the server operator and the key infrastructure operator. If this two roles are not executed by the same person, the risk of key compromising is minimized. More particular, the key operator should not have any access to the processing devices at all.

With carefully forging an environment in which the key infrastructure is properly protected, CP-ABE still provides a reasonable crypto environment for privacy-friendly data

storage. Because initial concerns about the possibly low performance of CP-ABE have been dispelled as well, CP-ABE can be a good choice.

Nevertheless, by implementing a crypto context using any traditional asymmetric crypto system like RSA, the security could possibly be improved further.

8. Interpretation

The last chapter focused on the evaluation of the architecture itself against privacy criterias. Now, it remains to verify that the questions stated at the beginning have been answered adequately. This is done by condensing the insights acquired in the preceding chapters of this work and then drawing conclusions on the subjects covered.

Which situations regarding energy monitoring do exist in which the privacy of subjects is in danger?

As described in chapter 4, the privacy of the subjects usually is in danger whenever the relationship between a set of data and the identity of a subject provides an observer with additional information. That means, with the desire to establish these links in order to improve the energy saving, conflicts with privacy sensibility of subjects arise almost inevitably.

The real-time energy monitoring in particular allows to track many behavioural facts of subjects affected by the monitoring aspect. For example, some kind of hidden presence detection could be implemented using energy monitoring. Even the energy consumption behaviour, i.e. how much energy saving is an employee doing, is an information not being in good hands with a malicious or simply unauthorized person.

In particular, the privacy of subjects is in immediate danger, whenever persons or authorities have an interest in viewing privacy-critical data, which is not associated with any energy saving matters. The prime example of such a privacy-violation, that must be prohibited at all cost, is the boss who wants to check the actual working times of her employees.

As a consequence, no simple solution exists that almost automatically allows to decide whether data collection and processing in a specific situation can be done in a privacy-friendly manner. Thus, the situation always requires extensive investigation and a set of boundaries preserving the privacy of the subjects.

However, when designing an architecture, a set of properties and requirements can be satisfied in order to provide the means to develop a strategy to make the architecture privacy-friendly.

Which properties allow privacy-friendly data monitoring?

In order to generally allow privacy-friendly processing of data gained by energy monitoring, some properties have been found that can facilitate the operation of monitoring architectures in privacy-critical situations, as stated in the analysis part in chapter 4.

Decentralization can help spreading the data processing and storage on a wide range, significantly decreasing the impact data leakage and compromising have on the privacy preservation of an architecture. Additionally, it allows to split the environment into separate domains in which the flow of the data can be directed in a way minimizing the processing of unrelated data.

Anonymization enables to remove privacy-critical information from the collected monitoring results specifically. By doing that and removing unneeded relationships between identities and data values the ability of an observer to derive additional knowledge may be greatly limited.

To further reduce the amount of data, *aggregation* can and should be used. Only by reducing the density of the information to the outright minimal amount possible (as also stated in [Hoep12]), a responsible dealing with privacy-critical is imaginable. In situations where *aggregation* is not possible and the direct processing of real-time data is required, further investigation has to be made in order not to violate the ideas of privacy-friendly energy monitoring.

While strict *access control* may provide some protection against the violation of the subject's privacy, this protection can be circumvented by persons having access to the permission policies. In particular, persons can get access to data sets after they already have been written to the storage system without the subjects being able to realize this.

In order to further protect persistently stored data, an *encryption* scheme can be used in addition to a traditional storage scheme. By encrypting data right before it is written to the database, the permissions to view data are set immediately once the data is measured. Afterwards, these viewing permissions can no longer be manipulated.

If *validation* methods are provided or at least supported, correctness as well as general approval can be enhanced. Only an architecture with its manner of functioning understandable by the subjects, can have the subjects' acceptance. This also holds true for giving the users immediate *feedback* on all data that is collected regarding them. This can be done by simply setting the access permissions accordingly, such that all information is always viewable by all persons who may be related to that kind of information. However, knowledge about other subjects may be contained in this feedback.

Additionally, attention should be directed to optimize the realization of security properties like confidentiality and authenticity when storing and transferring privacy-critical data. Only a secure architecture can reliably protect the subjects' privacy.

How can an architecture for flexible data processing and storage provide these properties?

Because the described and implemented architecture has been designed according to the properties mentioned above, it is expected that the architecture provides a good base for privacy-friendly energy monitoring.

The storage scheme developed for the architecture effectively hides any information from plain view that can be used to derive information. The configuration using GraphML files allows for a satisfying amount of flexibility when adapting the architecture to the situation.

However, configuring and testing the application in a real-world situation like the one at the chair at TU München showed (as expected and described above) that the architecture alone is not able to be privacy preserving entirely by itself. Instead, the configuration has to be done accordingly. Thus, the term *privacy-by-configuration* describes the application best. But the application, indeed, appears to provide the functionality necessary for building and operating a privacy-friendly environment for energy monitoring.

9. Conclusion

Finally, it is safe to say that the architecture designed and implemented along with this work can indeed be used for operating privacy-friendly energy monitoring environments. However, this always presupposes that the environment has been set up in a way appropriate for actually preserving the subjects' privacy.

Indeed, because the architecture requires the operator to decide explicitly whether an identity should be able to access data and in which form this data has to be, no automatic solution to the configuration problem has been found. Instead, the server application has to be configured accordingly by an authorized person. This person has to have insight to the requirements of privacy preservation and the desire on which kind of data should be collected. It remains the responsibility of this person to decide how the application is to be configured.

Because this role is predestined to easily have collisions of interests, the operator should be a person without any other obligations in the best case. Because the IDEM project is focused on providing multi-client capable solutions, one possible solution could be to let this role be staffed by the company or authority operating the building rather than a person employed at one of the companies residing in the building.

By now having an architecture for monitoring the energy consumption, the next step is the usage of the data gained by using the developed application. With intelligent energy monitoring still in the early stages, a lot of progress can be expected in the immediate future.

A. Building and deployment

A.1 Building

P4S uses Maven as a build management tool. To build all components of P4S, go to the directory **pparch/Source** in the Git repository and execute the **build.sh** script. This script will execute a Maven build (including all tests) on every component except the validation tools. After the build has completed, the generated JAR files are located in the **target** directories in the respective component folders.

A.2 Deploying

The deployment helper scripts are located in the **pparch/Deployment** directory. All these scripts require that a full build (described in the section above) has already been run.

A.2.1 Debian package

The **build_deb.sh** generates a .deb file from the built JAR files. This .deb file already has the dependencies required on a standard Ubuntu system set accordingly, but it should also work on other Debian-based operating systems. The package includes the desktop GUI, but does not depend on any X11 related packages. To actually run the graphical interface, a Desktop environment has to be installed. To run the script, the tool *dpkg-deb* has to be installed first.

A.2.2 Docker image

The **build_docker.sh** script builds a Docker image called *idem_p4s/server*. This Docker image is based on the default Ubuntu 14.04 LTS image and includes all components required to run the P4S server application.

A.2.3 Virtual Machine image

The **build_image.sh** script generates a QCOW2 root file system image file based on Ubuntu 14.04 LTS and including all components required for running the P4S server application.

While QCOW2 originally is the file format used by QEMU and, thus, many other virtualization environments do not support it, the QEMU command-line tools allow to easily convert the image file to any other proprietary format. Thus, the deployment of this image file should be possible for almost every other virtualization environment. The deployment has been successfully tested with VMware Player, VirtualBox and XEN.

A.3 Running

In order to run the P4S server application, a valid set of configuration files has to be provided using the command-line option –**config-file**. See appendix B for details.

A.3.1 Standalone application

Once installed the P4S server application can be run by issuing the **p4s** –**config-file** ***config_file*** command on the command-line.

A.3.2 UNIX daemon

The P4S UNIX daemon can be started using the default service management tool provided by the operating system. For example, on a standard Ubuntu the service could be started by issuing **service p4s start**.

A.4 Tools

A.4.1 Server (*p4s*)

```
usage: p4s [-h] -c CONFIG-FILE

The P4S server application  for  data  access,  processing and storage.
The application is started in console mode.

optional arguments:
  -h, --help             show this help message and exit
  -c CONFIG-FILE, --config-file CONFIG-FILE
                         The config file
```

A.4.2 Access client (*p4s-client-access*)

```
usage: p4s-client-access [-h] [-s PRIVATE-KEY] [-k PUBLIC-KEY]
                         [-i IDENTITY] [-u USERNAME] [-p PASSWORD]
                         [-l URL] [-o OUTPUT-FILE] from to

Access and query the flow database of a remote P4S server instance.

positional arguments:
  from                   Start of the time  interval (format yyyy-MM-dd
                         HH:mm:ss)
  to                     End of the time interval (format yyyy-MM-dd HH:
                         mm:ss)

optional arguments:
  -h, --help             show this help message and exit
  -s PRIVATE-KEY, --private-key PRIVATE-KEY
                         The  private  key   file   to   use  (default:
                         ../p4s/admin.key)
  -k PUBLIC-KEY, --public-key PUBLIC-KEY
                         The public key file  to  use (default: ../p4s/
                         pub.key)
  -i IDENTITY, --identity IDENTITY
                         The identity to use for the query (default:
```

```
                        admin)
  -u USERNAME, --username USERNAME
                        The user name for authentication (default:
                        admin)
  -p PASSWORD, --password PASSWORD
                        The password for authentication (default:
                        test)
  -l URL, --url URL     The server URL (default: http://localhost:8192)
  -o OUTPUT-FILE, --output-file OUTPUT-FILE
                        The output file (default: standard
                        output) (default: )
```

A.4.3 Input client (*p4s-client-input*)

```
usage: p4s-client-input [-h] [-n ID] [-u USERNAME] [-p PASSWORD]
                        [-l URL] [-i INPUT-FILE] [-f {list,single}]

Read data points from a file and inject them into the flow of a remote
P4S server instance.

optional arguments:
  -h, --help            show this help message and exit
  -n ID, --node-id ID   The id of the destination node (default:
                        default)
  -u USERNAME, --username USERNAME
                        The user name for authentication (default:
                        admin)
  -p PASSWORD, --password PASSWORD
                        The password for authentication (default: test)
  -l URL, --url URL     The server URL (default: http://localhost:8192)
  -i INPUT-FILE, --input-file INPUT-FILE
                        The input file (default: standard
                        input) (default: )
  -f {list,single}, --format {list,single}
                        The input format. 'list' a simple list of
                        data points, 'single' one single data point
                        (default: single)
```

A.4.4 Key Management (*p4s-key-tool*)

```
usage: p4s-key-tool [-h] {init,create,passwd} ...

Create and manage CP-ABE key files.

positional arguments:
  {init,create,passwd}  Supported sub-commands:
    init                Create a new infrastructure key
                        pair (master/public).
    create              Create a new private key.
    passwd              Change the password of a key file.

optional arguments:
```

```
  -h, --help            show this help message and exit

All required passwords are read from console or standard input
```

A.4.5 User Management (*p4s-user-tool*)

```
usage: p4s-user-tool [-h] [-t {database,xml}] [-u DBUSER] [-p DBPASSWORD]
                     [-x PREFIX] [-l URL] [-d PATH] username
                     {create,remove,passwd,show,enable,disable,
                     grant-statistics,deny-statistics,add-access,
                     remove-access,add-input,remove-input}
                     ...

Create and manage P4S user accounts.

positional arguments:
  username              The user name
  {create,remove,passwd,show,enable,disable,grant-statistics,
  deny-statistics,add-access,remove-access,add-input,remove-input}
                        Supported sub-commands:
    create              Create a new user.
    remove              Remove an existing user.
    passwd              Change the password of an existing user.
    show                Show the details of an existing user.
    enable              Enable an existing user.
    disable             Disable an existing user.
    grant-statistics    Grant access to statistics to the user.
    deny-statistics     Deny access to statistics to the user.
    add-access          Add the given identities to the access
                        list of the user
    remove-access       Remove the given identities from the access
                        list of the user
    add-input           Add the given input node ids to the access
                        list of the user
    remove-input        Remove the given input node ids from the
                        access list of the user

optional arguments:
  -h, --help            show this help message and exit
  -t {database,xml}, --type {database,xml}
                        The type of user management (default: database)
  -u DBUSER, --username DBUSER
                        The user name (for database user
                        management) (default: p4s)
  -p DBPASSWORD, --password DBPASSWORD
                        The password (for database user
                        management) (default: p4s)
  -x PREFIX, --prefix PREFIX
                        The prefix (for database user
                        management) (default: p4s_)
  -l URL, --url URL     The url (for database user management)
                        (default: http://localhost:5984)
```

```
  -d PATH, --dest-path PATH
                        The destination  path  (for  xml  user
                        management) (default: ./)

All required passwords are read from console or standard input
```

A.5 Further information

The directory **Documentation/Javadoc** in the Git repository contains the generated JavaDoc files for all components implemented in this work.

B. Configuration

B.1 Server configuration file

For any configuration node, properties can be defined by using the *property* tag. Each of this tags must consist of a *name*-*value*-pair of subtags. By *name* the property that is to be set is defined while *value* denotes the value to set this property to.

B.1.1 *server* tag

The *server* tag serves as root node of the document and, thus, contains all other nodes.

Properties of the *server* node:

- **server.port**: The HTTP port
- **server.secure_port**: The HTTPS port
- **processing.queue_size**: The maximum size of the task queue
- **processing.pool_core_size**: The default size of the thread pool
- **processing.pool_max_size**: The maximum size of the thread pool
- **statistics.enabled**: Enable the statistics module to collect statistical information at runtime

B.1.2 *accesscontrol* tag

The *accesscontrol* tag defines the method used for access control. This node may only be specified once and must contain a *class* node specifying the class implementing the chosen method.

Properties of the *accesscontrol* node:

- **path**: The path containing the XML user database (required by *XmlRoleHandler*)

B.1.3 *crypto* tag

The *crypto* tag defines the crypto method to use. This node may only be specified once and must contain a *class* node specifying the class implementing the chosen method.

Properties of the *crypto* node:

- **publickey**: The file containing the global public key (required by *CpabeCryptoContext*)

B.1.4 *knowledge* tag

The *knowledge* tag defines the knowledge provider to use. This node may only be specified once and must contain a *class* node specifying the class implementing the chosen provider.

Properties of the *knowledge* node:

- **filename**: The file containing the XML configuration (required by *XmlKnowledgeProvider*)

B.1.5 *dezem* tag

The *dezem* tag defines that a deZem mapping has to be loaded. This is required when using the deZem input interface. This node may only be specified once and must contain a *filename* node specifying the file containing the mappings.

B.1.6 *database* tag

The *database* tag defines the database to use. This node may only be specified once and must contain a *class* node specifying the class implementing the chosen database handler.

Properties of the *database* node:

- **url**: The database URL
- **prefix**: The prefix to prepend to all table names
- **username**: The user name for the database server
- **password**: The password for the database server

B.1.7 *graph* tag

The *graph* tag specifies the processing graph file to load. This node may be specified multiple times, but at least once, and must contain a *filename* node specifying the file to load.

B.1.8 Example file

```
<server>
  <accesscontrol>
    <class>
    de.tum.in.net.p4s.server.controller.DatabaseRoleHandler
    </class>
  </accesscontrol>
  <crypto>
    <class>
```

```
    de.tum.in.net.p4s.server.core.controller.CpabeCryptoContext
    </class>
    <property>
      <name>publickey</name>
      <value>pub.key</value>
    </property>
  </crypto>
  <knowledge>
    <class>
    de.tum.in.net.p4s.server.core.controller.XmlKnowledgeProvider
    </class>
    <property>
      <name>filename</name>
      <value>knowledge.xml</value>
    </property>
  </knowledge>
  <dezem>
    <filename>dezem.xml</filename>
  </dezem>
  <database>
    <class>
    de.tum.in.net.p4s.server.controller.CouchDbDatabaseContext
    </class>
    <property>
      <name>password</name>
      <value>p4s</value>
    </property>
    <property>
      <name>prefix</name>
      <value>p4s_</value>
    </property>
    <property>
      <name>url</name>
      <value>http://localhost:5984</value>
    </property>
    <property>
      <name>username</name>
      <value>p4s</value>
    </property>
  </database>
  <graph>
    <filename>graph.graphml</filename>
  </graph>
  <property>
    <name>server.port</name>
    <value>8192</value>
  </property>
  <property>
    <name>statistics.enabled</name>
    <value>true</value>
  </property>
</server>
```

B.2 Graph configuration file

To create graph configuration files preferably a graphical editor tool should be used. When creating and editing files manually, see the official GraphML reference documents in [BEHH+02] for details about the syntax.

The properties available for the individual node types are described in the Javadoc documentation for the *de.tum.in.net.p4s.server.core.processing* package.

B.3 Knowledge configuration file

Each knowledge XML file uses an (usually empty) *match* tag as root node.

B.3.1 *match* tag

The *match* tag matches regular expressions to the *class* and *tag* attributes of each meta data item. If a match occurs on an item, the child nodes of the *match* node are considered.

A *match* node may contain any combination of *match*, *add* and *remove* nodes as children.

B.3.2 *add* tag

This tag adds one or more meta data tags to the data point. The meta data class is specified by the *class* attribute while the tags are specified as plain text content of the node (one tag per line).

B.3.3 *remove* tag

This tag removes one or more meta data tags from the data point. The meta data class is specified by the *class* attribute while one or more regular expressions are specified as plain text content of the node (one tag per line). All meta data tags matching these expressions are removed.

B.3.4 Example

```
<match>
  <!-- Tag every data point with identity energymanager. -->
  <add class="identity">energymanager</add>

  <!-- Assign room numbers and additional room-related information
    to sensor ids -->
  <match class="dezem.sensor" tag=".*">
    <match class="dezem.sensor" tag="10000">
      <add class="room.id">1</add>
      <add class="room.name">Room 1</add>
      <add class="room.type">office</add>
      <add class="sensor.type">socket</add>
    </match>
    <!-- ... -->
  </match>

  <!-- Assign person names to office room ids -->
  <match class="room.id" tag=".*">
    <match class="room.id" tag=1">
      <add class="person">John Doe</add>
    </match>
```

```
    <!-- ... -->
  </match>

  <!-- Assign personal private key attributes to person names -->
  <match class="person" tag=".*">
    <match class="person" tag="John Doe">
      <add class="identity">doej</add>
    </match>
    <!-- ... -->
  </match>
</match>
```

B.4 deZem configuration file

The deZem configuration file uses a *config* node without attributes as root node. This node may contain multiple *sensor* nodes as children.

B.4.1 *sensor* tag

The *sensor* tag allows to configure all deZem sensors within an id range to use a specified set of properties. The interval is specified by the attributes *from* and *to*. The attributes *type*, *unit* and *factor* set the specific properties to be used when reading measurements from the sensors in that range.

B.4.2 Example

```
<config>
  <dezem id="i8">
    <sensor from="8184" to="8203" type="0" unit="W" factor="0.3536" />
    <sensor from="8204" to="8520" type="0" unit="W" factor="0.1020" />
    <sensor from="8184" to="8203" type="1" unit="J" factor="1.020" />
    <sensor from="8204" to="8520" type="1" unit="J" factor="0.1020" />
    <sensor from="8184" to="8520" type="9" unit="" factor="0.0001" />
  </dezem>
</config>
```

List of Figures

List of Tables

Literature

[AMKL+13] Michel Abdalla, Oleksandr Malichevskyy, Diego Kreutz, Vadim Lyubashevsky and André Santos. D6.1 - Cryptographic Requirements. In *Security for Future Networks*, 2013.

[BaGM13] Anca-Diana Barbu, Nigel Griffiths and Gareth Morton. Achieving energy efficiency through behaviour change: what does it take? EEA Technical report, No 5/2013, European Environment Agency, 2013.

[BEHH+02] Ulrik Brandes, Markus Eiglsperger, Ivan Herman, Michael Himsolt and M. Scott Marshall. GraphML Progress Report - Structural Layer Proposal, 2002.

[BeSW07] John Bethencourt, Amit Sahai and Brent Waters. Ciphertext-Policy Attribute-Based Encryption. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, SP '07, Washington, DC, USA, 2007. IEEE Computer Society.

[BoFr01] Dan Boneh and Matthew K. Franklin. Identity-Based Encryption from the Weil Pairing. In *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '01, London, UK, UK, 2001. Springer-Verlag.

[bund09] Bundesdatenschutzgesetz in der Fassung der Bekanntmachung vom 14. Januar 2003 (BGBl. I S. 66), das zuletzt durch Artikel 1 des Gesetzes vom 14. August 2009 (BGBl. I S. 2814) geändert worden ist, 2009.

[CDFS+07] J. Callas, L. Donnerhacke, H. Finney, D. Shaw and F. Thayer. RFC 4880 - OpenPGP Message Format. *IETF*, November 2007.

[deze] Intelligente lokale Datensammler. Overview of intelligent, local data loggers offered by deZem GmbH, https://www.dezem.de/de/produkte/hardware/datensammler, accessed 09.12.2014, deZem GmbH.

[ErTs12] Zekeriya Erkin and Gene Tsudik. Private Computation of Spatial and Temporal Power Consumption with Smart Meters. In *Proceedings of the 10th International Conference on Applied Cryptography and Network Security*, ACNS'12, Berlin, Heidelberg, 2012. Springer-Verlag.

[Founa] Apache Foundation. Apache Commons Daemon. http://commons.apache.org/proper/commons-daemon/.

[Founb] Apache Foundation. Apache CouchDB. http://couchdb.apache.org/.

[GaJa11] Flavio D. Garcia and Bart Jacobs. Privacy-friendly Energy-metering via Homomorphic Encryption. In *Proceedings of the 6th International Conference on Security and Trust Management*, STM'10, Berlin, Heidelberg, 2011. Springer-Verlag.

[Hoep12] Jaap-Henk Hoepman. Privacy Design Strategies. *CoRR*, 2012.

[idem] The IDEM project. http://idem-project.de/.

[Inc.a] Docker Inc. Docker - An open platform for distributed applications for developers and sysadmins. https://www.docker.com/.

[Inc.b] Pivotal Software Inc. Spring Framework. http://spring.io/.

[ISO11] ISO. Energy management systems - Requirements with guidance for use, International Organization for Standardization, 2011.

[KeDi13] Cameron F. Kerry and Charles Romine Director. FIPS PUB 186-4 Federal Information Processing Standards Publication Digital Signature Standard (DSS), 2013.

[KMPK+14] Holger Kinkelin, Marcel von Maltitz, Benedikt Peter, Cornelia Kappler, Heiko Niedermayer and Georg Carle. Privacy Preserving Energy Management. In *City Labs Workshop, SocInfo 2014*, 2014.

[LiPi08] Yehuda Lindell and Benny Pinkas. Secure Multiparty Computation for Privacy-Preserving Data Mining. *IACR Cryptology ePrint Archive* Band 2008, 2008.

[lTea] The libvirt Team. libvirt: The virtualization API. http://libvirt.org/.

[MKGV07] Ashwin Machanavajjhala, Daniel Kifer, Johannes Gehrke and Muthuramakrishnan Venkitasubramaniam. L-diversity: Privacy Beyond K-anonymity. *ACM Trans. Knowl. Discov. Data* 1(1), March 2007.

[RiCB14] Georg Riegel, Georg Carle and Helmut Brüggenhorst. IDEM Project Flyer, IDEM Project, 2014.

[RiSA78] R. L. Rivest, A. Shamir and L. Adleman. A Method for Obtaining Digital Signatures and Public-key Cryptosystems. *Commun. ACM* 21(2), February 1978.

[Swee02] Latanya Sweeney. K-anonymity: A Model for Protecting Privacy. *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.* 10(5), October 2002.

[Wang12] Junwei Wang. Java realization for ciphertext-policy attribute based encryption, Computer Science College of Shandong University, 2012.

[YaZW06] Zhiqiang Yang, Sheng Zhong and Rebecca N. Wright. Privacy-preserving Queries on Encrypted Data. In *Proceedings of the 11th European Conference on Research in Computer Security*, ESORICS'06, Berlin, Heidelberg, 2006. Springer-Verlag.