Department of Informatics
Technical University of Munich

TECHNICAL UNIVERSITY OF MUNICH

DEPARTMENT OF INFORMATICS

MASTER'S THESIS IN INFORMATICS

Privacy-aware Policy-based IoT Access Control

Ellen Maeckelburg

Technical University of Munich

Department of Informatics

Master's Thesis in Informatics

Privacy-aware Policy-based IoT Access Control

Privatsphäre im IoT durch Zugriffskontrollregeln

Author:	Ellen Maeckelburg
Supervisor:	Prof. Dr.-Ing. Georg Carle
Advisor:	Dr. Marc-Oliver Pahl
	Stefan Liebald
Date:	October 15, 2018

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

Garching, October 15, 2018

Location, Date

Signature

Abstract

At the core of the Internet of Things (IoT) is an exchange of data. Sensitive data is often not protected from unwanted access in the IoT, since removing data from the computational process can potentially limit the functionality of the system. The trade-off between protecting the user's privacy, and functionality is therefore currently decided in favor of the latter. However, user acceptance of IoT solutions is critical, since IoT systems are often deployed in personal spaces and therefore processes sensitive data. In order to give the users control over their privacy-relevant data, a fine-grained privacy control mechanism is offered to the user.

In this thesis we extend the group-based access control of the Distributed Smart Space Orchestration System (DS2OS), a framework for managing smart environments. The additional access control designed in this thesis can use any attributes of a system to express conditions in the policies, e. g. location-based conditions.

Besides the fine-grained, multi-factored access policies, the contribution of this thesis is the integration of a pre-processing unit into the privacy protection functionality. Obfuscation alters a data set so that it fits the user's privacy understanding, e. g. by summarizing or leaving out some data. This way, the coarse-grained data might still be functional to the system, while the privacy of the user is protected.

An expressive policy language is designed, using state of the art tools and frameworks, while keeping the understandability of the language, and a low system complexity in mind. The privacy policy is integrated into the DS2OS, where it is able to query oracles for evaluating policy elements, or to obfuscate the data. By using oracles to evaluate the policies, the system is highly adaptable to various usage scenarios and can transform with a growing system.

The prototype is evaluated for usability, and the performance of the system is compared to the previous state by measuring the latency that is introduced by the privacy protection functionality.

Zusammenfassung

Eine zentrale Eigenschaft des Internet of Things (IoT) ist der Austausch von Daten. Sensible Daten werden im IoT häufig nicht vor ungewolltem Zugriff geschützt, da ein verminderter Datensatz die Funktionalität des Systems einschränken kann. Aus diesem Grund wird die Funktionalität des Systems momentan als wichtiger bewertet als die Privatsphäre der Nutzer. Jedoch ist die Akzeptanz der Nutzer essentiell, denn häufig werden IoT Systeme in privaten Räumlichkeiten eingesetzt, wodurch sensible Daten verarbeitet werden. Um den Nutzern eine Kontrolle über ihre sensiblen Daten zu geben wird ein feingranularer Datenschutzmechanismus zur Verfügung gestellt werden.

In dieser Arbeit wird die gruppenbasierte Zugriffskontrolle des DS2OS, welches ein Framework zur Verwaltung von Smart Spaces ist. Diese Zugriffskontrolle kann jegliche Eigenschaft eines Systems in den Zugriffskontrollregeln nutzen um Bedingungen zu formulieren, ortsbasierte Bedingungen sind nur ein Beispiel hierfür.

Ein weiterer Beitrag neben der feingranularen Zugriffskontrolle, den diese Arbeit liefert ist die Eingliederung von vorgelagerter Datenveränderung durch die Datenschutzfunktionalität. Dadurch wird der Datensatz so verändert, dass er dem Privatheitsverständnis des Benutzers entspricht, z. B. durch eine Zusammenfassung oder ein Auslassen von Daten. Dieser weniger umfangreiche Datensatz kann immer noch ausreichend für die Verarbeitung durch das System sein, und schützt zugleich die privaten Daten des Nutzers.

Eine ausdrucksfähige Regelsprache wird unter Zuhilfenahme von neuesten Tools und Frameworks entwickelt, wobei die Verständlichkeit der Sprache, und eine geringe Komplexität des Systems beachtet werden. Die Zugriffskontrollregeln werden in das DS2OS integriert, wo es die Möglichkeit schafft andere Dienste anzusprechen, um Elemente der Regeln auszuwerten, oder um Daten zu verändern. Diese Dienste erhalten die hohe Anpassbarkeit des Systems an verschiedenste Einsatzzwecke, und können mit einem wachsenden System mithalten.

Der entwickelte Prototyp wird hinsichtlich seiner Nutzbarkeit analysiert, und die Systemleistung wird mit der des ursprünglichen System vergleichen durch eine Auswertung der zusätzlichen Latenz die eine Auswertung der Zugriffsregeln mit sich bringt.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Topic

The General Data Protection Regulation (GDPR) is a European privacy and data protection law, which became enforceable in May of this year. It aims at giving people the control over their personal data, and that data is protected by design and by default. This approach fosters an increased privacy awareness and protection. Guaranteeing privacy by default means that data processing entities have to make clear what data they process. The highest privacy protecting configuration should be the default setting of their system, so that users do not have to make an effort if they want to protect their privacy. Personal data should also only be processed if is required for a specific purpose. Privacy by design requires that those protection mechanisms are integrated into the system and considered for the whole data processing.

One area where privacy protection is still widely ignored is the Internet of Things (IoT). The Internet of Things (IoT) is an environment made up of smart devices, or *things*. Those are real-world objects and entities, which are able to perform computations and to interact with each other [16]. This creates a pervasive integration of smart devices into the environment. The environment changes dynamically, depending on the type of the devices that are in it.

Since the core of smart environments is data exchange and data processing, the main focus of current IoT systems is on high performance. A study by the Information Commissioner's Office [27] has shown that the majority of current smart devices do not inform users about the personal information they process, as it is required by the GDPR.

However, awareness for privacy in IoT is increasingly considered, not only because of the GDPR, since it was noticed that user acceptance of pervasive computing environments is low, due to the fact that users perceive a lack of control over their data [10]. Therefore security and privacy are seen as the biggest concerns in the IoT by some studies [1, 10]. Since IoT solutions are commonly deployed in personal spaces, the data processed by the IoT is also a highly sensitive one. In order to increase user acceptance of IoT solutions, privacy should be enforceable in the system, and even integrated from early development stages on, so privacy by design.

This thesis will implement a privacy protection functionality for the Distributed Smart Space Orchestration System (DS2OS), which is an open-source framework for managing smart environments [28]. The properties of a pervasive computing environment such as the DS2OS confront us with certain specific requirements. The mechanism has to be adaptable to new contexts, since the entities in a smart environment are heterogeneous and new types are introduced constantly. Actions are happening in real-time, therefore immediate responsiveness is important. A trade-off between performance and privacy has to be considered. Further problems are that smart environments have to be scalable, and resource efficient, since the devices are usually limited in their computational power.

Commonly, access control is a binary one, where access to data is either granted or denied. An approach that goes a way in-between is to obfuscate the data, which means that certain data points could be left out in the query answer, or it is summarized in a way that abstracts sensitive data to a coarser level, which is no longer considered as privacy relevant.

We will therefore define policies that let the users express their privacy understanding. Policies are chosen for declaring a privacy understanding, since they are adaptive, and allow for a flexible control over a system's behavior, which fits the requirements for IoT environments [24].

1.2 Goals and research questions

The goal of this thesis is to develop a mechanism that can protect privacy relevant data with a fine-grained access control policy. The policy has to be able to use any attributes of the environment that are available, to express conditions based on these attributes. The functionality has to be adaptable to the great diversity of pervasive computing environments. A mechanism that is new for IoT contexts is data obfuscation, which is also aimed for in this thesis.

The thesis develops a prototype that is based on the DS2OS framework, and the research that is the basis of this framework [28]. The existing access control of the DS2OS is a binary one, where access is either granted or denied, based on roles, or group membership. We plan to extend this access control to be more expressive and configurable, which allows the users to enforce their privacy understanding for more contexts, and with a finer control over the released data.

The following research questions declare the main challenges and important aspects that are addressed in this thesis. It is a guideline for the next chapters.
The individual research questions are numbered, and shortened to *RQ.X*. This allows us to later reference back to the goals more conveniently.

RQ.1 How can a privacy policy extend the existing access control, achieving a fine-grained access control?
The existing access control is based on group membership. What are solutions that can express a privacy understanding based on more environment attributes?

RQ.2 How can a high expressiveness of the policy be achieved while being understandable?
Declaring a policy language that is able to express a more fine-grained access control can easily become too complex for novices to understand. The developed policy language should be understandable without loosing the high expressiveness. Being able to alter the data that is disclosed gives the users a greater control over it, and allows them to define their privacy understanding at greater detail, therefore it is also included in the policy language.

RQ.3 How can the complexity of the DS2OS system be kept at the current level?
Understanding how a system works takes some time, and we do not want to introduce any major additional complexity. New functionalities has to behave similarly to the existing system. We want to reuse existing mechanisms of the system, as this helps us in maintaining the system's complexity and understandability.

RQ.4 How can the performance of the DS2OS system be maintained or appropriately slowed down?
Introducing privacy policies reduces the processing speed of the system, since it has to take more factors into consideration. We declare the goal that the response time of requests should stay below 1 second, ideally it should stay below 0.1 seconds to achieve seemingly instantaneous responses (see [26]).

Another factor that is considered is to maintain the autonomy of the DS2OS by keeping it distributed and self-managing.

1.3 Outline

In chapter 2, privacy challenges in general, and in the IoT domain are analyzed, and possible solutions which employ a policy language are inspected and assessed. The DS2OS is analyzed in regards to the features that are relevant for the access control.
Chapter 3 looks at related work that is closest to the privacy protection implementation proposed by this thesis, and compares and evaluates it for applicability.
A prototype is designed and implemented, which overcomes the challenges identified in the first chapters. The design is presented in chapter 4. Specific implementation details of the prototype are explained in chapter 5.
The developed prototype is tested against the requirements identified in the first chapters in chapter 6.
Chapter 7 summarizes the achievements of this thesis, and possible enhancements to the proposed solution are listed in chapter 8.

Chapter 2

Analysis

This thesis aims at defining a privacy policy language for an IoT environment that is able to express the user's understanding of privacy conform data sharing. As chapter 1 states, the policy language has to be able to include diverse environmental attributes as conditions for the access control (see also **RQ.1**). The data that is released upon an access grant also has to be configurable, as **RQ.2** states. To be able to design such a policy, we first have to understand what privacy is, what the parameters of IoT environments are, and what policy solutions exist for those challenges.

Privacy is a term that means different things to different people [25], and different application domains make the privacy definition even more difficult. Smart spaces are highly dynamic, and composed of diverse types of devices. The privacy requirements that users can have in general, and the challenge of introducing privacy in a pervasive computing environment are explored in section 2.1.
First, general privacy aspects are explored in subsection 2.1.1.
Privacy aspects that are specific to smart spaces are discussed in subsection 2.1.2.

Policy languages are discussed in section 2.2.
Policies chosen as a means to declare the user's privacy understanding, since they are an adaptive and flexible means to control a system's behavior. In order to control the system's components, a semantic description of is has to be present and capable of being integrated into the policy language. Therefore we extract relevant contexts for describing environments, and how policy languages in general use them in subsection 2.2.1.
In subsection 2.2.2 we look at existing policy languages for smart environments to find current challenges and solutions, and we gain an understanding for how a fine-grained language is designed.

A new mechanism in access control is data pre-processing. We want to integrate this into the policy language we design and therefore explore existing solutions in subsection 2.2.3.

The goal of this thesis is to design a fine-grained access control for the **DS2OS**. The aspects that are relevant for integrating this access control into the system are explored in section 2.3.

Lastly, section 2.4 summarizes the most useful findings from the literature.

2.1 Privacy

2.1.1 General aspects

Warren et al. define privacy as the „private life, habits, acts, and relations of an individual“ [38]. This broad definition shows that privacy starts where more insight into a person's life can be gained. This insight into a person's life can take many forms, as some for example willingly share photos of themselves with the world, while others regard them as confidential and would only share their photos with a specific subset of people, or even none at all.

Westin introduce the term *information privacy*, where a person decides „*when, how and to what extent* information about them is communicated to others“ [40]. So with this definition, the notion of privacy is moved from a person-centric view to an environment perspective.
Defining what privacy is to a person is difficult, as Shankar notices when asking people about their *personal data*. Shankar finds that the term is too abstract and therefore doesn't speak to the person, and only example scenarios which helped them see *which information might be interesting for others to gain and which they want to protect* [34].
Tang et al. notice that the number of recipients influences the privacy decisions a user makes [37].
Kwasny et al. focus on further aspects of information sharing: The *nature of the information, the access rights others have, how the data is stored*, and why it is requested [25].

This shows us that privacy can have multiple aspects when it comes to personal data, and that users should be in control of the access to their personal information. Therefore the most relevant privacy aspect which we keep in mind for our solutions is *user control over the conditions under which data is shared*, which can be time, the manner of sharing, the extent, and other conditions.

<PR.1> *Sharing conditions*

Notation information: In order to summarize the most relevant findings, and to later reference back to them, we numerate the most relevant privacy requirements and list them as <PR.X>.

Another aspect is that the users are even not aware of how information about them can be interesting for others. Therefore the users has to be assisted in gaining an understanding and awareness of how their privacy can be protected.

<PR.2> *User awareness*

2.1.2 Privacy in smart environments

Privacy has no universal definition, as the previous section shows, and therefore is individual to the use case and the user's needs. This means that we have to give the control back to the users, as only they know what their privacy definition is. The problem that arises when giving the control of the privacy definition and privacy realization back to the user, is that the users may not be able to convey their understanding of privacy in total clarity to another person, as Shankar notes [34]. We therefore have to determine aspects that might be privacy relevant for users, and thereby also raise awareness of private data. Since we want to design a finer privacy control for a smart environment, we explore the additional privacy challenges we face in pervasive computing in this section.

As Giusto et al. defines it, pervasive computing environments are spaces with „a variety of 'things' or 'objects' [...], which, through unique addressing schemes, are able to interact with each other and cooperate with their neighboring 'smart' components to reach common goals“ [16]. This variety of objects that interact with each other becomes a challenge, as no one-fits-all privacy consideration can be taken for the smart environment.

<PR.3> *Adaptability to high variety of entities*

Christin et al. extend Westin's definition of privacy to make it applicable to smart environments. They state that "privacy in participatory sensing is the guarantee that participants maintain control over the release of their sensitive information". The sensitive information they refer to are the sensor readings, and the information that can be inferred from the user's *interaction with the system* [8]. Therefore our privacy definition becomes more specific, it not only concerns the personal data, like acts and habits of a person, but also the environmental data like relations, as we state in the previous section. The meta data of a person's interactions with a system, as for example the frequency and timespan in which the person uses the system, are part of this environmental view.

Atzori et al. formulate it more explicitly by saying that privacy is not only for protecting against unauthorized access but also making spying on users harder, it protects from *surveillance* [1]. Shankar state that the surveillance potential shifts the power dynamics. In the use case they present, they see that it is bound to marginalize the elderly users, as they become a subject of surveillance in their own homes [34].
Protecting users from surveillance, and protecting the metadata that can be gathered in the system is currently not our focus, since the DS2OS is a framework that is deployed by the user, and therefore no malicious third party is involved which can exploit the users. However this aspect can be kept in mind for a further extension of the privacy protection functionality, since users in the system could possibly try to monitor other users.

Similar to Atzori et al., Panagiotopoulos et al. express the need for extended privacy protection, not only as unauthorized access, but also that the *amount of personal data should be limited and saved from unwanted processing* [29]. Atzori et al. further note that privacy in the IoT is facing a new challenge, since data collection, mining, and provisioning are done in a fundamentally different manner, and that this allows for data collection at a bigger scale.
This bigger scale of the data collection also relates to our goal of employing preprocessing of data, since control over the personal data not only has to be about access grants or denials, but also about altering the data set to be less sensitive. A big data set can for example become less privacy sensitive when it is summarized per day.

<PR.4> *Data richness*

This new form of data collection becomes even more threatening to privacy when considering that in the IoT even people who are not using the IoT services may be analyzed by the system [1].
On a general note, protecting passive participants is a challenge that is too broad for our design, as the way passive participants are affected by the system cannot be predicted, and it is individual to the use-case. However, a general solution for protecting passive participants of an IoT system might be the proposal by Panagiotopoulos et al. who suggest to implement basic privacy principles in all systems, despite their high diversity. Those basic privacy principles are derived from international organizations and span the following topics: Purpose specification, anonymity, security, individual participation, accountability.

- Purpose
  The *purpose* has to be specified, explicit, and legitimate.
  This can be a useful part of our policies, as users might want to share data only for

specific purposes. However it is close to the previous requirement for conditions (<PR.1>), therefore this requirements is extended to cover the purpose of the access request as well.

- Anonymity
  Anonymity is described in this context as being able to identify the "data subjects for no longer than it is required for the purpose for which those data are stored" [29].
  This is not part of the focus of our design, as we primarily want to protect the data from unwanted access. However, anonymity and data lifetime are important aspects for privacy, and should be addressed in a further enhanced privacy mechanism.

- Security
  Security means preventing data destruction, modification, and unintended or unauthorized disclosure.
  This is already implemented in the existing access control of the DS2OS and therefore is not a focus of our solution.

- Individual Participation
  For *individual participation*, participants are informed about the data stored about them.
  This is already addressed in our requirement for user awareness of sensitive data, see <PR.2>.

- Accountability
  The entities in the system that are processing user information have to comply with the previously mentioned principles and have to be *accountable* for it.
  The DS2OS employs certificates to identify entities, and to guarantee accountability. Therefore it does not have to be further addressed in this thesis.

Even if those basic privacy principles are implemented as a minimal guarantee for privacy, how can we be sure that the users understanding of privacy is represented well in the system configuration?
In the previous section, we mention Shankar's observations that it might be difficult for people to implement their conception of privacy because they don't *know which information is processed*, that might be sensitive to them [34] (<PR.2> - awareness). This becomes even more important in the IoT, as the main feature of it, the pervasiveness, makes the users unaware of the computations performed and the traces left behind, as Panagiotopoulos et al. note. They conclude that this limits the user's control over privacy [29].

The users might also not be able to foresee the way their configurations and actions impact their privacy [11]. Therefore we have to make the *configuration of privacy easy to understand and verifiable*, even for people having little privacy and technology expertise, as Henze et al. mention [17]. They foresee a challenge in making the privacy control *simple* for novices, but at the same time allowing for a *fine-grained control* for experts [17].
These aspects correspond to our research questions **RQ.1** and **RQ.2** that aim at a fine-grained, understandable, and expressive privacy protection, and are therefore important requirements for our solution. The policy has to enable the users to express their privacy to the level of detail they require. If the policy mechanism is not understandable, users are not able to protect their data, therefore understandability is highly important.

<PR.5> *Understandable privacy configuration*

Christin et al. highlight four main privacy challenges.
The first challenge is to include the users in the privacy decisions. A necessity in order to achieve this is to make the privacy configuration *easy to use*, as the user won't make use of the adaptability options, or won't understand their implications, if it is too complicated.
We already summarize those aspects in the requirement for a simple and understandable privacy configuration (<PR.5>).

The second challenge they see are *composable* privacy solutions. In order to do this, Christin et al. propose to implement privacy implemented at the system level.
The composable privacy solution we aim at is an extended access control that can include diverse contexts. It is implemented at system level, as Christin et al. suggest. Therefore a composable privacy protection is also part of our **RQ.1**, and is covered already be the requirement for adaptability (<PR.3>).

The third challenge is the *trade-offs* one is facing between *privacy, performance, and data fidelity*. One example for this is that emergency situations may turn privacy concerns into less prioritized issues.
As we state in **RQ.4**, we want to maintain a good performance of the DS2OS. A fine-grained protection of privacy can potentially lengthen the response time to requests, since more factors have to be considered and evaluated for it. Therefore finding a trade-off between a maximum of privacy protection, and a good performance of the system is an essential requirement for our solution. We also declared the goal to obfuscate data in order to increase the privacy protection. Therefore data fidelity can be decreased for more privacy, but the functionality of the system has to be kept in mind.

<PR.6> *Balance privacy, performance, and data fidelity*

The last challenge Christin et al. see is how to make privacy *measurable*. This issue is further divided into generalized privacy metrics, where the nature of the input and output parameters is considered, and provable guarantees for privacy, which is often difficult when the system is a black box and allows for no privacy guarantees or proofs [8]. This challenge is too broad to consider it in this thesis, since first of all a privacy protection has to be defined before it can be further analyzed.

From this section we can derive that our solution must support diverse factors for a good privacy protection. The user has to be in control of the conditions under which data is shared, and for what purpose it is requested (<PR.1>). Since the IoT is a diverse system, the privacy protection must be able to include a high variety of entities (<PR.3>). Those requirements can be linked to the research question **RQ.1**.
Requirements that are related to the research question **RQ.2** are <PR.5> (understandable configuration), <PR.4> (coping with rich data sets), and <PR.2> (user awareness of sensitive information). An expressive policy language has to be able to handle large data sets and the sensitive data in them. The user has to be supported in the policy declaration process in order to make the user aware of sensitive data, and to make the privacy configuration understandable.
The requirement for a balance between privacy, performance and data fidelity (<PR.6>) is linked to research question **RQ.4** and is considered throughout this thesis.

2.2 Policy languages

This section looks at privacy policy languages in general IT systems, and in pervasive environments. As Kagal et al. note, policy languages are used to express a desired behavior of a system and the entities in it. Since they provide a high flexibility they are used for controlling access rights for users and services, thereby ensuring the security of the system. An automated evaluation of the policy becomes even more useful when considering the diversity of pervasive environments, since ensuring the security or privacy in such rapidly changing environments can hardly be done manually [24].

If we want to control the behavior of the system and the entities in it with the help of a policy language, the first step is to look at the entities in an environment, and how they and the environment can be described. We do this both for general IT systems, and for IoT environments.
In subsection 2.2.1, we look at relevant aspects of policy languages in IT systems in general. In subsection 2.2.2, the focus is on smart environments and what aspects and challenges are special to them.

2.2.1 Policy languages in general

Describing an environment and the entities in it is approached via three aspects in this section.
The first focusses on relevant contexts that can be described in policies generally (subsubsection 2.2.1.1).
The second part explores the vocabulary, or in other words, ontologies that are commonly used in policy languages that describe environments (subsubsection 2.2.1.2).
When the environment is described the policy can make logical evaluations since it then has an understanding of the environment. How those decisions can be made and how the policy itself can be defined is discussed in subsubsection 2.2.1.3.

2.2.1.1 Contexts

Dey et al. define context as „any information that can be used to characterize the situation of entities [...] relevant to the interaction between a user and an application“ [12].
So in basic words it is a description of the entities in an environment and the interactions taking place between them.
A policy language must therefore be able to describe the information in an environment, as well as the entities and their interactions.

<C.1> *Information*

<C.2> *Entity*

<C.3> *Interactions*

Notation information: In order to summarize the most relevant contexts, and to later reference back to them, we numerate the most relevant findings and list them as <C.X>.

Other basic privacy contexts can be found in the literature when we look at the work of Ge et al. They list *data, policy, entity, and operation* as the main components that make up the privacy domain. *Data* is what is to be protected, the *policy* protects it, *entities* are the service provider and service user, and the *operation* is the action an entity can take on data [15].
This describes our previously found context for entities <C.2> more detailed as service providers and service users. The interactions (<C.3>) can also stand for operations that are performed on data. The policies can be also seen as context that should be expressible in policies, as they can be conditions for interactions. We therefore add it to our list of useful contexts.

<C.4> *Conditions*

Ye et al. emphasize that providing the correct context to the system it is a multi-factored challenge, where the system behavior, the recipient, the time, and place all have to be correctly modeled in order to be able to further work on that context [41].
Therefore additional contexts that are important for describing an environment are time and locations. The previously mentioned context for conditions can be able to express those environmental conditions as well, but since they are present in any system, they are listed as individual contexts.

<C.5> *Time*

<C.6> *Location*

The five contexts which we found in this section are the most basic descriptions of any environment. They allow us to describe the information contained in a system, the entities and their interactions, as well as the conditions under which an interaction should take place.

2.2.1.2 Ontologies

The contexts listed in the previous section are general terms. Poveda Villalon et al. explain that a formal semantic for the context is created when context is represented through ontology-based models. This formal semantic can be used to share and/or integrate context information [31].
Since we want to reuse existing context descriptions in our policy design, we look at several reference ontologies in this section. We do not require a full description of an environment, since the DS2OS already provides such a semantic representation. This is explained in more detail in section 2.3. However, existing ontologies indicate to us what contexts are important and universal to many systems.

Resource Description Framework (RDF) is a data model that can represent entities and relations between them as a graph. The graph consists of subjects, predicates, and objects.
The subject can be an *entity* (<C.2>), a feature, and instance, or in general a start node.
The predicate links the subject and the object, it expresses a *relationship*, an *attribute*, and many other properties. This is a more detailed description of the context for interactions (<C.3>).

<C.7> *Relationship*

<C.8> *Attribute*

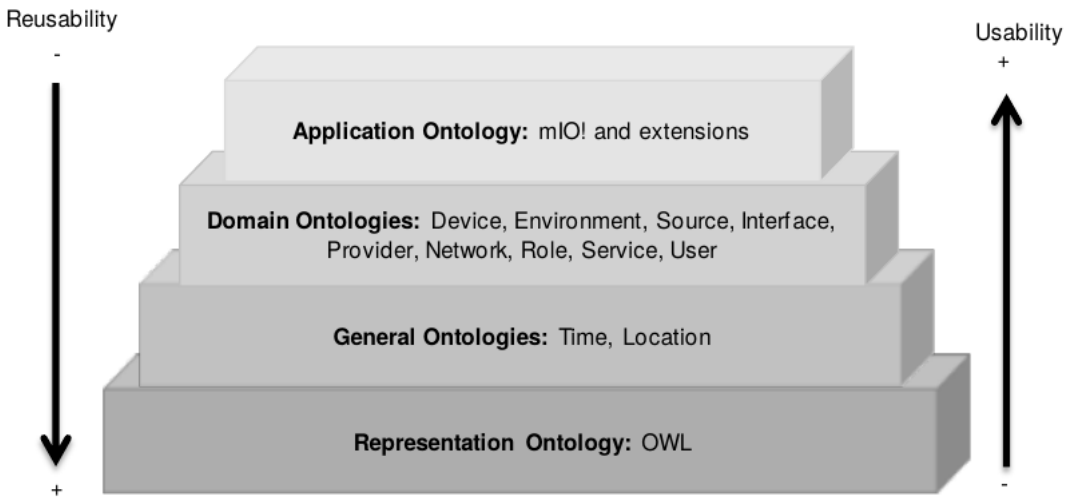

Figure 2.1: Ontology reusability and usability (from Poveda Villalon et al. [31])

The object can be a *value* (<C.1>) or an end node [9].
A useful approach of RDF is to name the link between entities as predicates, since this corresponds to the sentence structure humans use. This might help us in defining an understandable policy.

The *Web Ontology Language (OWL)* is a semantic language that uses RDF/XML as syntax which allows for sharing of the vocabulary. It aims at being a basis for creating ontologies [6]. With the help of OWL, other ontologies can map their domain-specific ontology onto a common-sense ontology [39].

Poveda Villalon et al. show how different ontologies are conceptualized for different levels of abstraction in Figure 2.1. They use it to classify the ontologies used in the mIO! ontology network they developed. It can be seen that OWL is an example for a very basic representation ontology. Through its basic level of expressiveness, it is not usable for a specific solution straight away, but can be reused by other ontologies easily. *Time* and *location* are identified by Poveda Villalon et al. as the ontologies in the mIO network that are universal and can be used independent of a specific domain. This enforces the decision to have individual contexts for time (<C.5>) and location (<C.6>).
The domain ontologies describe aspects of a domain, that are specific to it, but at a level that it can be reused by applications that are more specific in that domain. Examples are *device, environment, source, interface, provider*, *network, role, service* and *user*.
We can summarize the devices and other entities in the existing context for entities (<C.2>), whereas the role, and the network are attributes of the entities (<C.8>).

At the most individual level, the application specific ontologies are created to fit to a specific use case [31].

The problem in pervasive computing is that it is not sufficient to describe that a device is of a certain type, since even similar devices may support different standards and protocol, as Dixon et al. notes. Even activities are different for every setup [13].

An ontology that shows well how a domain can be described with is the Friend of a Friend (FOAF) ontology. It has multiple classes to describe different actors and higher level organization forms, such as *person, project, organization,* or *image*. Those classes can have different properties, such as *age, lastName,* and *title*, and even relationships can be described with properties like *has, made,* and *knows* [4].
Another more domain-specific ontology is Standard Ontology for Ubiquitous and Pervasive Applications (SOUPA). It uses OWL as a basis for specifying an ontology made for pervasive environments [6]. It is designed in a modular way and parts of its classes can be mapped to other ontologies like the FOAF ontology, therefore reusing existing ontologies and providing a higher level of interoperability. We have a closer look at it in subsection 2.2.2.

2.2.1.3 Policy

Controlling the access to information and thereby ensuring security and privacy can be done based on various contexts, which the previous sections explore. At the most basic level, it can be done based on the identity of the requesting entity and the requested object (<C.2>).
Solutions like MAC, DAC, RBAC, ABAC, or XACML are introduced which make access decisions based on those contexts and other parameters, such as attributes, categories, rules, etc.

- **MAC**
  In MAC, a fixed set of rules decides whether access is granted, based on security *labels* (<C.8> - attribute) [32]. The rules cannot be altered by the user and are checked for every action requested by every entity (<C.2>). Since this access control mechanism is quite static, it is good for a central policy that should be forced on all entities.

- **DAC**
  Discretionary Access Control (DAC) allows change to the policy by allowing the users to grant their access rights to another user. This forwarding of access rights can be restricted by using MAC as well. A known implementation of this is the

Unix access control for files, where the *owner* of a file can grant read, write, and execute permissions to the file to other entities in the system.

- **RBAC**
  Role-based Access Control (RBAC) is, as the name suggests, based on roles. Permissions are assigned to roles, and roles in turn can be given to users or entities (<C.2>). When an entity in the system got a role assigned and is authorized for this role, it can act on the *permissions* that come with the role. Permissions are just another term for conditions (<C.4>).
  RBAC is commonly used in companies to manage access, since RBAC roles can be grouped to express a hierarchy. Through this hierarchy, permissions from roles that are lower in the hierarchy are grouped into a higher level role, making role management easier [5].

- **ABAC**
  Attribute-based Access Control (ABAC) was designed because limitations of MAC, DAC, and RBAC were emerging [23]. The complex constraints that can't be expressed in these policies are manageable in ABAC. ABAC can also be used to enforce MAC and DAC.
  The access control is done via policies that use attributes to evaluate a rule set. The attributes can be of various types, such as user, resource, subject, object, environment, and others [23].
  This is covered by our contexts for entities (<C.2>), and attributes (<C.8>).

  Sandhu also adds *action* and *context* to the attribute types which are included in the authorization decision. The contexts for interactions (<C.3>) covers actions, and context is represented through the more general term *information* (<C.1>). However it is interesting to see how those can all be described as attributes, which is also one of our contexts identified so far (<C.8>). He explains how attributes are *name-value* pairs, and that the *values* can be complex data structures.
  The possible attributes can be associated with the above mentioned attribute types. This makes ABAC extensible, which is a primary requirement of this thesis (<PR.3> - adaptability).

  For access control, the attributes are converted into rights through the policies. [32] The attribute based policies allow for a greater flexibility of access control, as the attributes can be adapted to the intended domain, but it also comes with greater complexity, since it employs a whole architecture for the policy evaluation, as Jin et al. explain. Additionally, they note that the higher flexibility of ABAC causes the comprehensiveness and policy declaration to be more difficult [23].

We have a closer look at ABAC in chapter 3, since it features quite a few aspects that are useful for the goal of this thesis.

- **XACML**
  eXtensible Access Control Markup Language (XACML) is also attribute-based and defines a policy language, and a request/response language for deciding the access control. The policy language describes the requirements that have to be fulfilled to grant access. It can be extended to include new data types. The request/response language describes the request to access a resource, and an action that is to be performed. This request is then interpreted, evaluated, and a result is returned. The response can either be *permit, deny, indeterminate*, or *not applicable* [36]. We have a closer look at XACML in chapter 3 as well.

Both ABAC and XACML are designed to be used with a Policy Decision Point (PDP) and a Policy Enforcement Point (PEP). A Policy Enforcement Point (PEP) is responsible for forming the request caused by the action that is to be performed. For this it takes the attributes of the requester, the requested resource, and possibly other information. The request is then sent to the PDP.
The Policy Decision Point (PDP) knows the policies of the system and looks at the ones that are applicable to the request. It evaluates the request based on the policy and thereafter knows whether access is allowed. This result is then sent back to the PEP. The PEP grants or denies access to the requester based on the result [36].
Since this approach is used by the two fine-grained policies ABAC and XACML, its concept is considered for our design.

Chen et al. strike a balance between those approaches by using an ontology to do policy evaluation. First a policy is defined using SOUPA, and then sent to a PEP. The PEP transform all actions that reach it into a SOUPA action representation. The resulting representation and the corresponding ontology are loaded into a PDP, which in their case is a description logic reasoner. The result of the reasoning is fed back to the PEP, which then looks if the classification of the action is of type *pol:PermittedAction* which allows access to be granted [6].

It can be concluded that context descriptions and policies both have the characteristic of varying from being broadly applicable but not very expressive, to being highly expressive but only for a specific domain and being of higher complexity. We consider the presented policies in our design of our solution. The context that seems to be the most useful one when integrating it in a policy is an attribute (<C.8>), since an attribute can express all relevant attributes of an environment.

2.2.2 Policy languages in smart environments

This section explores additional contexts and ontologies that are relevant for smart environments (subsubsection 2.2.2.1). It further looks at existing policy languages of this domain in subsubsection 2.2.2.2.

The previously explored aspects of privacy are mainly focused on how the system should behave, on how the user can be integrated, and to some extent, on the sensitive data that has to be protected. The identified ontologies are aimed at describing general application domains.
In pervasive computing, describing the domain with its actors, the environment and their interactions with an ontology is especially useful since the interaction between the participating agents has to be done in a machine-readable format.

For privacy rules, we have to rely on a description of the environment in order to be able to evaluate the policies. The user cannot be asked to review the correct functionality of the privacy protection since the user would even not be involved in many interactions in the system. The main feature of pervasive computing is that the devices communicate with each other without a need for human supervision. Therefore the knowledge that lies in the ontology is used for automated policy evaluation. This section explores the contexts that are of high relevance to pervasive environments, and existing policy solutions.

2.2.2.1 Contexts and ontologies

As access control is done to restrict access of other entities, one of the main context in pervasive computing is the *user or service* that is requesting access to the data (<C.2> - entity). For this context, multiple properties of the user or service may be useful, as for example the group a user belongs to, the organization, the employment status, the mobility pattern, the skills [31], or the account validity [13]. This list of attributes is different for every environment. A commonly used ontology for linking people and information is the FOAF [4] ontology we mentioned in subsection 2.2.1.

As we explained in subsection 2.2.1, Poveda Villalon et al. declare *time* (<C.5>) and *location* (<C.6>) as general contexts, since they can be used in any knowledge domain. Time is also identified by Iacob et al. as a key feature [22]. Therefore we explore those two contexts more elaborately than other contexts, showing how relevant contexts for a policy language can be identified on this working example.

Time (<C.5>) can be described in various forms. It can be expressed as *temporal relations* and *events* [6], *temporal units and entities*, *instants*, and *intervals* [31], for

example the time since collection, the time zone the data is collected in, or the schedule of an agent. An example for a time interval is presented by Hong et al., where access is granted for example from 9 to 11 a. m. on Fridays [20]. Using a time windows and time comparison is also done by Dixon et al. [13].
We can use those fine-grained descriptions of the time context to consider it in a time-based policy.

A reference ontology for time is the DAML-time ontology, which describes temporal concepts, and properties of time [18]. It is defined with XML and RDF. DARPA Agent Markup Language (DAML) describes different *temporal relations*, ranging from *temporal entities* like *instants*, *intervals*, relations like *before* that work on the temporal entities, to *temporal units*.
Temporal units are *second, minute, hour, day, week, month, year*.
Intervals can have multiple relations, which is addressed in the DAML ontology through evaluating if two intervals are either equal to each other, one starts before the other, whether they meet or overlap, start or end at the same time, or if one is during the other interval.
DAML is a time ontology, but can link to an event ontology and thereby express the relation between time and events. This is done with the help of the following predicates: at-time, during, holds, and time-span.
The ontology also includes an ontology for a clock and a calendar, but this is beyond the interest of this thesis.

The take-away from this time-based ontology is a reference design for including time-based conditions into our policy. Time is not part of the ontology of the DS2OS, therefore a logic for evaluating time conditions has to be implemented for this thesis. Some more detailed time concepts are discussed in chapter 3.

Location (<C.6>) is considered by Poveda Villalon et al. as a general ontology for describing the majority of domains, as we mentioned before. Location context is also present in many setups found in the literature, especially [8, 20, 7, 13, 22, 19]. Hong et al. identify location privacy as one of three major themes in research (beneath smartphone privacy, and pervasive sensing applications) [19]. For Iacob et al., location is one of the key features [22].
Therefore, it is explored at greater detail, even though location is a context that has to be expressed in the ontology of the DS2OS. This thesis does not implement a semantic description of location information, but it aims at finding expressive means to declare privacy conditions for various contexts. Location serves as an example for these contexts.

Hong et al. use location information as part of their description of their *subjects* and *objects* (<C.2>) [20]. Christin et al. provide an example, where the *presence of trusted people* like family or friends is taken as a decision criteria. Another example by them is whether a location is considered as *sensitive*, which has to be defined as such a priori [8]. Schilit et al. propose using different granularity levels for location context, reaching from basic *coordinates* over *rooms and floors of buildings* to *streets, neighborhoods, cities, and countries* [33].
For our implementation, we consider it useful to allow policies to express that the presence of other entities is required. The different levels of granularity, at which location information can be expressed are considered.

Standard Ontology for Ubiquitous and Pervasive Applications (SOUPA) is an ontology that contains a core ontology and an extension ontology. It is build on top of the OWL ontology. The SOUPA Core can be used to describe contexts that are common in pervasive computing environments. It can model *intelligent agents*, *persons*, *space*, *time*, *events*, *actions* and *policies*.
Those are contexts which we already declared as important ones. We can base our policy design on those core contexts to our policy, since they are a fundamental description of an environment.

The SOUPA Extension ontology extends the Core ontology, and is aimed at defining more specific application contexts. It also acts as an example for other ontologies that might want to extend the Core ontology [6]. Figure 2.2 shows the contexts of both ontologies and how they are linked through OWL. SOUPA is employed by Poveda Villalon et al. to formalize knowledge about buildings, coordinates, distances, countries, etc. [31].
We can relate to this design, since it provides high adaptability for new contexts in diverse environments. The design in this thesis is not made for a specific use-case, therefore it has to be adaptable to any setup. By keeping the policy open for new contexts, as it is done by SOUPA, we can ensure this requirement (<PR.3>).

2.2.2.2 Policy

Different policy languages exist for pervasive computing environments, and we explore some of them in greater detail in chapter 3, since they give us a good understanding of how a policy language has to be designed for smart environments. The language addressed in the next chapter are Rei [24] and Bark [20], together with other policies identified in subsection 2.2.1.

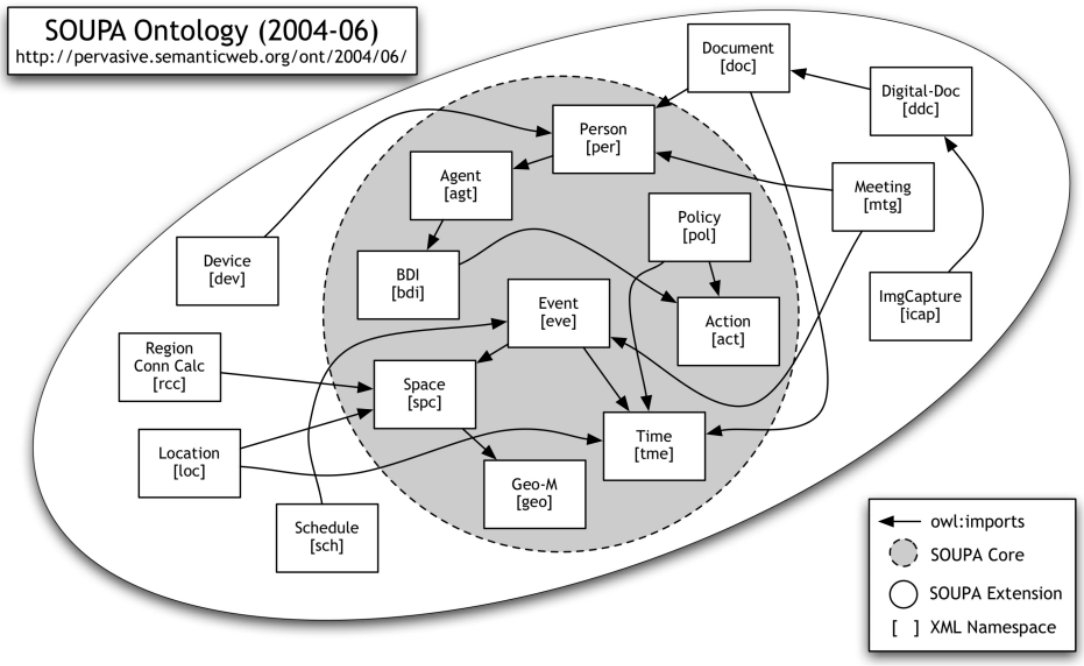

Figure 2.2: The SOUPA ontology with the Core and Extension ontology (from Chen et al. [6])

A general aspect in regard to introducing policies is mentioned by Henze et al. They declare default policies in their setup, which allow the user to adapt the default to their individual privacy requirements. This way, privacy experts can define appropriate and extensive privacy policies, making use of the full capabilities of the systems privacy configuration, and can convey their full privacy understanding in a policy [17]. Privacy novices can use and adapt these default policies.

The HomeOS platform described by Dixon et al. provides a user-friendly representation and implementation of a pervasive environment in a home setup. Privacy is protected in the HomeOS by granting access to sensor data only to applications that are allowed to access the corresponding devices.
Similar to RBAC, HomeOS has hierarchical groups for users and devices. This tree hierarchy is chosen since it is simpler to understand and retrace by the user. It avoids the case that a user, who is not in an allowed group, can gain access by being in another group. The devices are also modeled in a hierarchical manner, with the difference that the root is a spatial point. This best fits the way humans conceptualize devices in their home. The device groups also allow for a group-specific policy on top of device-specific policies.
A group that is unbounded from location groups is the high-security group. Devices that are assigned to this group are not accidentally accessible, but rather have to be explicitly

allowed for certain applications. Dixon et al. use this group to also automatically assign certain types of devices (e. g. door locks) to the high-security group, thereby making them even more fail-proof.
We consider implementing default rules into our prototype, since they highly improve the usability and understandability of policies.

The access policy of the HomeOS is expressed with Datalog rules. They are of the following format, where `r` is the resource, accessible by group `g`, using module `m`, in the time window `T` at day of the week `d`, with priority `pri`, and an access mode `a`:

`(r, g, m,` T_s`,` T_e`, d, pri, a)`

The access mode can either be *allow* or *ask*. By making the access mode either an automatic permit or an interactive request, the user doesn't have to define all allowed accesses, but rather can leave some for later.
The way the policy is defined provides a high functionality, since Datalog rules can be easily evaluated. All contexts we listed is covered in this policy, except for location context, although this might be expressible through the module. However, we do not want to explicitly define access policies that require a user involved, since this slows the performance down. We rather choose to have policies that restrict access, which is explained in further detail in chapter 3 and chapter 4.

Priorities can resolve conflicts when two entities want to access the resource at the same time. The Datalog policies are then evaluated by formulating Datalog queries. Dixon et al. stress that this policy language has a good usability, as it can be translated into English sentences, and can be easily visualized. It might be too limited for some setups, but they conclude that for the HomeOS it is able to express all relevant privacy rules [13].
The idea of transforming the raw policy into English sentences is useful and is explored in chapter 3.

2.2.3 Data Pre-Processing

The previous sections list solutions that consider multiple aspects for privacy and how policies can allow or deny access to data based on those aspects. A functionality, that only few systems even consider, is data pre-processing, or data obfuscation. With data pre-processing, the data is altered and adapted to the users understanding of privacy, before it is shared with others. This helps in privacy concerns because by altering it, the data needed by a service is still available, it only is less precise. This way, the

functionality of a service can be kept unchanged, while the owner of the data has no privacy concerns with this level of data sharing.

An example for this is presented by Atzori et al., where a sensor shares only the approximate location of the individuals it senses. This is a trade-off that might be sufficient for the processing services [1].

Patrick describes a process that can use PEPs and PDPs to assemble a privacy conform response to a request. First, the request is sent to the PEP, which asks the PDP for the correct access rights on the requested data. The PDP returns them, and the PEP obfuscates any information that is not accessible by the requester. This is either done by removing it from the response, or by encrypting it. The decision on what data is included in the response is made based on policies. The patent proposes using XML queries or XML Path Language to remove or alter inaccessible data [30].
Using policies to declare how data should be altered is a useful approach, since we can thereby integrate it into our access policy.

Davies et al. propose a privacy enforcing architecture that comprises a mediator that is able to obfuscate data aspects based on privacy policies. They name a video feed as an example, where a few still images can be gathered from the feed, thereby keeping the complete video private while still allowing some usage of the data. Another example is to blur the faces of all people, or only the faces of a specific set of people [10].
Those obfuscation examples, and the pre-processing architecture are useful examples for data obfuscation.

Taking these examples, more privacy related data actions can be thought of that can be applied to the data in order to protect sensitive information. We explore more possibilities in chapter 3 and chapter 4.

2.3 Relevant aspects of the VSL

The Distributed Smart Space Orchestration System (DS2OS) framework developed by Pahl [28] is an integrated approach to the fragmented smart device market. It offers the possibility to connect smart devices from different vendors to a smart space, overcoming vendor silos. This section explores the most relevant aspects of the DS2OS.

The Distributed Smart Space Orchestration System (DS2OS) consists of a VSL middleware, a Smart Space Store (S2Store), and VSL μ-services. The VSL is implemented in Java.

In order to gain an overview of the functionality of the system, the first sections explain the connections between the individual central entities.
The services of the VSL are data centric, which means that services do not offer a certain functionality, but instead offer data of a certain type. How the data and the μ-services are managed in the VSL is explained in subsection 2.3.1.
The S2Store is an App store for smart spaces and is described in greater detail in subsection 2.3.2.
The DS2OS can consist of multiple VSL smart spaces, in other words, DS2OS sites. Those sites each have a SLSM, which is described in subsection 2.3.3.

Understanding how data is managed in the DS2OS requires us to examine the data format, which is described through context models. They are explained in subsection 2.3.4.
The data is stored and managed by KAs, which are introduced in subsection 2.3.5.
Services in the VSL are wrapped and distributed as a service package, which is explained in subsection 2.3.6.
Since this thesis introduces a more fine-grained access control to the DS2OS, we look at the existing access control mechanisms in subsection 2.3.7.
In subsection 2.3.8, the functionality of search providers is explained, as they provide the central service of discovering context.

2.3.1 VSL

The Virtual State Layer (VSL) is a distributed Peer-to-Peer (P2P) network of entities called KAs, which manage the context of the different services. It can be described as a distributed operating system of the smart space [28]. The KAs are central for the implementation of access management and therefore vital to enforce more fine-grained access control based on privacy policies. See subsection 2.3.5 for a detailed description of the functionality of KAs.

The VSL is highly dynamic and extensible, as it is a μ-middleware. A μ-middleware provides basic functionality, to which new services with new functionality can be added during run time. Since the μ-middleware takes care of the smart space management, the programmer doesn't have to worry about this and thereby simplifies the development process of new services. It has the central task of provisioning context to the services in the smart space. New services are added via a unified interface whereby the interoperability between services is guaranteed [28]. The unified interface in the VSL are context models, which are further explained in subsection 2.3.4.

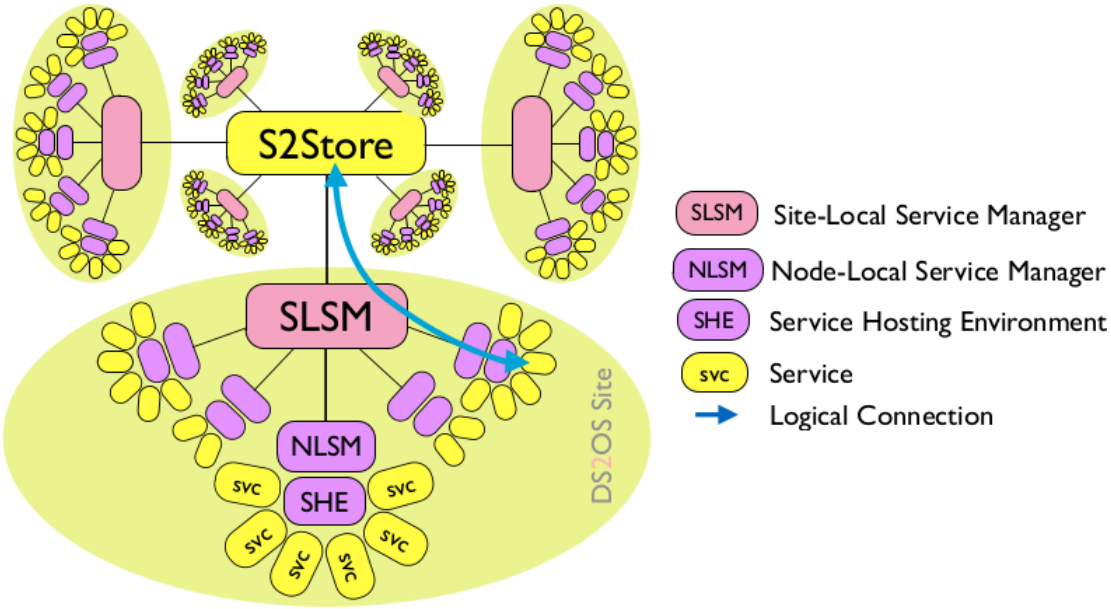

Figure 2.3: Topology between the App Store and the DS2OS sites (from Pahl [28])

2.3.2 S2Store

Understanding how a service is installed and used in the DS2OS is required in order to assess where the extended access control is implemented. One central, global entity to the DS2OS is the Smart Space Store (S2Store). It functions like a smartphone app store, as its task is to manage service executables (jar-files in case of the VSL) globally. The service executable are contained in a service package, which we explain in greater detail in subsection 2.3.6.

Developers upload their new services as service packages to the S2Store, where they are made available to everyone. See Figure 2.3 for a schematic view of the functionality of the S2Store. When some node in the VSL wants to install a service that is available in the S2Store, the service package is downloaded from the store. The blue arrow in the figure shows how the service is instantiated in the local site. The Site-Local Service Manager (SLSM), that is part of this process, is explained in the next subsection (subsection 2.3.3).

Since the manifest of services is considered as the location where the policies are declared, the process of service distribution is explained to such extent. We also intend to offer the possibility for default policy rules, which are uploaded with the service package to the S2Store.

Part of the S2Store is the also globally unique CMR, which takes care of managing the context models. The CMR offers crowdsourced convergence mechanisms for the developed services and the context models, and offers an information exchange platform for service developers and users. The CMR stores and distributes context models which can be used in multiple smart spaces [28]. The functionality of the CMR is further explained in subsection 2.3.4.

2.3.3 SLSM and NLSM

As the name suggests, the Site-Local Service Manager (SLSM) performs management tasks for the local DS2OS site. One SLSM instance is run on one of the nodes of a site, managing the site's resources and services by coordinating the Node Local Service Manager (NLSM)s on each node. A NLSM manages all the VSL services on its node, by monitoring, starting, stopping, or pausing them.

When a service is to be deployed on the local site, the corresponding service package (see subsection 2.3.6 for more detail) is downloaded from the S2Store into the service repository of the SLSM. The SLSM then redistributes the service package to a chosen node in the site which stores the package in its NLSM. It does the same in case a service update is available at the S2Store [28].

2.3.4 Context models and the CMR

The first sections of this chapter describe the contexts that are used in other implementations and related literature. This section explores how context is described in the VSL, and how the semantic representation of the entities is structured.

Context models describe the information about the real world (i. e. context) that is produced by a service, using XML as markup language. They structure the context and thereby create a description of the world. A description of a world is an ontology, and is usually defined as *subject, predicate, object*. In the VSL, the Context Model Repository contains the ontology, since it contains the context models of the VSL. The identifiers of the CMR are the *subjects* and *objects*. The *predicates* are matched by deriving and composing, which is explained in the next paragraphs.

The basic data types, with which all further data types can be created, are */basic/text, /basic/number, /basic/list,* and */basic/composed*. A new data type that inherits from those basic types can for example be a boolean value. The following context model can be *derived* for this purpose:

```
<boolean type="/basic/number" restriction="minimumValue='0',maximumValue='1'">
  0
</boolean>
```

Listing 2.1: Context model for a boolean value

The creation of new data types is not limited, for example the previously shown boolean value can be used to describe the functionality of a lamp. A lamp can either be on, or off, and to represent this, the data type */derived/boolean* is inherited, and the initial state of the lamp is set to "0".

```
<lamp type="/derived/boolean">
  0
</lamp>
```

Listing 2.2: Context model of a lamp

Composing a new context model can be done by defining a new ModelId that contains an existing context model, and can be multiple nodes of different types. An example is:

```
<specialLamp type="/basic/composed">
  <lamp type="/derived/boolean">0</lamp>
  <greet type="/basic/text"></greet>
</specialLamp>
```

Listing 2.3: Context model that is composed of a lamp, and a text node

The context models can inherit from multiple other context models which allows a node to have different functionalities. To stay with the lamp example above, our lamp could also have the possibility to change its hue. The context model then looks like this:

```
<betterLamp type=".../hueChangingLamp, .../lamp">
</betterLamp>
```

Listing 2.4: Context model of a multi-purpose lamp

The lamp can then be used either as a normal lamp, which can be switched on and off, or as a color-changing lamp where the hue can be set.

Since context models describe the world, a semantic search of the services and the represented data can be performed, which is elaborated in subsection 2.3.8.

In order to understand how new services are created from those context models, we have to understand the CMR. The Context Model Repository (CMR) is a global repository and part of the S2Store. It is responsible for storing and exchanging the context models of the DS2OS to the VSLs, making them accessible to everyone. When a service is deployed, the corresponding context models are instantiated from the global CMR into the VSL context repository.

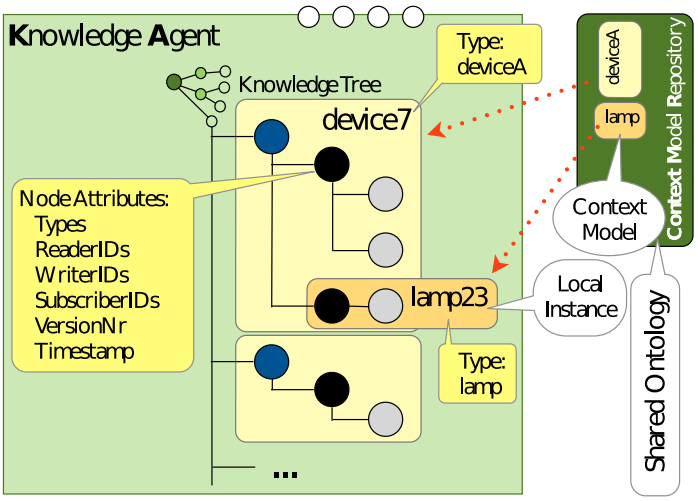

Figure 2.4: VSL meta model with global CMR and local KA (from Pahl [28])

By involving the global CMR in the service instantiation process, the publication of context models is encouraged, which in turn helps with crowdsourcing services. Through this, the shared context models can be found by other developers and reused.

When a new service is developed, the developer uploads the context model to the CMR. Context models have a unique ModelId, which ensures that the context models are consistent throughout the whole network. The unique id also ensures that the context models are consistent in the whole smart space, which in turn makes services portable. In the local site, where the service is to be registered, the context model gets loaded from the CMR into the local CMR service. A new instance of the context model is instantiated and is assigned a new identifier. The CMR service is a caching proxy for accessing the CMR. The created context model instance is then associated to a KA and used to create a VSL node. See Figure 2.4 for an overview of this functionality.

The context node is of a certain type, described by its context model. In the case of Figure 2.4, it is of type *deviceA* and is assigned a new name, in this case *device7*. The node is instantiated based on the context model in the CMR of *deviceA*. Since the context model of *deviceA* contains a node of type *lamp*, a node of type *lamp* is automatically created inside of *device7*.

The nodes also have further attributes, that are passed via the context models. Besides the data types, ReaderIDs and WriterIDs are passed, which declare the access rules. How the access management works, using these attributes, is shown in subsection 2.3.7.

The type of the context node is used to identify the data type (e. g. number, text). Besides the data type, the type also declares the functionality of the context node, since

the ModelId points to a context model in the CMR. This globally stored context model is associated with a functionality (e. g. door, lamp, ...) and therefore describes the functionality of the context node.

The described VSL types represent an inheritance relationship. They declare that a certain context node *is-a* certain type of service (e. g. lamp). As each site is uniquely named, and the context nodes are also assigned unique names, hierarchical addressing is implemented. Through the hierarchical addressing, a composition relationship can be described, as it can for example be seen that node A *has-a* lamp service. Those two relationship descriptions make the CMR an ontology for the VSL. The ontology is extensible, self-managing, and collaborative, since the CMR stores the context models. We make use of this ontology, as it is explained in subsection 2.3.8. This description of the environment also relieves us of the task of defining a new ontology for the data, which we need when formulating the policy language.

2.3.5 KA

The nodes of the P2P middleware are called Knowledge Agent (KA)s. Sensors and other smart devices are attached to the KAs, A service that uses those smart devices is most often connected via services to the corresponding KA. The KA stores the information of the service in the form of context nodes in its context repository. The context nodes are organized in a knowledge tree (see Figure 2.4). Since the KA takes care of the data storage, the developer of a service doesn't have to worry about it.

The developer can further use the KAs for context routing, as the KA provides various methods to access the stored context. The main methods are *get, set, subscribe,* and *notify*. The nodes are addressable via the agentID, the serviceID, and the path that is described by the context model. The *lamp23* from Figure 2.4 could for example be a lamp on a smart mirror, and would be addressed via */bathroomAgent/mirror/lamp23*. See Figure 2.5 for a schematic view of the KA with the context manager and the context repository.

Since we have a distributed P2P network of KAs, where every KA provides different services, we need a way to discover those services to be able to access them. The KAs synchronize the node structure of their context repositories periodically, which forms the VSL μ-middleware. Through this, a KA has a local representation of the context available in the VSL, making context access more reliable since even during a connectivity loss the context is still available.

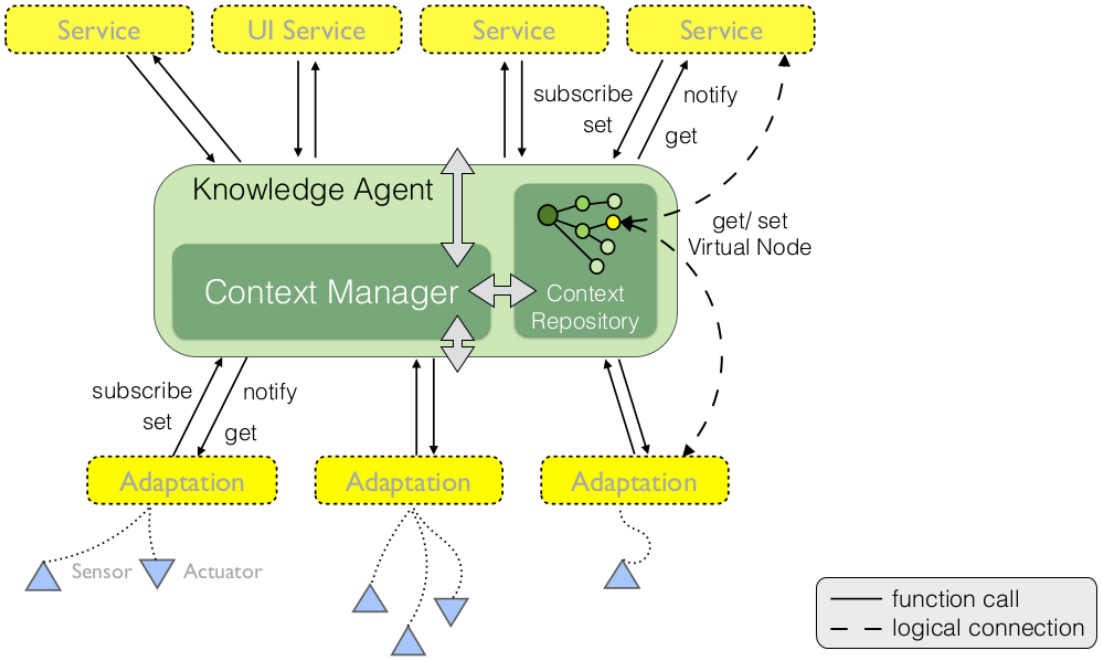

Figure 2.5: Context Management Architecture of the VSL (from Pahl [28])

2.3.6 Service Package

The service package is mentioned in subsection 2.3.2 and subsection 2.3.4. It is a container that packages the service executable (jar-file), the meta data of the service as a service manifest, and the service certificate.

The service manifest protects the service executable by storing a cryptographic hash, so that tampering with the jar-file is noticed. It further contains the unique name of the service, the developerID, a version number, and the context models that are required, obsoleting, or conflicting the service. For further service management, the computational resource requirements, and resilience requirements are included.
The service manifest is of interest to this thesis, since it can be extended to include a privacy policy. Storing the policies in the manifest has the advantage that default policies can be distributed with the service executable, and therefore ease the usability of the policies. The decision on where to declare the privacy policy is discussed in chapter 4.

The service certificate protects the service manifest since it includes a hash over the manifest. The service package is created and protected using the developer's public key, and then uploaded to the S2Store.

2.3.7 Access management

In subsection 2.3.4 we mention that the context model passes other attributes of the context node, such as the ReaderIDs and WriterIDs. These IDs are used to determine which entities have access to the context nodes.

All basic context types have the accessID "*" in their *readerID* and *writerID* fields set, which grant access to any service:

```
<text reader="*" writer="*">
  Text
</text>
```

By default, all accessIDs are inherited from the data type the context node inherits from. So if a context model inherits from this basic text type, it is by default accessible by every service. To further understand how access can be limited with the use of these IDs and how this access management is enforced, we have to look at the general security architecture of the DS2OS.

In order to be able to decide whether someone is granted access, we have to verifiably know who that someone is. This is commonly solved by introducing certificates, which are issued and signed by a Certificate Authority (CA) to create a trusted validation of identity. In the DS2OS, each site has a CA, called Site-Local Certificate Authority (SLCA), where certificates can be issued for the entities of the corresponding site. Users, developers, and services have certificates, which hold required information in them and can assert the identity of the entity. We already mentioned the service certificate in subsection 2.3.6, and the next paragraph explains what a developer certificate is used for. In general, the user of a VSL site is issued a certificate by the SLCA and it contains certain accessIDs. Those IDs are group IDs and grant the user access to all services which require membership of this group.

Having a CA locally per site instead of a global one makes the VSL more resilient to connection loss and keeps the site management decentralized. Trust between multiple sites is done explicitly. Each site's SLCA uses a public-private key pair to sign certificates for the entities that wants to interact with the VSL. The SLCAs are considered as trusted entities.

As subsection 2.3.6 mentions, the service package is protected by the developer's public key, as it is used to create the service certificate. The service package is then uploaded to the S2Store, where it is verified using the public keys of the developer, which is stored at the S2Store. If this verification is successful, the S2Store signs the service package with its own public key, adds a signed certificate to the package, and makes the service

available to everyone. The S2Store is also a trusted CA, as its keys are initialized once at the beginning.

When the service is to be deployed to a local site, it is downloaded from the S2Store and the user can choose to keep the access groups as they are declared in the context model, or whether to restrict the access by defining different access groups. The SLCA then creates a new signed service certificate, in which the accessIDs the user chose are saved. Only services that are signed by the local SLCA can be run in a site.

Each context node has meta data, which declares the access groups the node belongs to. The access groups are also contained in the service certificate. Once an entity wants to access a context node, the accessIDs that the entity has in its certificate are compared to the accessIDs in the meta data of the context node. In the context model of the service, the developer declares the access groups that are allowed to perform get or set actions on the context. This is done via the before mentioned *readerID* and *writerID* fields in the context model. If the access mode is a *get* operation, the *readerIDs* are checked, if it is a *set* operation, the *writerIDs* are checked. If at least one of the accessIDs in the node are matched by the entity's ID, the access is granted. Since the access rights are stored in the meta data of a node, the access control can happen decentralized and faster. The accessIDs cannot be changed, since they are safely stored in the certificate, protected by the signature of the SLCA. If new accessIDs should be allowed, the new service package has to be signed by the SLCA.

The access groups that are used in the service certificates and the context models are stored on the S2Store. A developer can create a new access group identifier by uploading it to the S2Store. A description of the access group is handed in with this upload, so that the user who uses these groups to grant access to the new service knows who is included in the group.

The enforcement of the access rights happens on the KAs, as Figure 2.6 shows. The requests that are sent out and received at the KAs are passed through the Request Router (RR) and forwarded to the access control. If the access is granted, the Knowledge Object Repository (KOR) can be accessed, meaning the context can be retrieved.

So the KAs are the main components of the VSL, since they not only store the context, but also make it accessible to other KAs. We introduce the concept of PEPs and PDPs in subsection 2.2.1. In the DS2OS, the KAs act as both PEP and PDP, more specifically the RR is the PDP, since it stores and evaluates the policies, and the access control unit that sits between the RR and the KOR is the PEP, since it enforces the access decision made by the PDP, or in our case, the RR (see Figure 2.6).

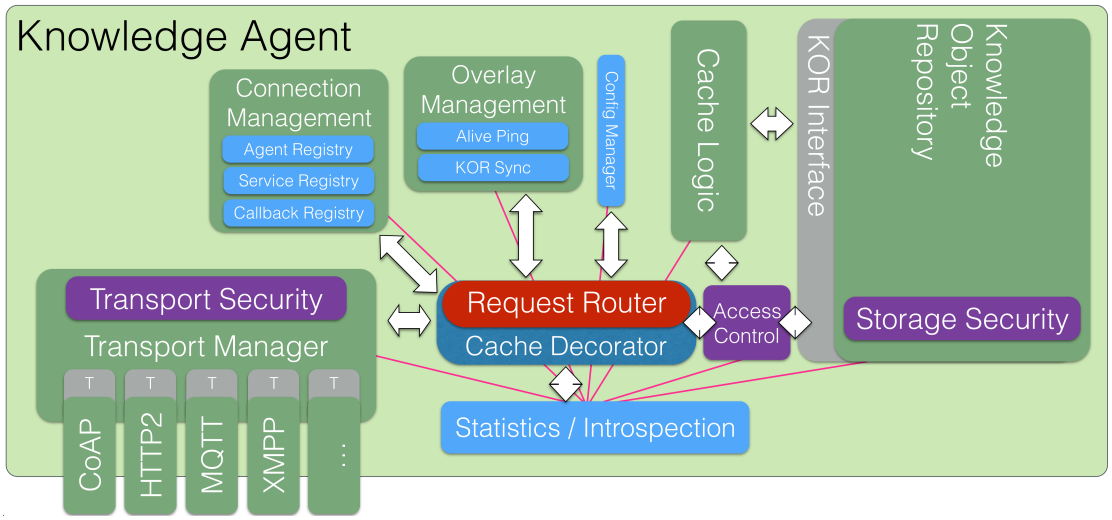

Figure 2.6: VSL system architecture (from Marc-Oliver Pahl, adapted)

2.3.8 Ontology and Search Providers

We explore the ontology of the VSL in subsection 2.3.4, where we describe the *is-a* relationship that is achieved through the context models and the therein declared data types, and the *has-a* relationship, which is achieved through the hierarchical addressing. Figure 2.7 shows the *is-a* relationship, and the way the real world objects are described using a domain ontology. This ontology is expressed in the VSL through context models, which are therefore a semantic description of the context nodes of the VSL.

One search provider that is already implemented in the DS2OS is the type search. The type search can be used to search the VSL for the location of all the context nodes that are of a certain type. The search is executed by performing the following *get*-operation:

```
get /search/type/basic/text
```

It searches for the nodes that are of type */basic/text*.

The result of the *get*-operation is the address of the context nodes that are of type */basic/text*, e. g.

```
/agent1/mirror/weather///agent1/welcome
```

These addresses can then be used to access the node and to use it for further functionality.

A search can also have multiple parameters that are passed. Separators between those parameters can be `?` and `!`, where `?` is the separator between parameters, and `!` allows for multiple context types to be passed. An example is mentioned in [28]:

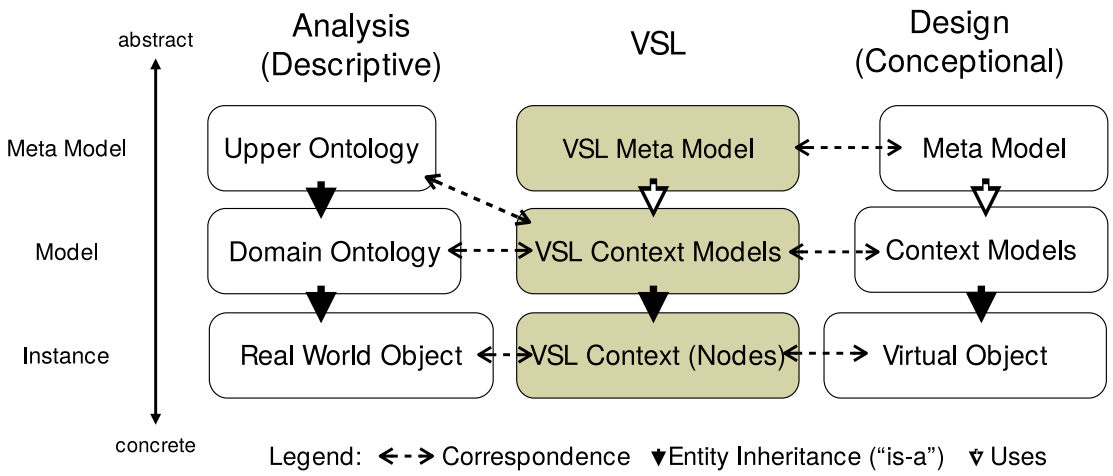

Figure 2.7: Context related terminology (from Pahl [28])

```
get /search/typeAtLocation/livingRoom/!/bathRoom/?/basic/number/!/basic/text
```

Multiple parameters can also be separated by a `&`:

```
get /search/location/key1=value1&key2=value2
```

Since we can describe the environment semantically through different ontologies by defining new data types in the context models, we can also have different search providers in the VSL. We can extend it to offer a location search provider for example. The location information of each context node could either be set in the context models of each node through type inheritance, or by saving location information in a separate database. The type inheritance has the disadvantage, that all existing context models do not inherit the location information, therefore have to be updated. The current implementation of the DS2OS therefore uses a separate database to store the location of the context nodes, since the location information also has to be stored in the case of type inheritance for caching reasons.

2.4 Summary

This chapter explores the functionality of access control management in the DS2OS and existing components for semantic evaluation in the literature. The focus of the next chapters is on how the privacy can be enhanced in the DS2OS and which constraints have to be met while introducing this new functionality.

A conclusion that we can derive from this chapter is that our prototype must support multiple privacy requirements (see section 2.1). They are numbered from <PR.1> to <PR.6>, and namely are *Conditions under which data is shared and the purpose of the request, Raising user awareness, Being adaptable to a high variety of devices, Handle rich data sets, Providing an understandable privacy configuration*, and *Striking a balance between privacy, performance, and data fidelity.*
They are providing a guideline for how the privacy policy can address important privacy aspects. Together with the relevant contexts listed in the next paragraph, these requirements are formed into functional, and non-functional requirements for our design in subsection 2.4.2 and subsection 2.4.1.

The list of relevant contexts we identify in this chapter is numbered from <C.1> to <C.8>. They are *Information, Entity, Interactions, Conditions, Time, Location, Relationship*, and *Attribute* (see section 2.2).
Those contexts let us derive which contexts have to be expressible in a policy language. They serve as wrapping terms for all possible sub-contexts might be relevant for a privacy policy in a specific domain. For example an entity can be a user or a device, location can be defined at various levels, from GPS data points to certain rooms in a building.

Data pre-processing is an interesting addition to the privacy policy, since it extends the focus of the policy from solely being about access control, to also managing the data set that access is granted to. Data obfuscation can use the ontology of the DS2OS to define data redaction methods for certain types (see subsection 2.2.3).
An example for this is that for a temperature sensor, one redaction method is to only publish the average temperature of the last month, or to share only the highest or lowest temperature measurements in a given time frame. We address this topic in chapter 3.

Since the DS2OS is a framework that is adaptable to various use cases, this thesis does not design a policy for a specific sub-context. It already features an ontology that can describe contexts at high flexibility through the context models, therefore we do not need to design an ontology, but can use the example ontologies of this chapter as a guideline for how the context of the DS2OS can be integrated.

As we discuss in section 2.3 (Relevant aspects of the VSL), the existing access control in the DS2OS is a binary one, that either allows a user to access a service or not. It factors in other context for the access decision, like the data, the operation (get/set), the entities and their groups, but it does not allow other parameters to influence this decision. The users are therefore limited in expressing their privacy understanding, and are forced to model it to the best extend via groups, which over time leads to a high

number of groups. Access is also either a complete access grant or denial.
It is better to allow the users to use the context of the system to express their understanding of privacy, because no new environmental description has to be created, as it currently done via groups. Instead, a policy can also include a check for the status of a lamp, if it is switched on, access is granted, or denied if it is off. Modeling this via access groups is almost impossible.
Including data obfuscation also makes the data set adaptable to privacy needs, as access does not have to be either grant or deny, but can be a *grant to blurred pictures* or *grant to average of last week's measurements.*

Time and location are contexts that are used in many systems for privacy policies. We focus on using time and location as new contexts for the privacy policies on top of the existing access control, since they are the most common context. We cannot predict that the privacy requirements in a DS2OS system stay the same over time, so we have to design our policy language in a way that it remains extensible for other contexts to be included. The functionality of the DS2OS supports this extensibility, since context discovery is done via search providers, which can be added to the system and can semantically search for certain contexts. The possible design choices are elaborated on in chapter 4.

The next two sections summarize the findings of this chapter into functional (subsection 2.4.1) and non-functional requirements (subsection 2.4.2). The previously defined requirements and contexts are integrated into them.
The last section maps the requirements onto the research questions that are listed in section 1.2.

2.4.1 Functional requirements

Notation information: In order to summarize the most relevant functional requirements, and to later reference back to them, we numerate them in the following manner: <FR.X>

As we explain in subsection 2.3.7, the access control in the DS2OS is done via the *readerIDs* and *writerIDs* in the context models. The access groups are set by the developer of the service and can be adapted by the user who instantiates the service in the VSL site. This way the user can chose different access groups, but cannot incorporate other factors beside group membership into the access decision process. Extending the existing access control of the DS2OS is the goal of this thesis, and therefore a functional requirement.

<FR.1> *Multi-factor privacy decision.* Including attributes that describe the system in a privacy policy allows for a fine-grained privacy declaration. It therefore is a relevant extension to the access control in the DS2OS to evaluate other contexts in the access decision process, transforming the binary access decision into a multi-factored one.

Time and location are common privacy contexts in the literature, and are therefore a starting point for implementing more contexts in the access policies [13, 20, 31]. They are also such basic descriptions of any environment, that they are probably featured in most use cases for the DS2OS.
An example for location based access control id that the lights in a room are fully accessible by all devices in the same room. Only the device or house owner is allowed to change the state of the lamps from other locations in this scenario.
An example for time based access control is a smart door lock, which allows access to cleaning staff from 9 to 10 on Fridays.

This also integrates all of the contexts (<C.1> - <C.8>), since being able to address all of the contexts of the system makes the policy language as fine-grained as it can be. The privacy requirement for expressing the conditions under which data is shared (<PR.1>) is integrated by including all attributes that describe the system into the access evaluation. The privacy declaration also becomes adaptable to a high variety of entities (<PR.3>).

<FR.2> *Data obfuscation.* The decision to share data should not only be based on whether a person fulfills certain criteria to be able to access the full data set. Instead the possibility to alter the data set depending on environmental context is beneficial, as it is explained in subsection 2.2.3. This allows for a more fine-grained data sharing decision, and keeps the usability of the system high, as data is still accessible, just not to the full extent.

This integrates the privacy requirement for making the conditions under which data is shared expressible (<PR.1>), at an even greater level of granularity as the previous functional requirement. This way, the richness of the data set can also be handled appropriately, as it is necessary for compliance with <PR.4>.

A major advantage of the DS2OS is the self-managing organization of the nodes, where new services can be added by a user without having to take care of the adaptation into the existing system. The KAs take care of context access and cache the context node structures of the VSL locally, therefore do not rely on a central node for context discovery.

<FR.3> *Autonomy.* The distributed nature of the DS2OS is a key feature and provides several benefits and is therefore declared as a key requirement. It is not beneficial to the system if the prototype introduces a single point of failure, e. g. in case of connection loss. Caching previously queried information can be beneficial to maintain this autonomy.

A balance (<PR.6>) between a fine-grained policy, and a functional and autonomous system is considered in this requirement.

<FR.4> *Adaptability.* The existing DS2OS implementation is aimed at providing a highly adaptable system for various use cases. In order to maintain this adaptability our design makes use of the existing ontology of the system.

In the DS2OS, the search providers provide a functionality that keeps the privacy policy extensible for further contexts that is added to the privacy description later. The privacy requirement for adaptability is matched directly in this functional requirement (<PR.3>).

<FR.5> *Reuse.* The DS2OS already provides an ontology and a semantic discovery functionality. Reusing the existing ontology minimizes the workload of this thesis, and ensures a better understandability for the users, since no completely new concept has to be understood by the programmer.

This integrates the privacy requirement for understandability (<PR.5>).

2.4.2 Non-functional requirements

Notation information: In order to summarize the most relevant non-functional requirements, and to later reference back to them, we numerate them in the following manner: <NFR.X>

<NFR.1> *System complexity.* Maintaining the complexity of the system keeps it functional and understandable. Designing a solution that follows the logic of existing functionalities of the system (such as the access granting logic) also increases understandability for the programmer.

For example since the current access rights are set per node and are not inherited by the child nodes. It is therefore helpful to low system complexity when the new access policy rules also do not influence the child nodes, but instead are relevant only for the current node.

Henze et al. state how experienced programmers and privacy experts should be able to define their privacy understanding at a fine-grained level of detail [17].

<NFR.2> *Expressiveness.* An expressive privacy policy can represent a user's privacy understanding to the full extent.

This relates to the privacy requirement for defining the conditions for which data is shared (<PR.1>).

An expressive policy is useful for privacy experts, but making the policy declaration understandable for privacy novices has to be considered in the policy design, as the system is not only used by experts.

<NFR.3> *Understandability.* A privacy idea can only be expressed if the policy formulation of privacy policies is easy and understandable. The understandability of the access policies also ensures their usage, since only a functionality that is understood by the user is employed to the full level that it was intended.

This matches the privacy requirement for understandability (<PR.5>), and even covers the requirement for user awareness of privacy relevant information that is processed, as an intuitive policy declaration helps the user (<PR.2>).
Introducing default policies is a good approach to help the user with the service setup, as default policies can capture the most common privacy needs. This default privacy policy can then be extended or reduced by the user.
Using natural language is recommended to increase the understandability of the privacy rules [20]. Having a graphical user interface for policy configuration also helps with allowing non-experts to define a fine-grained, individual privacy policy [13].

2.4.3 Linking to research questions

The functional requirements for *multi-factor privacy decision* (<FR.1>), *obfuscation* (<FR.2>), and *adaptability* (<FR.4>) can be mapped to the research question **RQ.1**, which explores solutions that use an extended access control.

Research question **RQ.2** requires that the policy is understandable and expressive. It covers the non-functional requirements *expressiveness* (<NFR.2>), and *understandability* (<NFR.3>). *Reuse* (<FR.5>) also plays into this topic, as a reuse of functionality helps the programmer to recognize and understand the inner workings of the system more easily.

System complexity (<NFR.1>) and *reuse* (<FR.5>) are requirements that are related to **RQ.3**, which expresses a need for maintaining complexity in the DS2OS.

The performance of the system is at the center of **RQ.4**, and can be linked to the requirement for continued autonomous operation (<FR.3>).

Chapter 3

Related Work

This chapter presents findings from the literature, that are related to our problem, and that can answer some of the Research Questions identified in section 1.2.

In the first section (section 3.1) we look at how the standard access control, which is usually done based on groups (see MAC, DAC, RBAC, etc.), can be extended to cover more contexts for defining and evaluating an access policy. We want a more fine-grained access control which also provides the possibility to alter the level of detail for the returned data. This aims at answering the research question for an extended access control (**RQ.1**).

The section 3.2 analyzes existing policies to find a solution that conforms to the requirements identified in subsection 2.4.1 and subsection 2.4.2. The goal of this section is to answer how an expressive and understandable policy can be designed (**RQ.2**).

Section 3.3 lists good working examples for managing the trade-off between performance and privacy. This section covers the research questions for low system complexity (**RQ.3**) and for maintaining a good system performance (**RQ.4**).

We summarize the findings in a tabular overview in section 3.4.

The existing solutions analyzed in this chapter are **ABAC**, **XACML**, **Bark** by Hong et al., and the work by **Davies et al.**. The solutions are presented in a short *Overview* when they contribute to the challenge of a section. Since some solutions provide contributions to multiple challenges, later sections refer back to the presented solution via a *Conclusion* section.

3.1 Extended access control

In section 2.1 we explore a great number of possible contexts that are relevant for privacy, both in general, and in pervasive computing. The conclusion from this section is that beneath the usual user group based access control (like in e. g. RBAC), contexts like location and time are commonly employed for the privacy policy declaration. Privacy needs vary greatly from user to user, and from environment to environment. Therefore it is reasonable to introduce more aspects which can be included in the privacy policy definition, so that the users can declare the level of detail they understand their privacy at (see the functional requirement for a multi-factored privacy decision <FR.1>).
The goal of this thesis is to extend the binary access control in the DS2OS to a more fine-grained one. Currently, access is either granted if the access restrictions expressed through the `readers` and `writers` fields are evaluated to true, or denied if otherwise. By introducing data obfuscation (see functional requirement <FR.2>) we can determine the level of granularity that the shared data has to have.
Our solution is not tailored to one specific use case but rather is adaptable to various domains. We look at solutions that can be extended later on by plugging in new functionality, or by addressing external sources. See the functional requirement of adaptability for a specification (<FR.4>).

By addressing these topics, this section finds solutions for an extended access control (**RQ.1**).

3.1.1 Multiple contexts

The traditional access control is based on user identity, or basic attributes like group membership or roles. This section presents the solutions in the literature about how more access policies can be added to the existing access control, which incorporates more context for this decision. See the functional requirement for a multi-factored privacy decision: <FR.1>.

3.1.1.1 ABAC - Overview

With Attribute-based Access Control (ABAC), all kinds of attributes can be used for access policies. This is done by associating attributes with the predefined attribute types *user, subject, object, environment, policy*, and *actions*, as we describe in subsection 2.2.1. Like this, the decision whether to grant or deny access can use arbitrary attributes in ABAC [21]. Hu et al. note that ABAC is only limited by the level of granularity that the

attributes and the computation language are able to express. This removes the need to define individual relationships between each entity.
Hu et al. also note that one of the main advantages of ABAC is that attribute based policies do not require prior knowledge about the individual entities which may join the network later, since not individual relationships between entities are described, but entities are rather assigned the appropriate attributes and thereafter the correct policies apply to them [21].

The access control mechanism of ABAC needs to gather the attributes which are required for the policy decision, which can be done using a context handler, or a workflow coordinator. To evaluate the policy elements with the gathered, available information, a Policy Decision Point (PDP) is often used. A Policy Enforcement Point (PEP) enforces the decision made by the PDP [21].

3.1.1.2 ABAC

The high adaptability of ABAC is a working example to fulfill the corresponding requirement of this thesis (<FR.4>). The existing access control in the DS2OS is one, where access is granted based on the group membership of the entities. Since the DS2OS has a semantic description of the environment, we do not have to be limited to this basic access control, but can extend it to include all imaginable context. An example that is named before is that the state of a lamp can be used to determine if access should be granted, e. g. if the lamp is switched on, access is granted.

Using adaptable, dynamic attributes to refine the existing access policies makes sense to achieve a privacy policy which is adequate for the specific environment.
We look at the architectural benefits of ABAC in the following sections again, but for now it can be summarized that we already have an attribute based ontology, which allows us to use ABAC, but we still have to see how those attributes can be integrated into a policy.

3.1.1.3 Hong et al. - Bark - Overview

Hong et al. introduce the policy language *Bark*, which uses natural language to express privacy requirements. A speciality of their approach is that communication between all entities is default-off, and sharing of data has to be enabled explicitly. This whitelisting approach makes the system highly secure and privacy-friendly.
They also propose a two-factor authentication scheme for granting access to information.

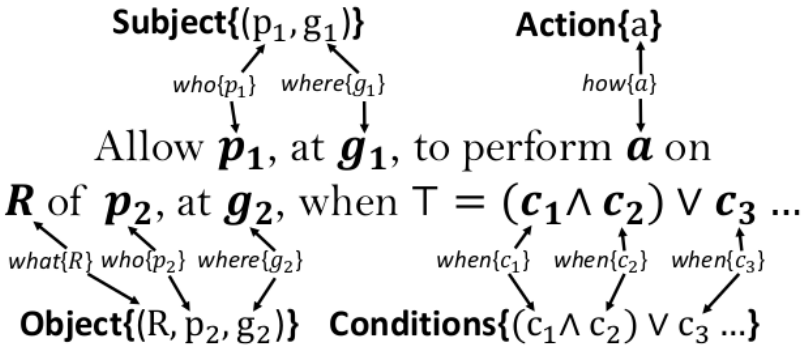

Figure 3.1: The sentences are constructed using the natural questions (from Hong et al. [20])

The second factor is a non-network context, or a user making decision in real-time. This can be done via SMS, passwords, or external oracles [20].

Bark is constructed using the natural questions **who, what, where, when**, and **how**.
Who are the *devices* and *apps*, and they can have an *owner*.
What are the *services*, that the **who**s offer. Those two questions can be mapped to the context <C.2> - entity.
Where describes the topology of the *environment* or the *network* (related to context for location - <C.6>).
When is either a *time restriction*, a boolean function, or an oracle to evaluate *functions* (<C.5> - time, <C.4> - conditions).
How describes a command or an *operation*, and can take many forms, from HTTP, over Bluetooth, to TLS/UDP exchanges (<C.3> - interaction).

The sentences consist of a **subject, object, action**, and **condition**.
Subject can be a **who** or **where**.
Objects can be **who, where,** or **what**.
Actions are the **how**s.
Conditions are the logical connectives $\wedge, \vee$, and $\neg$, which are used to link the **when**s together to create a algebraic expression.

These building blocks can then be used to construct a policy, as Figure 3.1 shows.

The set of rules is connected by logical disjunctions ($\vee$), meaning that if at least one rule is matched, access is granted. If none is matched, no access is granted.
Bark also allows for grouping of principals, the gateways/locations, and services. This way, they can be addressed as one.
Wildcards are allowed in Bark. The problem of groups and wildcards is that it has to be differentiated whether the rule must only apply to the requesting entity, or to all.

Hong et al. approach this problem by adding *all* and *one* annotations to the groups and wildcards.
The individual rules can be activated by the user, and Hong et al. suggest that policies can be created that are specifically adapted to a certain type of device, for example by experts [20].

The solution by Hong et al. manages the policies centrally, but enforces them locally on the data plane. In order to cope with an unreliable network, where the centrally stored policies might not be reachable, they introduce caching and leases. Gateways, that sit in the network of the devices, store a copy of the central policies locally for a certain time, defined in the leases. After this time the policy has to be renewed, and if this is not possible, the corresponding access is returned to the default, which is no access. The responses from the external oracles can also be cached for a time specified by the oracle.
They also use access revocation, but it is not explained in more detail [20].

3.1.1.4 Hong et al. - Bark

We identify some aspects of Bark that are already implemented in the DS2OS:
Since the policies are defined for a service, the *requester* of that service is identifiable in the VSL, therefore being the **subject - who** in this example. The location of the requester, i. e. **subject - where**, is not yet implemented in the VSL and has to be added by our design.
The requested operation, either *get* or *set*, corresponds to the **action** that is to be performed in this example.
The *requested service* on a KA can be mapped to the **object - what**, the *KA* being the **object - who**. Again, the location of the **object - where** is currently not integrated in the VSL.
Existing **conditions** in the DS2OS are the *group based access restrictions* declared in the context models.
Valuable extensions to the design of our policy are therefore more fine-grained time and location evaluations.

The grouping of entities described in this paper is already partly established in the DS2OS. The access control is done by allowing certain groups to perform read or write operations. Grouping of attributes in general might be useful, but it already can be done with the existing DS2OS solution, since all, lets say, fancy lamp modification are derived from a basic type for lamps. Managing the attributes, or data representation in case of the DS2OS, is the responsibility of the DS2OS, and not part of this thesis.

Wildcards, as proposed by Hong et al. are not something we consider for our design, since currently no need arises for defining those. In the DS2OS, requests by specific entities reach specific services. Since the policy designed in this thesis is declared per service based on attributes, we can use the hierarchical structure of the context models. Lets say we want to define a policy for all hue-shifting lamps, we declare that requesters have to have the attribute type *hue-shifting lamp*.
We are focussed on defining restrictive rules, instead of explicitly naming all allowed traffic, like Hong et al. do.

3.1.1.5 XACML - Overview

We briefly introduce eXtensible Access Control Markup Language (XACML) in subsection 2.2.1. It consists of a policy language and a request/response language. The policy language and the request/response language of it are both written in XML.

The policy language defines the conditions under which access is granted. Sun Microsystems highlight that policies are generic and therefore one policy can be used for different applications [36]. Another advantage is that policies can refer to other policies, thereby making them distributed. Thereby no one central policy storage facility is needed, instead XACML collects the individual results from all policies and combines them into a final decision.

The request/response language describes what entity wants to access a resource, and the action that should be performed in a request, and interprets the returned response. This response is either Permit, Deny, Indeterminate (due to some error it is not possible to return a clear decision), and Not Applicable (the addressed service cannot answer the request). The entities involved in the request/response messages are the PDP and PEP, as we explain in subsection 2.2.1.

We go into more detail on the policy language of XACML, since this is the focus of this thesis.
The XML root is a Policy, or a PolicySet. A PolicySet can contain other Policies, PolicySets, or remote policies references. A Policy is an access control policy, which can contain a set of rules.
The possibly multiple rules in a Policy(Set) have to be evaluated to form one final access decision. XACML uses different combining algorithms for this, one example is a *Deny overrides algorithm*. It evaluates to Deny if no Permit is in the evaluation result, or if any evaluation returns Deny. Six other standard combining algorithms are available, namely *Permit overrides, First applicable, Only one applicable, Ordered deny overrides,*

Ordered permit overrides, Deny unless permit, Permit unless deny. The possibility to define a new algorithm beneath the seven standard ones is given.

A part of the policy is a tag called Target, and it is used to define who the policy applies to. It consists of subjects, resources, and actions. An example of a Policy that is matched to all requests looks like this:

```
<Policy PolicyId="Example" RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-
    combining-algorithm:deny-overrides">
    <Rule RuleId="LoginRule" Effect="Permit">
        <Target>
          <Subjects>
            <AnySubject/>
          </Subjects>
          <Resources>
            <AnyResource/>
          </Resources>
          <Actions>
            <AnyAction/>
          </Actions>
        </Target>

        <Condition>
          <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:time-greater-
              than-or-equal"
            <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:time-one-and
                -only">
              <EnvironmentAttributeSelector DataType="http://www.w3.org/2001/
                  XMLSchema#time" AttributeId="urn:oasis:names:tc:xacml:1.0
                  :environment:current-time"/>
            </Apply>
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#time">
                09:00:00
            </AttributeValue>
          </Apply>
        </Condition>
    </Rule>
</Policy>
```

Listing 3.1: Example XACML Policy that allows access to all entities if the current time is after 9 a.m. (adapted from Sun Microsystems [36])

The conditions declared in those tags are evaluated using boolean functions, and if they are met, then the corresponding Policy(Set)s, or rules apply. XACML also uses the target descriptions to index the policies, making them easily findable.
Mapping the request to the target tags of the policies is the task of the PDP.

And finally, contained in those layers of policies are the Rules. Most rules contain a Condition, which is also a boolean function. If this function evaluates to true, then the Effect that is specified for the Rule is returned. This can be Permit or Deny. In our example policy above it is Permit.

Since XACML is an implementation of the ABAC functionality, XACML can also express all sorts of attributes. It uses XML to express the type of the information. Taking time

as an example, there are multiple types and functions associated with it in XACML. There are *time, date, dateTime, dayTimeDuration* and *yearMonthDuration* for data types in XACML. The data format of those types are `yyyy-mm-dd±tzdiff` for *date* (tzdiff is the time zone difference to the UTC), *time* is defined as `hh:mm:ss±tzdiff`, *dateTime* has the format `yyyy-mm-ddThh:mm:ss±tzdiff`, *dayTimeDuration* is of format `PdDThhHmmMss.sS`, and *yearMonthDuration* is defined as `PyYmmM` [2]. Capital letters are fixed characters. `P` indicates that the declared time is a period.

There are also many functions associated with those types, like *time-equal, dateTime-add-dayTimeDuration, time-in-range, time-greater-than-or-equal, time-is-in, time-intersection, time-at-least-one-member-of* [35]. We can use those time types for our time based privacy policies. The functions are useful as an example for evaluating the time based privacy policies.

3.1.1.6 XACML

The fine-grained and extensible the language of XACML is a useful reference design. As the example of time shows, different types are declared and evaluable with useful functions. Reusing this existing framework is helpful as it saves us time in coming up with the functions to evaluate the different data types.
However, the DS2OS already describes the system semantically, therefore the definition of the data types is superfluous and only creates more complexity for the DS2OS. Only the functions that link the different data types and can compare them are useful for our design.

3.1.2 Obfuscation

Introducing privacy in the DS2OS aims at designing a more fine-grained access control, with rules that use more contexts. Beneath this access control, data pre-processing is identified to increase the privacy protection and is part of the intended design. The access control is moved from a binary *allow* or *deny access* policy to a fine-grained *allow access, allow access to obfuscated data*, or *deny access* policy.
This also increases the performance of the system since the data is still available to the service, it is not held private just because it contains sensitive elements. Instead, the data is obfuscated or denatured to a level of abstraction where the owner no longer considers it a privacy threat to share the data. This reduced data set can still be expressive enough for a service to operate on, so the service functionality is not compromised, and the user's privacy can be protected.

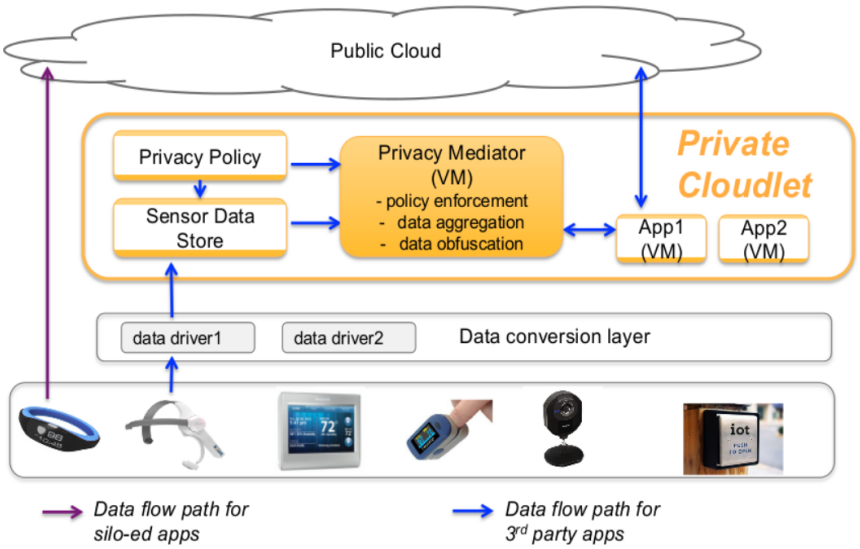

Figure 3.2: Privacy architecture (from Davies et al. [10])

Possible solutions for how data can be obfuscated are explored in this section. This relates to the functional requirement for obfuscation: <FR.2>.

3.1.2.1 Davies et al. - Overview

Davies et al. introduce a solution consisting of a local cloud, called cloudlet, which stores the data of the sensors, contains a set of privacy policies, and operates a privacy mediator (see Figure 3.2). The mediator is responsible for storing and aggregating data, and for enforcing privacy policies with a possibility for obfuscating the data. The mediator can be specific to a certain data type, or for a sensor class. If proprietary data formats are used, a conversion layer might be needed before the mediator can act on the data. Mediators can be developed by third parties, like an open source community, or experts. This makes the mediators with their privacy policies understandable and easy to use. Davies et al. suggest that this can spark off an industry branch for developers of trusted mediators and thereby trustworthy privacy instances, like anti-virus software works nowadays. Trust in those externally produced mediators can be ensured through certificates for example. Through shared mediators, the privacy policies can also be shared, thereby allowing for reusable privacy policies for a specific application class, or specific devices [10].

The proposed denaturing mechanisms of the mediator can vary greatly, since the functionality depends on the type of the data, or the sensor. Davies et al. name different ways, in which the sensor data can be denatured in order to make it less sensitive.

Different levels of granularity greatly depend on the type of data:
Images and videos can be blurred, and if face recognition is possible, the obfuscator can be told to blur the faces of certain people.
Sensor reading can be left out or only represented at a coarse granularity level during a specified time interval.
Summaries of the data can be created, either by temporal or spatial context: The temporal summary can be a view of a day, a week, or other time intervals, summarizing the minimum, maximum, average, totals, or other statistical evaluations of the data produced in the time frame. Spatial summaries can abstract the data to a coarser level, for example GPS data can be summarized to ZIP codes.
The decision of data granularity has to be balanced in order to keep the value of the data high enough for data reasoning. A useful approach is to make it dependant on the context [10].

3.1.2.2 Davies et al.

We learn from the paper by Davies et al. that data obfuscation has to be done per data or sensor type. Therefore we have to keep the system and our policy language extensible for new obfuscation mechanisms, as new data or sensor types are bound to occur. The examples listed for different granularity levels are kept in mind when designing an interface for the obfuscation mechanism.

The adaptability of the mediators in this solution fits the requirement for adaptability - <FR.4>. A useful idea for our design is that new mediators can be integrated easily through this design. The idea of allowing for an external development of those mediators is relatable to the crowdsourcing approach of context models the DS2OS takes.
We explore the adaptability aspects of this system in subsection 3.1.3 in greater detail, but for designing an obfuscating entity it can be said that for our design a loosely coupled obfuscator is considered useful.

3.1.3 Adaptability

This section tries to find good solutions for achieving adaptability to different use cases, since the DS2OS is designed to be usable in any domain, and to be adaptable to any pervasive computing environment. Therefore introducing a privacy solution that is specific for a certain domain, or environment setup contradicts the goal of this thesis. See the functional requirement for adaptability: <FR.4>.

3.1.3.1 Hong et al. - Bark

For a description of the work by Hong et al. see subsubsection 3.1.1.3.
The conditions of the Bark policy language can name external oracles for further logic evaluation. An aspect of the approach by Hong et al. that is considered useful is how the privacy functionality is designed in an extensible manner, allowing for new context to be added to the decision process. An adaptable system fulfills the functional requirement <FR.4>.

This also doesn't contradict our other goals, since this dynamic extensibility keeps the complexity (<NFR.1>) of the system at a manageable level. This is because the functionality of the external oracles doesn't have to be added to the core functionality of the system. It also helps in making the system understandable (<NFR.3>), since the inner working of the external oracles doesn't have to be understood by a programmer who just wants to use it.

3.1.3.2 Davies et al.

For a description of the work by Davies et al. see subsubsection 3.1.2.1.
Similar to the external oracles in the solution by Hong et al., Davies et al. propose a modular design where so called mediators are responsible for the access control and data obfuscation. Those mediators are interchangeable, the authors even consider a market niche for those mediators. This is a logical conclusion since data obfuscation heavily depends on the data format or the device type, therefore specialized mediators are needed for the different types.

In the proposal by Davies et al., the mediators can be created and shared by security experts, by trusted providers, or by an open source community. The latter is an approach that the DS2OS features, where the context models are shared by all service developers on a smartphone app store-like platform (see section 2.3). Therefore it is considered a useful approach for our implementation to also share the obfuscation rules and functionalities, and to have data obfuscation modules that can be integrated into the policy language.

3.1.3.3 ABAC & XACML

For a description of ABAC see subsubsection 3.1.1.1. For a description of XACML see subsubsection 3.1.1.5.
ABAC and XACML do not define specific relationships between entities but rather base the access decision on the attributes that an entity has. Therefore they can be

adaptable even in an established setup, since changes to the environment can be solved by assigning affected entities the new attributes and creating new policies for those attributes, instead of having to change the relationship descriptions between individual entities.

3.2 Policy languages

This section explores how a policy language has to be constructed in order to be expressive, but at the same time understandable, which is the purpose of research question **RQ.2**. The requirements that play a role in the creation of a good policy are the non-functional requirements *expressiveness* (<NFR.2>), and *understandability* (<NFR.3>), and the functional requirement *reuse* (<FR.5>). They are addressed individually in the following sections.

3.2.1 Expressiveness

In the non-functional requirement for expressiveness (<NFR.2>) we declare that the policy has to be able to express a fine-grained privacy understanding. This allows privacy experts and experienced programmers to implement their fine-grained privacy understanding.

3.2.1.1 XACML

For a description of XACML see subsubsection 3.1.1.5.
The attributes in XACML can be compared to the context information in the VSL which we can gather with the search providers. The PDP evaluates the attributes that are to be checked, which in our case is for example the location search provider looking up the proximity of the requester to the device. The PDP then evaluates if the attribute value fits the required value stated in the policy.

We do not have the need to declare such an extensive Targets with subjects, resources, and actions for our privacy policies. We can however adhere to the suggested parameters that define the conditions under which the policy applies.
The *resource* is the service *context* the policy applies to, like the humidity sensor of a weather station service. The *action* is either *get* and/ or *set*. The *subject* is the requesting entity, which relates back to the *group* based access control, only being more fine-grained here.
Since these parameters are not too diverse, we do not have to define extensive new

matching functionality. The above mentioned target identifiers are already of a certain data type, for example the requested context is the address of the context. The action can be get and/or set. The subject is a group for which the policy applies.

An example looks like this:

```
requestedContext=humidity,
action=get,
subject=guest
```

This is explored in greater detail in chapter 4.

The privacy policy evaluation is partly done by the Request Router, since it receives the request, therefore it acts as PDP.

The way the Effect of a rule is specified in XACML, e. g. Permit or Deny, is considered a useful approach for our solution. It allows the user to either say "Allow access from 9 to 11 on Fridays", or "Deny access from 9 to 11 on Fridays". If we want to use only one effect type, the latter case can be expressed using negation, so "Allow access if time is not 9 to 11 on Fridays", but it is less intuitive for the user.
Further, allowing the user to define the combining algorithm for the different rules achieves a high expressiveness. We discuss in chapter 4 how applicable it is to our use case.

3.2.1.2 Hong et al. - Bark

For a description of the work by Hong et al. see subsubsection 3.1.1.3.
The logical connection of the different contexts in the conditions ($\wedge, \vee$, and $\neg$) are a simple but powerful way of evaluating the different conditions. Since the users of the DS2OS are not normal users, but programmers, they have an understanding for the way those operators work. Therefore this is an easy and valuable policy expression logic. It can be also easily translated to and from natural language formulations.

3.2.1.3 Davies et al.

For a description of the work by Davies et al. see subsubsection 3.1.2.1.
Davies et al. don't specify a specific policy language but instead suggest to use profiles that contain policies, aimed at sharing and reusing them [10]. They identify *smart default privacy profiles* and *active privacy assistants* as possible approaches to define policies. This is too imprecise for our setup and therefore not useful for us.

The fine-grained obfuscation mechanism that Davies et al. integrate in their privacy setup is a useful approach. We aim at including this to achieve a finer granularity for our policy.

3.2.1.4 ABAC

For a description of ABAC see subsubsection 3.1.1.1.
The limits of ABAC are that the specific policy implementation is not defined, since the attributes are specific to every use-case. A PEP and a PDP have to installed, and policies have to be defined. The main concept of ABAC, assigning attributes to entities is not part of this thesis, since we have an attribute based ontology in the DS2OS, we want to find a policy that can express rules for those attributes.

3.2.2 Understandability

In order to help the user in protecting their privacy, we have to make the policies easily understandable. Otherwise their usage is requiring too much effort and discourages the user. Only if the users understands how they can pour their privacy understanding into a privacy policy will they feel private. Therefore the policy language has to be understandable for novices, and the possible configuration options shouldn't overwhelm then. See the non-functional requirement for understandability: <NFR.3>.

3.2.2.1 Hong et al. - Bark

For a description of the work by Hong et al. see subsubsection 3.1.1.3.
By using natural language to define the policy, Hong et al. made their policy language easily understandable and expressible for humans. This is also desirable for our setup, since a natural language means that the understandability and the ease of use is facilitated for the programmer (<NFR.3>).

Making rules activatable by the user, and allowing specifically designed rules for a certain type of device is also a good way to improve the understandability.
For once, it allows for the creation of default rules for specific types of devices or services. Specialists can create those default rules, or more experienced users can share their configuration with others.
Besides that, by providing a possibility to activate policies, currently unused rules do not have to be deleted. This way, rules can be defined and kept for possible future situations.

3.2.2.2 Davies et al.

For a description of the work by Davies et al. see subsubsection 3.1.2.1.
Davies et al. introduce an interchangeable privacy mediator, and propose that third parties develop useful mediators for user. Thereby they allow for default policies which can be designed by privacy experts for example. This highly improves the understandability (<NFR.3>) of the privacy policies, since the users can review the default policies and possibly adapt them to fit their individual privacy needs.
Davies et al. also propose smart default privacy profiles that represent the privacy understanding of the average user. This is a solution that would be nice to have, but is not further considered for this thesis.

3.2.2.3 XACML

For a description of XACML see subsubsection 3.1.1.5.
The policy definition in XACML is quite complex and takes some time to understand, therefore integrating XACML into the DS2OS is not considered as a solution. Having nested PolicySets with Policies and Rules makes it great for evaluation, but writing the policy rules by hand is a tiresome act. Therefore a Graphical User Interface (GUI) is probably needed to simplify the policy creation process. This complexity is not something we aim for with this first prototype.

3.3 Privacy vs. performance

This section explores how the complexity and performance of the DS2OS system can be maintained at the current level, how we can avoid to introduce unnecessary complexity, or avoidable performance loss. The research questions **RQ.3** and **RQ.4** express this. The requirements that play a role in the creation of a good policy is the non-functional requirement for *system complexity* (<NFR.1>), and the functional requirements for *reuse* (<FR.5>), and *autonomy* (<FR.3>). They are addressed throughout the following sections.

3.3.1 System complexity

One goal of this thesis is to keep the complexity of the whole system at the current level. Introducing complex new functionality means that the programmer has to take time to understand this new part as well, making the usage of the whole system less desirable. See the non-functional requirement for system complexity: <NFR.1>.

3.3.1.1 XACML

For a description of XACML see subsubsection 3.1.1.5.
Although we see the advantage of having a request/response language to have a clear separation between the processing entities, it makes the system too complex to introduce another communication language. For now, we work with the existing request messages, and any possible calls to evaluate entities are done using existing entities, like the search providers.

3.3.1.2 Davies et al.

For a description of the work by Davies et al. see subsubsection 3.1.2.1.
The complexity of the system introduced by Davies et al. is kept low by making the mediators an entity which can be shared and thereby changed to fit a solution by someone else. Through this interchangeability, the mediators can be described precisely in their functionality. The interface for the mediator can be reused to fit in new operational units.

3.3.2 Reuse

Reusing existing functionality of a system, in our case the DS2OS, is declared as a functional requirement for our design (<FR.5>). The last sections show that we already have a good basis for a privacy policy, since the VSL has an ontology to semantically describe the environment with its entities. Therefore we can for example use the search providers to evaluate the attributes of the privacy policy. Chapter 4 discusses the aspects of creating a policy with low complexity and reuse of existing, familiar functionalities.

3.3.3 Autonomy

The KAs in the VSL are decentralized nodes in the self-managing P2P network. They manage the storage of the data produced by a service, and provide context discovery by caching the node structure of the VSL locally. This makes the nodes autonomous and resilient against network outages. The self-management of the DS2OS also proves to be a great advantage for the user, as it makes the introduction of new services easy for the programmer. See the functional requirement for autonomy: <FR.3>

3.3.3.1 Hong et al. - Bark

For a description of the work by Hong et al. see subsubsection 3.1.1.3.
The great advantage of the DS2OS is that it is a distributed, self-managing network of nodes that can operate without a central management unit. Therefore introducing a central policy storage counteracts the autonomy requirement (<FR.3>). The approach taken by Hong et al., to cache the policies locally, is a valuable solution to maintain the distributed, self-reliant nature of the VSL.
The solution of Hong et al. of storing policies centrally and caching them locally can be applied to the VSL by caching the central policies on the KAs. However, the alternative of storing the policies directly on the KAs is preferred for our solution, as chapter 4 discusses.
It is not required for our system to have default-off communication since the entities in the DS2OS are trusted, there are no communication links to third party entities.
Where to store the policies and how we can increase autonomy of the KAs by caching for example is discussed in more detail in chapter 4.

The proposed two-factor authentication using a user-in-the-loop is not considered useful for our setup, since it prolongs the response time. Involving a user also makes the KAs less autonomous, since they have to maintain a communication with the owner of a service for example.

The downside of the whitelisting approach this paper takes is that the user has to explicitly declare all allowed communication beforehand, having to foresee all eventualities that might be required. This makes the system prone to access errors, conflicting with our requirement of autonomy <FR.3>, and the performance **RQ.4**.
However it is a good compromise to have the same approach if policies are defined, so if there are privacy policies associated with a service, we evaluate them and if at least one of them evaluates to true, then access is granted, otherwise access is denied. If no policies are installed, then the access is allowed.

We don't have to consider the handling of the general communication, like the default-off strategy of this paper, since communication handling is done by the DS2OS. It guarantees good security by encrypting the traffic on link layer, and validating the identity of the entities in the network by issuing certificates (see section 2.3). Therefore it is advisable that we define policies which restrict the access, instead of going the whitelisting approach this paper proposes.

3.3.3.2 Davies et al.

For a description of the work by Davies et al. see subsubsection 3.1.2.1.
The modular design Davies et al. propose is a decentralized approach where the cloudlets can be installed on different computing entities, since they are portable. The cloudlets contain the privacy policies and sit in between the sensor and the next processing service.
The high flexibility and the autonomy of the cloudlets are useful references for our design, since they resemble the operation of the KAs in the VSL. We therefore make an effort to resemble the cloudlets in their autonomy, and try to keep the privacy policies as decentralized and independent of other entities as possible.
The takeaway from this solution is to allow the KAs to evaluate the privacy policies independently from the rest of the network, in order to make them resilient against network problems.

3.3.3.3 ABAC & XACML

For a description of ABAC see subsubsection 3.1.1.1. For a description of XACML see subsubsection 3.1.1.5.
The ABAC and XACML policy languages have great similarities to the multi-factored context decision we envision for our privacy solution. Since ABAC and XACML require instances to evaluate and enforce these policies, we take into consideration which entities can act as PDP and PEP.
In the DS2OS the access control is currently checked on each KA. In subsection 2.3.7 we explain that the requesting entity has a certificate signed by the local site, which also contains the groups / accessIDs the user is assigned to. The context model of the service declares the access rights needed to perform get and set operations for the individual contexts. Since the KA has all the information it needs to make a privacy decision for this access control, it is autonomous and can work in a decentralized manner. We aim at maintaining this decentralized operation and therefore extend the existing autonomy of the KA by having a node-local PDP and PEP for our privacy policy evaluation.

3.4 Summary

This chapter explores some useful approaches to the multi-faceted problem of introducing a privacy policy in the DS2OS. We sort those findings into the research questions formulated in section 1.2. Analyzing the different aspects of the literature along the

requirements that we associate with those research questions is a good structure to compare different technologies for their approaches, their usability.

Since *reuse* (<FR.5>) is a requirement we define for our implementation and is specific to the DS2OS, it is not something we can trace in related literature, therefore it is excluded from the comparison.
Table 3.1 summarizes what literature is useful to us for which aspects.

Notation information:
The symbol ◯ expresses that a property is not described in enough detail in the literature to take any conclusions from it.
The symbol ⊕ expresses that advantages and disadvantages came with the proposed solution.

Bark, ABAC, and XACML answer our **RQ.1** to a great extend, they show us how an access policy can be based on different contexts, mainly environmental attributes. These reference policies are a basis for the policy language we design in the next chapter. Especially the high adaptability that comes with these solutions is considered. Multiple examples for integrating diverse contexts into a policy language are shown.

RQ.2 is not as clearly answered by the literature. XACML is highly expressive, it is the language that comes closest to the expressiveness we envision for our setup. However it is far too complex and requires a GUI if understandability is also a goal.

Requirements	Hong [20]	Davies [10]	ABAC	XACML	DS2OS (plan)
RQ.1					
Multi-factor	✓	◯	✓	✓	✓
Adaptability	✓	✓	✓	✓	✓
RQ.2					
Obfuscation	✗	✓	✗	✗	✓
Expressiveness	✓	◯	✓	✓	✓
Understandability	✓	✓	✓	⊕	✓
RQ.3 & **RQ.4**					
System Complexity	✓	✓	◯	✗	✓
Autonomy	✓	✓	✓	✓	✓
Decentralized PEP	✓	✓	◯	◯	✓
Decentralized PDP	✗	✓	◯	◯	✓
Focus	service	data	data	data	data
Policy language	Bark	◯	ABAC	XACML	tbd
Policy evaluation	Bark	Mediators	PEP	PEP	tbd

Table 3.1: Related work and the aspects covered in them, compared to the goal state of this thesis (see on the right).

Previous work on data obfuscation, especially in the context of smart environments is sparse. The work by Davies et al. proposes the integration of an obfuscator into the access decision process, and describes multiple ways context can be obfuscated. It is quite vague on the way the obfuscator is instructed, the way the it is integrated into the system however is a useful example for our design.

In regards to **RQ.4**, the question for autonomy is answered by all related works, mainly through dynamically plugging in policy evaluation entities. The **RQ.3** is naturally not as clearly answerable, since it also depends on the system. However, by describing the system setup clearly and by keeping the entities extensible, Hong et al. and Davies et al. present useful approaches for our design.
Even though ABAC and XACML emphasise the benefits of a design separation of PEPs and PDPs, it is not discussed in further detail how they process the policies. The most explicit description of the policy evaluation process is done by Hong et al.

The DS2OS is a data-centric system, therefore it is interesting to compare other systems for this property. Except for Hong et al., all designs are also data-centric.

It becomes clear that none of the related work covers all of our requirements, but individually contribute useful ideas to our design. In chapter 4, we use those findings to discuss the best approach to design a privacy policy that is more fine-grained than the current access control of the DS2OS.

Chapter 4

Design

The design of the policies weaves in the findings from the previously discussed literature, especially the ones addressed in chapter 3. We adhere to our functional and non-functional requirements defined in subsection 2.4.1 and subsection 2.4.2, and try to answer our research questions declared in section 1.2.

The first requirement is to design a privacy policy, that is able to express access requirements at a more fine-grained level than the existing access control (<FR.1>). We look at how we can reuse (<FR.5>) the existing architecture and functionalities to evaluate those additional contexts.
Including more contexts to decide the access control is a useful approach, but already done by many systems. A new approach to access control and privacy policies is to also obfuscate (<FR.2>) the returned data along the privacy understanding of the user. How the obfuscation can be defined and done is discussed in this chapter.
We keep in mind not to create a domain-specific implementation of the policy, but to instead keep the system adaptable (<FR.4>) to various use cases.
Those topics are addressed in the first section, section 4.1.

In order to reuse (<FR.5>) the existing architecture and the functionalities of the DS2OS, we look at how the policy can be integrated well. We make an effort to keep the VSL as autonomous (<FR.3>) as possible, so that the communication is kept as reliant and safe from outages as possible.
We keep in mind that the system should not become too complex (<NFR.1>), by ensuring that the programmer does not have to understand a whole new access logic, or learn how to define the privacy policy.
This will be discussed in section 4.2.

Expressiveness (<NFR.2>) aims at a policy language that is fine-grained but also understandable (<NFR.3>) to non-experts. This also keeps the complexity (<NFR.1>) of the system low. These usability aspects is discussed in section 4.3.

Along those topics, we design a privacy policy step by step that fits our needs and is adapted to the DS2OS.

4.1 Privacy policy

4.1.1 Fine-grained policy

As we elaborate in the previous chapters, the more fine-grained the policy is, the better it is able to represent the privacy of the individual. Therefore we looked at possible contexts that should be expressible in a privacy policy, and mainly identified location and time as essential privacy contexts.

4.1.1.1 Location and other context based policy

In analyzing the Bark privacy policy (sec. 3.1.1.3) we notice that in the DS2OS we already know a lot of the parameters that belong to a request. We can match the Bark *subject* to the requester of a service is, since the request sent to the responsible KA includes the ID of the requesting entity. We also know the *action* that is to be performed, since it is delivered in the access request to the service, and can be either *get*, or *set*. The *object* is the KA that is running the requested service (and also the one who is processing this access request). The *object* is the service itself. The *conditions* currently are only the group based access restrictions.
What we cannot match are the location of the requester (*subject*) and the location of the requested service (*object*). This shows us that we mostly have all the relevant information needed for constructing a privacy policy, and we can also add the possibility to have more conditions for the access request.

How the DS2OS works is explained in section 2.3. We mention how search providers can do semantic discovery of context. If new context is defined, it can become discoverable by adding new search providers. Therefore we adhere to our functional requirement of reuse <FR.5> and use this functionality to add location discovery through a location search provider. We do not develop a specific location search provider (adhering to our requirement for adaptability - <FR.4>), but rather pass location queries to the search provider interface. The location search provider can be registered at the address */search/location*, therefore a request to it looks like this:

```
get /search/location/livingRoom
```

The result of this request consists of all the nodes that are located in the living room. A very basic location based policy therefore looks similar to this:

```
location="livingRoom"
```

If a type of device is searched at a specific location, we use the multi-parameter search mentioned in subsection 2.3.8:

```
get /search/typeAtLocation/livingRoom/!/bathRoom/?/basic/number/!/basic/text
```

The policy then looks something like:

```
(location="livingRoom" or location="bathRoom") and (type="/basic/number" or type=
    "/basic/text")
```

This is easily translatable into the corresponding query to the search provider.

Following those examples we see that the VSL is very adaptable, it becomes possible to have search specific to a needed use case, so in the example above to have a search provider for finding services that are of a specific type at a given location. Therefore our policy can also be specific to those use cases, and thereby to different contexts. If we want to evaluate an access request for the context *owner*, then we can define a search provider that searches for the owner of a certain device.

4.1.1.2 Time based policy

For a time based policy, we do not need a search provider, since this can be evaluated based on the current time.

XACML has a number of time representation types, and functions on those types, as we show in section 3.1.1.6. We use the data types *time, date, dateTime, dayTimeDuration*, and *yearMonthDuration*, though only in their type usage, not in the exact format used by XACML. The format of XACML includes an expression for time zones. Since the policies are evaluated in the VSL, we do not have to consider time zones for our design, as the local time of the machines can be used for the policies. So `date` in our design is of the following format `yyyy-mm-dd`, `time` is defined as `hh:mm:ss`, and `dateTime` has the format `yyyy-mm-dd hh:mm:ss`. `DayTimeDuration` is expressible in the format `PdDThhHmmMssS`, and `yearMonthDuration` is `PyYmmM`.
We expect that `date`, `dateTime`, and `yearMonthDuration` will not be used very often in our context, since it is unlikely that a user defines a specific date in the policy. It is

more likely that recurring patterns such as policies for a day of the week, or a specific day of the month is defined, instead of year-long safety measures. Therefore we also define `dayOfTheWeek`, which is not part of XACML, but which can be useful to express recurring patterns such as `Allow access on Fridays from 9 to 10`. It is included in our design as a string, e. g. `Mon, Wed`.

The functions that are deemed useful for our policies are `X-equal, X-greater-than, X-greater-than-or-equal, X-less-than, X-less-than-or-equal`, and `X-intersection`, where X is one of the above mentioned types. Those functions are applied in XACML directly to time declarations in the policies. Since we want to have an easily understandable policy (<NFR.3>), we do not use those functions directly in the policy, but rather have a natural-language-like policy where statements such as `time="after 10:00:00"` or `time="between 10:00:00 and 16:00:00"` can be expressed. We then evaluate those expressions with `time-greater-than`, or `time-intersection` respectively.

4.1.1.3 Obfuscation of data

A novel attempt to enhance the privacy protection is to also add pre-processing to the access control, as we show in subsection 2.2.3. An example for this is a smart air conditioning system that can display the outside and inside temperature readings, as well as returning stored measurements from the past. We can think of a privacy understanding where the device owner and all house inhabitants can access the complete history. In this scenario, a technician is only allowed to access the daily average over the data from the past year. All other people are only allowed to read the minimum and maximum readings of the past week, and only if they are within a 10 meter distance.

Another example is a security camera, where the faces of the house members are blurred, and any service working on the camera feed can only access a single frame every 10 seconds.

It does not make sense to implement the functionality to transform the data, so to define the data obfuscation functionality, as this heavily depends on the data type. Therefore we define only the interface to an obfuscation mechanism. Any obfuscation mechanism has to be provided to the system for the specific data type that is to be obfuscated. Many data types can be handled with one obfuscation function, as for example any sensor measurements can be analyzed through statistical measures, such as average, min-max, or distribution. We pass to the obfuscation mechanism the parameters *granularity* and the *time span* with which we aim at modulating the data. Those two parameters are the parameters along which Davies et al. suggested to obfuscate the data. Since we

obtain the data type through the context model, we do not have to explicitly state the data type in the policy.
The obfuscation policy statement is of the following format: `granularity='frames', timespan='1 week'`.

The complete privacy policies for the first example, where the temperature measurements are obfuscated, looks like this:

```
group="technician"
obfuscation="granularity='average of day', timespan='1 year'"
```

Listing 4.1: Technician is allowed to access the temperature readings as daily averages for the past year.

```
group="inhabitants, deviceOwner"
obfuscation=""
```

Listing 4.2: House inhabitants and the device owner can access the data completely.

```
location="distance='10 m'
obfuscation="granularity='minDay, maximum of day', timespan='1 week'"
```

Listing 4.3: Temperature reading for all other entities that are in a 10 meter distance can access the minimum and maximum values of the past week.

For the security camera the following policies are declared:

```
group="*"
obfuscation="granularity='blurredFaces of inhabitants'"
```

Listing 4.4: The faces of the inhabitants are blurred for everyone.

```
group="service"
obfuscation="granularity='frame' frequency='10 seconds'"
```

Listing 4.5: All services get access to single frames every 10 seconds.

Obfuscation only makes sense for *get* operations, as sensitive data is returned in a minimized extensiveness. Writing data needs to be done to the full extent, since the sent data has to be stored once in its full capacity. The submitter can also modify the data that is to be written before sending it, therefore obfuscating the data at the receiving end doesn't make sense.
However we cannot simply say that all policies therefore only apply to read operations, since write access might also be privacy sensitive, as for example switching on a camera is. Therefore an additional attribute is added to our privacy policy, which is `action`, which can either be `get` or `set`.
If obfuscation is declared for a `set` policy, the programmer might be warned that no obfuscation can be done with this kind of action.

The obfuscation is performed by an obfuscation service which has to be deployed and registered in the VSL. As anchoring point, `/system/obfuscator` is suggested and used for the prototype. In the future, it might be useful to have the address `/obfuscator/X`, e. g. `/obfuscator/statistical`, just as the search providers are addressable as `/search/X`, e. g. `/search/type`.
The raw data that is to be obfuscated is sent to the service, along with the parameters that are passed in the policy. The obfuscating service then processes the data and returns it to the KA.

What we can see when we look at the policies defined above is that it is not too clear how they are expressed as a coherent policy set, and how the evaluation manages priorities. Is the first matching rule applied? Is access allowed when the rules match, or is it denied? Do all policies have to be matched, or is it sufficient if one of them allows access? What happens when no rule matches?
These and more questions are explored and answered in the following sections.

4.1.2 Policy Language

We observe in the literature that a natural language is useful for intuitive policy formulation. Allowing the users to formulate the policies however they wish to requires too much of natural language processing though, therefore we introduce some keywords that sort the privacy parameters into logical subunits, like *time* for example. This is also the approach we take in the previous section.

The advantage of formulating the policy along predefined keywords also has the advantage of later mapping those keywords to natural language constructs, as it is done by Hong et al. [20]. They map subject, objects, and actions from the data types, like the *subject - who*.
In subsection 2.3.4 we also explain how the VSL has an ontology that has *subjects, objects*, and *predicates*, and can express *has-a* and *is-a* relationships. Therefore we can also use keywords and map them to the *subjects, objects*, and *predicates* of the VSL ontology. In chapter 3 we show that we can map almost all of our existing ontology elements to a policy language as proposed by Hong et al. By using the *subject - predicate - object* structure, users can understand the policy well [20].

Another solution to making the language intuitive for the user is to also use a GUI, where the keywords can be woven into a policy definition interface that is even more understandable, as it is done by Dixon et al. for example [13]. The keywords are then mapped to an intuitive, natural language representation of the policy.

In the current context models, the access control is also done via the keywords *writers* and *readers*. Therefore the programmer is already used to this structure (<NFR.1>).

An alternative is to use the XML-based policy language of XACML, but it entails integrating the whole architecture of XACML. Additionally, the language is far too complex for our setup, as we explain in chapter 3. Furthermore, it doesn't include the necessary structure to offer obfuscation. Therefore we only pick useful concepts and functionalities from XACML.

4.1.2.1 Keywords

The search provider functionality is a major benefit for semantic discovery, as we explain in the previous sections. We can only do semantic discovery for what semantic representation is present in the system. Therefore we define our keywords along the types of the search providers. We already have the *type* search in the current DS2OS implementation. As we explored in subsection 2.3.8, we can extend the search functionality by adding new search providers to the VSL. By defining keywords as the main policy constructs we also make clear what search providers are needed.

So, with the exception to *group*- and *time*-based policies, we can define new search providers for adding new policy evaluation possibilities. For example for *location*-based policies, we can define a search provider that can handle queries that search for the location of an entity, or answer to a query for the distance between an entity and another.

Making the search providers the main entity to our privacy policies keeps the system adaptable (<FR.4>), reuses existing technology (<FR.5>), thereby also keeps the system complexity (<NFR.1>) at the current level, as we outsource a lot of the information gathering process to the search providers. We therefore start with the keywords *type, group, location*, and *time* for our prototype.

How the adaptation to the search providers works is answered in subsection 4.2.2.

4.1.2.2 Logical connection

We decide on using keywords, but how do we combine them into a coherent logic? For singular expressions, it is quite simple, to just say `time="between 9 and 10"` for example, but when more contexts are added to the policy, like `time="between 9 and 10", location="livingRoom"`, it has to be declared whether both contexts have to evaluate to `true`, or if it is sufficient when one of them is fulfilled.

For this, we can use logical operators such as ¬, ∧, and ∨. Hong et al. [20] use logical operators to connect conditions, and rules, and in XACML, the different rules are also connected via logical functions (see [36]).
An example policy can then look something like `time="between 9 and 10" OR location="livingRoom"`.
We can also think of scenarios where it becomes necessary to group certain logical conditions together, and to evaluate the group against another expression. In XACML this is solved by nesting conditions into each other. We can solve this by simply introducing brackets to our policy.
Therefore our policy will look something like `time="between 9 and 10" OR (location="livingRoom" OR location="kitchen")`.

What we have to figure out now is whether the policy outcome should be seen as a permission to perform an action, or whether those conditions are describing a case where access should not be granted. This is explored in the following section.

4.1.3 Policy effect

In XACML, policy rules can declare one of two kinds of effects, *permit*, or *deny*, which we explain in chapter 3. The effect is associated with each rule, therefore either one of the effects is returned on evaluation [36]. In case of an error, either *indeterminate* or *not applicable* is returned.

Hong et al. used a whitelisting approach in their work, meaning that only actions that are explicitly allowed can be performed [20]. Therefore the effect of all of their policies is a *permit*.

In general it is known that whitelisting is more secure, but also requires more management of the rules, since all possible scenarios which might need access rights have to be foreseen and declared in the policy. This makes a new service not usable if it has context that doesn't apply to the existing rules. This then leads to a scenario that it is too restrictive in regards to allowing access to data, thereby reducing the functionality of the system, which contradicts our **RQ.4**. The nature of IoT environments is the high connectivity and the operation on data, therefore it reduces the performance of the system.

Blacklisting is less secure, as it can only prohibit context which is already known, so new threats might get past the policy. It is more convenient for the user, as by default all communication is allowed except for the entries in the blacklist.

In the DS2OS, access is currently granted when an entity is a member of a certain group. Since the most basic context models declare that everyone has access, and all other context models are derived from those basic models, the default is to have no access restrictions. If access restriction is required, it can be set for each individual node in the *reader* and *writer* ID fields of the context model. Therefore by default all entities are allowed to access everything, but once a specific accessID is allowed access, no other entities are allowed, therefore it is whitelisting. We use a whitelisting approach for this reason, in order to be consistent with the logic of the system (<NFR.1>, <NFR.3>). Any privacy policy rule that is declared therefore is a whitelist of access. If no privacy policy is named, then no privacy has to be enforced and access is granted to the full data set to whoever has passed the basic group-based access check.

To name an example, `location="livingRoom"` means that everyone who is in the living room is allowed access. In the future the *effect* property can be introduced , through which blacklisting is enabled.
Blacklisting can improve the understandability of a rule, since statements like *From 8 to 10, nobody located in the living room should be able to perform the action* can be expressed with an blacklisting approach as `effect=deny, policy="time='from 8 to 10' and location='livingRoom'"`.
With whitelisting, it looks like this: `effect=permit, policy="not (time='from 8 to 10' and location='livingRoom'"`.
The latter is less intuitive for the user and therefore can easily lead to unintended behavior. We make sure that a later introduction of policy effects can be easily done for our solution.

4.1.4 Policy combining algorithm

In subsubsection 4.1.1.3 we use multiple policy rules to define a desired behavior. The question that remains is how those different policies are combined to return a final evaluation. Taking the presented use case as an example, where the inhabitants, and the device owner are allowed full access, a technician can read the daily average of the past year, and everyone else gets access to daily minimum and maximum values of the past week.
Are the policies evaluated one after another, and the first match is taken? And what if multiple policies are applicable, which one is applied?

Hong et al. use a logical $\vee$ to combine their different policies. Therefore if at least one of the rules returns true, then access is granted. It is a useful approach to achieve a

good performance, since we can stop evaluating all other policies once one of the rules returns an access grant, returning a result more quickly.

Directly applying this to our scenario is not possible though, since we want to use obfuscation mechanisms. We can think of the following scenario with two policies in place:

```
group="family"
obfuscation="granularity='average of day', timespan='1 week'"
```

```
group="owner"
obfuscation="granularity='average of day', timespan='1 year'"
```

If the owner is part of the group *family*, only the data of the past week is available, since this was the first rule that matched. We have a look at other combination algorithms to evaluate if there is a better way to solve this.

We show in subsubsection 3.1.1.5 how XACML uses different combining algorithms. The one we address in the previous paragraph is called *first applicable*.
The *deny overrides* algorithm returns deny if one evaluation returns deny. This is the same in vice-versa for the *permit overrides* algorithm.
The *ordered deny overrides* and *ordered permit overrides* algorithms works in the same manner, but also takes the order in which the policies are defined into consideration.
The *only one applicable* algorithm is not usable for combining rules, but for policy sets and policies. Only one of the policies can return a permit or deny result, all other policies have to return the *not applicable* effect [3].

However refined the policy combination algorithms are, they introduce too much complexity to our system, since we do not want to include policy effects yet (as stated in subsection 4.1.3). Also the user has to get to know all the different policy combining algorithms before being able to define a policy, making the policy formulation process less understandable <NFR.3>, contradicting **RQ.2**. We therefore resort to the first example, where we consider applying a *first applicable* policy.
In order to avoid the scenario described above, the programmer has to declare the rule with the least obfuscation first. Through this, the functionality of the system is also kept high, as the least obfuscated data set can enable the best extend of operations on it.

4.1.5 Rule matching

We do not check how well a policy matches, since this requires a Target declaration as it is done in XACML. This is too much complexity that is introduced to our system

(<NFR.1>). What we do instead is to differentiate between the `action` type of the request, namely if it is a `get` or a `set` request. Those actions represent different data accesses altogether. A user might be willing to let others activate the camera (send a `set camera on` command), but does not allow read access to the camera (send a `get cameraFeed`).
In order to increase the usability of the policies, and to avoid duplicates, a policy can be defined for both actions. An example is that access to the temperature service is only allowed from inside the house, it doesn't matter if read or write access is requested.

We also differentiate between the contexts that are requested, so to say the address of the context for which the policy applies. The services of the DS2OS can be composed of multiple other contexts, as we explain in subsection 2.3.4. Since the different contexts are of possibly different data types, we have to make a context specific policy declaration possible. An example is to have a service that is composed of a temperature and a humidity measuring service. Therefore one policy addresses the temperature service, `requestedContext=temperature`, another policy handles all requests to the humidity service: `requestedContext=humidity`.

We therefore evaluate all rules that match the request parameters, until either access is granted, or no rule returns `true`. In case none returns true, we adhere to the prevailing whitelisting approach, therefore no policy allows access and the requester is denied access. We make one exception to this behavior, because if no rule is defined, then we grant access. This resembles the functionality of the context model based access policies.

4.2 System integration of privacy policy

This section looks at how we can adhere to our research questions that want to keep the system at the current complexity (**RQ.3**), and to maintain the performance of the system (**RQ.4**). Therefore we look at what is the best place to declare and store our policies in (subsection 4.2.1).
We already consider how the policies are logically evaluated, now we focus on how the policy evaluation can be done considering the system setup, so which entities are responsible for the evaluation, and what other factors play into its functionality (subsection 4.2.2).

4.2.1 Storage point

The decision where to declare the policies has multiple possible locations to choose from.

The first possibility is to store the policies in the context models of every service. This also suggests itself since the current access control is also done in the context models. Since the context models are stored at the CMR this has the advantage that all KAs in the network are able to query the policy of the service before sending the actual request. This allows for fewer traffic and less latency. This approach is not chosen for multiple reasons.
One reason is that the policy is likely to be different for every service and therefore would result in a multitude of context models. The benefit of the DS2OS context models is that they are abstract service interfaces which can inherit properties from other context models, or meta models. By creating a specific context model for every service, this abstraction is counteracted (<NFR.1>, <FR.4>).
Another reason is that the XML tags of the context model gets bloated if they contained the accessIDs plus an arbitrarily long privacy policy. This reduces the understandability (<NFR.3>).

The next possibility is to store the policy at service initiation time in the corresponding database of the KA. This approach has the advantage of keeping the policy declaration point (like possibly the context model) abstract and general. It has the disadvantage that the changes over time of the policy are not traceable. Also changes to the policy then require a convenient database access interface, like a GUI. Therefore it is not chosen for our design.

Another storage point is a policy database centrally in the VSL. This idea is not pursued as it doesn't bring any advantages besides having an easy implementation approach. The disadvantages are in greater number, as this introduces a single point of failure, and defeats the reliability and autonomy (<FR.3>) that is achieved through distributed nodes.

The fourth possibility is to store the policy in an additional meta data field of the services. The meta data is part of the service manifest, which in turn is part of the service package. The service package further contains the service executable, and the service certificate, all of which we explained in the subsection 2.3.6.
This possibility is chosen for our design, since the service manifest is downloaded from the DS2OS, and therefore default policies can be integrated in it conveniently. It also avoids the multitude of specific instances, which the first possibility features. The service

manifest is adapted to the specific VSL context during the installation already, therefore it is convenient for the user to also adapt the policy at this point. The policies are stored locally at the KA, instead of at a central storage point, as the third possibility suggests. Compared to the first possibility, the understandability is achievable in a better way since the JavaScript Object Notation (JSON) format can have objects as values, which allows us to declare sub-values. In the XML-based context models, this is only achievable by defining a new XML tag.

A hash of the manifest is stored in the service certificate and is therefore verifiable. Since this implementation is a first prototype, the policy is not part of the signature. Therefore it is not verified, but the rest of the meta data, and general service information such as the certificate remain the same and are therefore verifiable. In the future, the policy can become part of the verifiable hash. Until the policy becomes part of the service certificate, it is declared as a `xPolicy` element, the `x` being a marker for the unverified part of the service package.

By storing the policies in the service manifest we can declare the policies per specific service instance, without making the context models of the services too individual. The existing structure of the service manifest also allows us to define the policies more intuitively, since it is a JSON file. The service manifest that is presented by Donini represents additional meta data of a service, and currently looks like this:

```
{
  "serviceId":"",
  "developerId":"",
  "versionNumber":"",
  "dependencies": [],
  "resourceRequirements":{
        "CPU": {},
        "RAM": {}
  },
  "requiredContextModels":[],
  "conflictingContextModels":[],
  "executableHash":""
}
```

Listing 4.6: Manifest file as seen in [14]

Following our previous discussions and design decisions, we define a solution where the manifest includes a way to define how the data is obfuscated, and the policy itself. We decide on having a new attribute-value pair for each policy. It can have individually set names, those are not important for the policy evaluation, but can help the user in understanding what the policy is doing. Included in those policies are the attributes we declare so far, namely `action`, `requestedContext`, `obfuscation`, and the `policy` itself. Further discussions follow in the next sections, but for now we envision a manifest file that looks similar to this:

```
{
  "serviceId":"",
  "developerId":"",
  "versionNumber":"",
  "dependencies": [],
  "resourceRequirements":{
        "CPU": {},
        "RAM": {}
  },
  "requiredContextModels":[],
  "conflictingContextModels":[],
  "executableHash":"",
  "xPolicies":{
        "policy1":{
                "action":"get",
                "requestedContext":"temperature",
                "policy":"group='family'",
                "obfuscation":""
        },
        "policy2":{
                "action":"get",
                "requestedContext":"temperature",
                "policy":"group='guest' and location='house'",
                "obfuscation":"granularity='daily average', timespan='1 week'"
        }
  }
}
```

Here we declare the least obfuscated rule first, in order to adhere to our previous conclusion to allow access on the first access grant.
New parameters can easily be added afterwards, such as the effect of the policy (permit, deny), or a policy combining algorithm if needed.

The service package is downloaded from the S2Store to the SLSM, which then redistributes the service package to a chosen node, its NLSM. On downloading it from the S2Store, the programmers can declare their individual access rights for the service. Therefore the policy is available locally, it is not stored on a central entity, making the policy evaluation resistant to network problems, thereby autonomous <FR.3>. Also latency is reduced since the policies do not have to be queried on a remote location, but can be evaluated directly on the KA. Even if we counteract this problem by caching policies locally, we run into the problem of consistency. Therefore the advantages of this approach are clearly in greater number.

This way we can also define default policies for a service. The default policy is declared by the developer and stored in the service manifest. This manifest is then uploaded to the S2Store. When the user installs the service into the local VSL, the default policy is included in the service manifest, but by default is not active. The user then adapts the service manifest, and thereby the policy set, to the parameters of the VSL. The user can activate the default policy, or can design a new one with the default policy

as a guideline. This way, a default policy can help with understanding <NFR.3> and defining an individual policy. We discuss default policies in subsection 4.3.1.

4.2.2 Policy evaluation

In the previous sections we decide on what the policy language looks like, where the policy is stored, and how they are logically evaluated. What we want to focus on in this section is how the additional data needed can be obtained and made available, how we can handle different service policies, and how latency can be avoided.

4.2.2.1 Querying oracles

We explained how search providers are the functionality we base our policies on. The keywords we use are the search providers we address for the individual policy aspects. The disadvantage of this approach is that we rely on the availability of the search providers. If for example a location search provider is not available, no location based policies can be evaluated. The alternative is to implement all possible evaluation functions for privacy policies inside the policy evaluation functionality. This strongly contradicts our requirement for keeping the system adaptable (<FR.4>), it does not reuse existing functionalities (<FR.5>), and increases the system's complexity (<NFR.1>). Even if implemented, it is not able to solve all problems, since location information might still be represented differently in other domains. Therefore the best approach that remains is to use the search providers.

When a search providers is present in the system, but becomes unavailable due to some network or agent failure, the policy can no longer be evaluated. Therefore the usage of search providers for the policy evaluation means a decreased autonomy of the KAs, which is a requirement of this thesis (<FR.3>).
In this case we can employ caching of previous search provider results to make the KAs more autonomous. Then, if a search provider is called but doesn't respond, it is checked whether the request has already been performed and if the response is still cached. If this is the case, then that cached evaluation is taken as a valid response, therefore cached search query results have unlimited validity in the cache. Once the search provider becomes available, any new requests of course uses the up-to-date response of the search provider.
However if no query results are in the cache, then access is denied, since it is a security and privacy threat to grant access once a policy cannot be evaluated. This leads to unavailable and not functional services, but it is at a balanced cost through caching (<PR.6>).

4.2.2.2 Latency

As we explain before, we have to query for additional information to evaluate our policy. An example for this is to query if a certain device is in the living room. For this, the location search provider is called, the relevant context is returned and then the policy can be further evaluated.
A normal access request without a search provider involved is evaluated only for the group based conditions. The evaluation latency we introduce by using search providers compared to a basic, group based access check is tested in chapter 6. We also evaluate how well caching improves the response time.

Latency can also be introduced by having a high number of policies, which all have to be evaluated until one of them grants access. If none does, the whole set is evaluated. Since the evaluation may include querying search providers, this potentially is a big latency that is introduced.
To define a high number of policies is an effort for the programmer, and also decreases the understandability. Therefore the number of policies per service is likely to remain low.
In order to make the removal of policies easier, we consider introducing the possibility to activate and deactivate policies. Since this is also a useful property for default policies, this concept is explored in section 4.3.

The complexity of the policies can also be a factor for latency, since many nested conditions require a recursive evaluation of the policy.

Another latency that is introduced is the obfuscation of the data, since this involves sending the data to the obfuscating service, the obfuscator processes the data according to the parameters given, and then returns the obfuscated data to the KA, which can then finally send it to the requester. In order to avoid as many calls as possible to this service, we only send out the obfuscation request once all other parameters of the policy are matched, so if it is certain that the requester is allowed to access this data.
Latency can be further reduced by having the obfuscation service on the same KA that is processing the request.

All mentioned latency factors are taken into consideration in chapter 6.

4.2.2.3 Policy conflicts

An advantage of our setup with a decentralized policy storage is that all policies are in one single place, therefore an overlook over all policies for one service is possible. This makes the maintenance of the service functionality more feasible, because too broad

or too restrictive policies can be identified more easily than if they are synchronized between different policy storage points.

We also avoid having policy conflicts by defining them in one place, since we only evaluate the policy set in the service manifest of the service. We decide on evaluating the policies using the *first applicable* combining algorithm in subsection 4.1.4. This also avoids policy conflicts, as possible conflicts do not have to be evaluated, but instead the first rule that allows access makes the evaluation process stop. If no rule is applicable then access is denied.

A representative policy that implements all the previously discussed aspects looks like this:

```
{
  [...]
  "xPolicies":{
    "policy1":{
        "requestedContext":"temperature",
                "action":"get",
        "policy":"group='family'",
        "obfuscation":""
    },
    "policy2":{
        "requestedContext":"humidity",
                "action":"get",
        "policy":"group='guest' and location='house'",
        "obfuscation":"granularity='daily average', timespan='1 week'"
    }
  }
}
```

4.2.2.4 Policy inheritance

Another consideration in the designing process is whether policies should be passed down from the parent node to the child nodes. In this scenario, the children is obliged to adhere to the parent's policy, and that the parent's policies overwrites the child node's policies. This is more restrictive for the child nodes and leads to an access granting behavior that is not traceable.
Another possibility is that the child nodes are able to overwrite their parent's policy. This results in the same confusing behavior.

The alternative to these scenarios is that every node implements their own policy and nothing is passed on to child nodes. This brings the disadvantage that the policy has to be defined for every single node, which causes repetitiveness and reduces the ease of programmability. This effect can be diminished by introducing default policies (see subsection 4.3.1).

The current access control in the VSL works after the latter schema, therefore the privacy policy uses the same behavior, which makes the evaluation process and mechanics more comprehensible to the programmer (<NFR.3>).

4.2.2.5 Decentralized access management

We explain in previous sections how for example XACML uses PDPs and PEPs to manage the policy evaluation. Since the policies are stored on the KA that runs the service, it acts as a decentralized PDP. The requests reach the KA in the RR, as it is shown on Figure 2.6. In the DS2OS, the RR is the PDP that stores and evaluates the policies. The unit responsible for the access control on the node acts as the PEP, where the result from the PDP is enforced, so either access is granted or denied. By implementing the policy evaluation functionality in the RR of the KAs, we make the policy evaluation as decentralized and autonomous as possible. This helps us in keeping the latency low, and also reuses existing structures of the VSL.

4.3 Usability

The aspects considered in the previous sections define the technical functionality, but the usability for the programmer is considered only as a minor aspect. However in order to allow users to define their privacy requirements, the privacy policy has to be accepted by the user. In this section we discuss aspects we found in the literature that are useful for increased usability.

4.3.1 Default policies

Henze et al. argue that privacy novices have to be accounted for while at the same time allowing a fine-grained policy definition for privacy experts [17]. Hong et al. and Davies et al. suggest the use of default policies as they can help the users with integrating the full potential of the policy language in their system [20, 10]. The users can then also use the default policies as a basis to adapt the policy in order to represent their individual privacy understanding better. Especially for the newly introduced data obfuscation interface, it is useful for the user to have an example policy, since the users might not be used to having this extended control over the data. Therefore we decide to implement default policies that can be specifically designed for the type of context.

Since the service package is uploaded by the developer to the DS2OS, and since any user who wants to use this service downloads it to their local VSL, we can define the default

policies in the service package in the service manifest. The developer of a service knows what data types are used, is usually more experienced in the domain, and can also get example policies presented through the DS2OS. Thereby the policy definition is easier for the developer.

However, the users might not want to use privacy policies just yet for their service, or they might want to define multiple policies for their service and later decide at run-time which policies should be in place. Instead of having to delete existing rules to keep them from being evaluated, we make policies activatable. This is explained in more detail in the next section.

The default policies will by default be deactivated, and can be activated if the programmer finds them to be useful. This provides a useful example to the user, without making the system instantly more restrictive. If every default policy would be in place, the functionality of the system would be significantly reduced. The user is more equipped to consider the protection of privacy by restricting data access, and weighing it against service functionality that might need extended data access.

Having default policies in the service packages which are stored in the DS2OS also allows a crowdsourcing of the privacy policies, as users can look at the default policies that are defined for other services of that type in the store.

4.3.2 Activation

We mention in the previous section how default policies can be activated by the user, and that by default they are not activated. Default policies are useful for the user, especially for privacy novices who are not aware of the possible ways they can protect their privacy. However, the user might not care for privacy at the beginning because the system just consists of the user, no external entity exists against which privacy has to be protected. Maybe the service isn't sensitive to privacy, or the user is willing to share the data of this service and cares more about the performance of the system. The user could also wish to keep an existing privacy rule for later, so deleting it means that once the rule is needed again, it has to be defined again, making it more time consuming and error prone.

Those scenarios show that rule activation is a useful mechanism for different use cases and increases the usability of the privacy policy significantly. One disadvantage is that the service manifest becomes too crowded when too many policies are defined, but inactive. This can be counteracted by moving all deactivated policies to the bottom, or to delete rules altogether.

4.3.3 Error behavior

We explain in subsection 4.1.4 how policies in our design evaluate to grant access if no policy is defined. This can now be refined to say that access is allowed if no policy is active.

If a policy is active, but a required search provider is not available, it is good for a higher expressiveness if the programmer can specify what evaluation schema is used.
The programmer can for example decide that high reliability of services is preferred, and therefore cached policy evaluation results are taken as an intermediate solution, e. g. the last location of the queried device is good enough to decide the access decision on.
Another option is that the programmer decides that the privacy requirement is of higher importance than reliability. With this schema applied, the unavailability of the search provider means that the policy cannot be evaluated and therefore no decision can be made. This results in an access denial.
For the prototype implemented for this thesis, access is granted if previously received search provider results are still in the cache. If no cached query results are available then access is denied. A corresponding error message is returned so that the requester knows that under different circumstance access might have been allowed.
This is the behavior of the prototype, for future implementations this error behavior can be made more flexible by allowing the programmer to chose what to prioritize, privacy or performance.

If the policy involves a search provider that is not installed in the VSL, then the policy cannot be evaluated and therefore returns an access denial. In this case, a special error message is returned so that the user knows that access cannot be granted due to the missing search provider.

If an active policy is evaluable and all search providers are reachable, then access is granted if one of the rules evaluates to true, otherwise access is denied. The access grant may be access to an obfuscated data set.

For the obfuscation policies, if no obfuscating entity is present in the VSL, then no obfuscated data can be returned. In this case, this access policy is evaluated to deny access, since we don't want to leak the user's private data. If any other policies are in place which might return data without the missing obfuscating entity involved, then those may return an access grant, therefore policy evaluation is not hindered too much. However an error message is printed so that the user knows that under different circumstances, more or other data might have been available.

4.4 Summary

Following the discussions in this chapter, our privacy policy is declared in the service manifest, designed by Donini [14]. We add the `xPolicy` element to the manifest:

```
{
  "serviceId":"",
  "developerId":"",
  "versionNumber":"",
  "dependencies": [],
  "resourceRequirements":{
    "CPU": {},
    "RAM": {}
  },
  "requiredContextModels":[],
  "conflictingContextModels":[],
  "executableHash":"",

  "xPolicies":{
    "policy1":{
        "active":"true",
        "requestedContext":"temperature",
        "action":"get",
        "policy":"group='family'",
        "obfuscation":""
    },
    "policy2":{
        "active":"true",
        "requestedContext":"temperature",
        "action":"get",
        "policy":"group='guest' and location='house'",
        "obfuscation":"granularity='daily average', timespan='1 week'"
    }
  }

}
```

Listing 4.7: Service manifest file, extension of the work by [14].

The policy is defined in the service manifest of the service package. Default `policies` can be declared by the programmer when the service is created. The default policy is stored in the service manifest and gets uploaded in the service package to the DS2OS. The default policies are by default `deactivated`. The user who downloads and installs the service can decide on what default policies to activate. New policies can be added, following the example of the default policy.
The policies are therefore defined per service. To address the different contexts of this service, the `requestedContext` key works as a vital identifier for the policy evaluation, as the type of the requested context also influences how the policy can possibly obfuscate the data.

The attribute that is added to the service manifest is named `xPolicies`. The `x` is added at the front because the privacy policies are not yet included in the signature

over the manifest. We decide to have multiple additional attributes in our privacy policy, which are stored alongside the rule, such as `active`, or `requestedContext`. The JSON attribute can hold multiple policies with arbitrary names, here `policy1/2`, which can help the user to better understand what the policy protects.

The `action` is the type of the access request, and it can be *get* and/ or *set*.
`Obfuscation` is applied to *get* requests and also depends on the context type. An additional obfuscation service has to be present in the VSL to answer to the obfuscation request of the KA.

The prototype evaluates the policy set consecutively. It first analyzes which policies are applicable. This is done by filtering for the rules that are for the requested context, and that are active. It then evaluates the applicable rules until one grants access. If none grants access, access is denied.
If no policy is applicable, then access is granted, since no restrictions are imposed.
Therefore, for applicable policies it works after the whitelisting approach suggested in the literature.

Chapter 5

Implementation

In this chapter, we look at the prototype that is developed for this thesis. The features that are explored in chapter 4 are used as parameters for the implementation.

A short overview over the code structure is given in section 5.1.
The extended constructor of the service manifest (see Listing 4.7), as well as the Policy class are explained in section 5.2.
An example request to an obfuscation service is shown in section 5.3.
In section 5.4, the issues that are encountered during the implementation are summarized and the steps taken are presented. The limitations of the prototype are addressed.

5.1 Structure

The existing access control is done by the Knowledge Object Repository (KOR) on a request by the Request Router (RR). We therefore implement the privacy policy evaluation in the KOR, using the caching functionality of the node. Specifically, this is the class `KnowledgeRepository`.

The policy evaluation is performed for *get* requests after the group-based access decision is made. For *set* requests, the policy is evaluated first, since the *set* function performs an access check during the call to the function `setValue` in the Node Tree. It has to be implemented for both actions individually, since *set* requests do not employ an obfuscation functionality.

The policy evaluation, and calls to the obfuscation service are both implemented in the `Policy` class. It implements the functions for evaluating time contexts in the policy, since this is the only universal context that is not searchable in the DS2OS.

5.2 Service Manifest

The service manifest as it is extended for this thesis is presented in Listing 4.7. In the code, the existing service manifest constructor is extended by an Object that represents the policy set declared in the JSON manifest. The Policies are mapped onto a HashMap that consists of a String (the name of the individual policies, given by the user), and an object of type `Policy`.
The Policy class has the properties that are also present in the service manifest, namely `active, requestedContext, action, policy`, and `obfuscation`. This simplifies the parsing of the manifest, since the policy objects are directly mapped onto the Policy class.
The service manifest parsing and gathering is only implemented at a rudimentary level, since another thesis extends the existing structure of the service manifest to include more parameters in it [14].

The design includes key-value pairs in the policy rules, such as `time='from 10 to 13'` or `timespan='1 week'`. Passing parameters in this manner helps us with later passing those parameters to the obfuscator, or to the search providers. This is explained in the next section.

5.3 Request formulation

As we explain in subsection 2.3.8, parameters can be passed via the request to a node. Those parameters can then be extracted from the address.
The DS2OS already implements a functionality to return the parameters of the request as a `Map<String,String>`. This makes the evaluation of the passed parameters easy and functional.

Listing 4.7 shows an example policy with possible parameters. For example, if an obfuscation service is located at `/agent1/obfuscator`, the request that includes the obfuscation rule from the example looks like this:

```
get /agent1/obfuscator/&timespan='1_week'&granularity='daily_average'&/nodeValue
```

We also pass the node value, that is to be obfuscated, to the obfuscator.

The passed parameters can then be evaluated and used to modify the value.

5.4 Issues and workarounds

We extend the constructor of the `ServiceManifest` to include the privacy policies. This is shown in section 5.2.
When a new service, that doesn't import the `Policy` class, is added to the VSL, an error because of the unknown type `Policy` is thrown. This is fixed by restoring the original constructor, and listing it alongside the new one. Like this, all services can still be run, and new ones can make use of the policies.
In the future, it is advisable to change the setup of all services to include privacy policies in their service manifest.

As mentioned in chapter 4, the service certificate contains a hash over the service manifest, thereby guarantees its integrity. Once the policy gets changed, the hash also changes, and therefore the certificate has to be updated.
Including the privacy policies in the hash is not part of this thesis. It is therefore declared as a special element, marked by a leading `x` in `xPolicies`.
A future work might include the policies in the certificate, since the certificate is created by the SLCA once the service gets downloaded from the S2Store to the local VSL. Here, the policy and other meta data can be adapted. The service certificate can then be managed at the local site to cope with these changes at run-time. This protects the privacy policy from being unintentionally altered.

A specific obfuscation service is not implemented as part of this thesis, since the focus is on introducing a policy language. However, through the high adaptability of the system, obfuscation services can be dynamically added later, and a type search for a needed obfuscation service can be performed to discover them.
This makes sense especially as the functionality of the obfuscator is dependent on the semantic description of the data.
Therefore implementing an obfuscator for a certain semantic value is not done, but rather a dummy obfuscator is implemented in order to evaluate the performance (see chapter 6.

We have to escape any spaces in the policy rules and the obfuscation parameters, since a request to a node cannot include them. Therefore they are replaced by underscores.

Chapter 6

Evaluation

This chapter evaluates the developed prototype for the requirements declared in section 2.4.
The functional requirements are a *multi-factored privacy decision*, *obfuscation*, *autonomy*, *adaptability*, and *reuse* (<FR.1> to <FR.5>, see subsection 2.4.1).
The non-functional requirements are listed in subsection 2.4.2, namely *complexity*, *expressiveness*, and *understandability* (<NFR.1> to <NFR.3>).

In section 6.1 we address the qualitative factors, how our implementation meets the different requirements that are not of a quantitative nature.

In section 6.2 we look at the general latency that is introduced by the policy functionality. For analyzing the quantitative performance, we evaluate how the implementation of the prototype behaves in different setups, compared to the original system. This addresses **RQ.4**, which aims at keeping the DS2OS at a good performance. As a quantitative measure for the system performance we look at the response time of a request. In subsection 2.4.1 we set the goal that the implementation should ideally cause no higher delay than 0.1 seconds, as this is the delay that becomes noticeable by users. At a maximum, 1 second response time is tolerable.
We perform measurements to gain an understanding of how the added functionality for the policy processing influences the system performance. For this, we measure how much the response time to a request is longer for evaluating different types of policies, compared to the original implementation of the DS2OS.

Lastly, we analyze how the characteristics of a policy influences the latency in section 6.3.

6.1 Qualitative evaluation

In the research question for extended access control (**RQ.1**), we address <FR.1> and <FR.4>, by looking for an extended access control mechanism that takes more contexts than just the group based attributes of an entity into consideration.
Our implementation offers the possibility to name diverse privacy contexts in the privacy policy, as the evaluation capabilities are only limited by the search providers available in the system. Therefore this goal is met, and the system is kept adaptable to new domains, as the policy is not tailored to a specific use case.

The **RQ.2** addresses the need for a policy language that is both expressive and understandable. This covers the requirements <FR.2>, <FR.5>, <NFR.2>, and <NFR.3>.

One factor that guarantees a high expressiveness (<NFR.2>) of the privacy policy is achieved by implementing the interface to an obfuscation service functionality (<FR.2>). This allows the user to declare parameters along which the context of a service should be altered to fit the user's privacy need. The possible parameters that can be passed to the obfuscation service are only limited by the implemented parameters of the obfuscation service. Therefore the obfuscation rules are highly expressive and can be adapted to new requirements easily.
Obfuscation is a privacy functionality that is not used in many cases in related works, as we see in chapter 2 and chapter 3.

Another factor for an expressive policy is the use of the same functionality for the possible contexts for which a privacy policy can be declared. As long as a fitting search provider exists, any context can be semantically discovered and analyzed in the VSL. Therefore it is extensible and expressive.
By using logical connectives to express the privacy policy, the policies are easy to evaluate in our implementation, and are still understandable enough for the programmer of a service. The programmers have a basic understanding of how the logical connection of different parts works, and by using natural words for the logical connections in the policy (*and, or, not*), even novices are able to figure out what the policy conditions do (<NFR.3>). However, other policy languages achieve a higher understandability by using natural language. Chapter 8 highlights some aspects which can further improve the understandability of the policy.

By using the search providers to semantically discover relevant context, we also adhere to the functional requirement of reuse (<FR.5>). We reuse the service manifest to declare the policies, thereby integrating the policies into the declaring entity of system requirements.

In research question **RQ.3** it is declared that the complexity of the DS2OS should be kept at the current level. This is achieved by declaring all functions related to the policy evaluation in a new Policy class. The obfuscation functionality is currently also part of this class, but can be extracted and individualized in the future. The privacy policy evaluation is called from the function that also checks access, so it is at the location where a programmer looks if the access control should be changed.
Reusing the existing semantic discovery of context, and only extending the access control also keeps the system complexity at the current level.
For the user, no additional complexity is introduced if the privacy protection is not used. The usage of the system is therefore maintained at the previous complexity level. The access granting behavior changes with the policies, as the data of a node may now be altered depending on the obfuscation policy rules. However by declaring privacy restrictions only for the current service, and not for the possible child nodes, the general access scheme is continued and thereby understandable.

6.2 Latency

In this section we test the quantitative performance of the prototype by using different setups, so the unchanged system with the *original code*, the setup with a *policy and obfuscation evaluation functionality* in place, and the setup that additionally has *caching* functionality for the results of the obfuscator. The latency is tested by measuring the time difference of when a *get* request is sent out, to the time the result is received at the requester. *Set* requests are not tested, since obfuscation is not applicable to them, and we can measure the latency of a fine-grained access check for *get* requests already.

We use different policy sets for our evaluation.
First, we test a setup where *no privacy policies* are restricting the access.
If policies are used, they involve a relatively simple location policy that restricts access to entities located in the living room. This therefore involves a request to a location search provider. We evaluate cases where the *access is allowed*, or *denied* after the result from the search provider is received. A combination of those policies is also tested, and in order to keep the charts comprehensible we abbreviate an *allow* policy to **1a**, and a *deny* policy as **1d**. Ten *deny* policies are therefore abbreviated to **10d**.

For the tests, the service manifest with the policies is read from a specific location on the system. This approach is chosen to make the changing of the policies more convenient for testing. For using the policies in a running system, they will be gathered from the service metadata stored in the NLSM, and then parsed for the policies.

The measurement results are displayed in the Appendix in Table A.1 and Table A.2.

6.2.1 General

We want to gain an understanding of how introducing the privacy policy functionality influences the system in terms of latency. We expect that the response time is slower for all requests, since at least processing the manifest file takes some time.

In order to test how quickly the system responds to requests with and without policies installed, we test the latency for local, and remote requests in both the original system, and the new implementation that can evaluate privacy policies and obfuscate data. However, no policies are listed in the service manifest just yet, in order to get the raw processing latency of parsing the manifest and other additional functionalities we have in our implementation.
Caching is analyzed in subsection 6.2.4, as we first want to see the general processing differences.

We send out five requests to an installed service, once from a local node that runs the service, and in the other case from a remote node. Having five requests to analyze gives us a minimal understanding of the performance, and keeps the evaluation setup manageable. Multiple first requests would mean to shut down the entire system and restart it, which is considered as too extensive, especially since already the first measurements showed a recurring value range.

We expect that the latency is increased only slightly, since no requests to external entities are sent, only the manifest file is processed.
Figure 6.1 shows the results of this measurement in milliseconds.

The latency of the first request is higher than, or at least equal to all other latencies for that request type. This is because caching is already done by the VSL, so after the first request, an answer is sent out faster. The requests after the first are more interesting to us, since they give a more balanced view of the processing latency. We can also expect that in a real world use case, requests are repeated, since the evolution of a value is of interest.

We can see in Figure 6.1 that for local requests, the difference is not very high, the average latency is higher by 2.5 ms for the policy augmented system.
For remote requests, we even have a reduced latency, which is surprising. This could be due to a measuring error, but since the average measurements are only 4 ms apart, it is considered as a valid result.

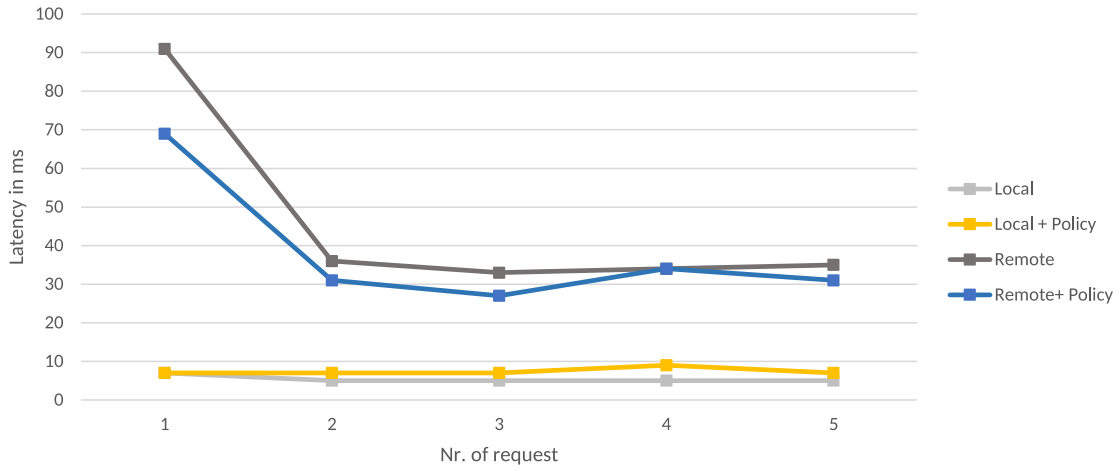

Figure 6.1: Latency introduced through policy processing functionality in code, no policies are processed.

As conclusion it can be said that adding the processing functionality introduces something around 0 to 3 ms latency. This result can be used as comparison in later measurements.
Therefore having our functionality implemented slows the system down to a negligible extent.
First requests take around 7 ms for local, and around 80 ms for remote requests. Local requests are on average up to 7.5 ms long, remote requests up to 35 ms.

6.2.2 Search provider queries

Since the introduction of the functionality to process policies keeps the reaction time of the system at roughly the same value, we now test how high the latency of processing a single policy is. For this, we add a policy that restricts access to entities located at a specific location to the manifest. The policy mechanism queries a location search provider for the devices at this location. In our case, the requester is listed as being in that requested room and therefore the privacy enforcing mechanism grants access to the requester. In the charts, this is abbreviated to `1a`.

We test this setup for local and remote requests, and compare the results to the response times of the original setup.
We expect that a significant latency is now taking place for the requests. For local requests, it should stay below the original latency of remote requests, since requesting a local service and a local search provider should not take longer than requesting a remote service.

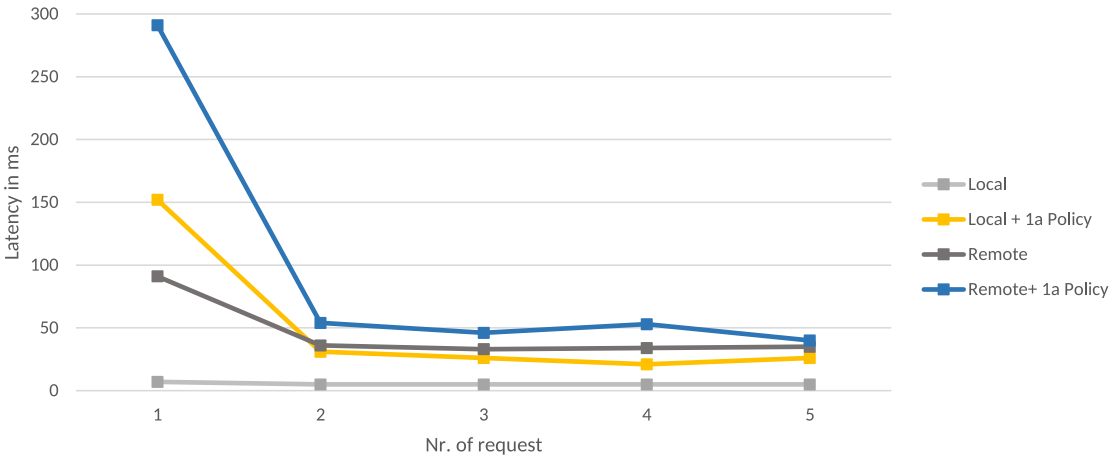

Figure 6.2: Evaluating the total latency introduced by processing polices, at the example of a simple policy that allows access.
Original request latencies are included for comparison.

Remote request delays are probably higher than with the original setup, but the difference is likely to be somewhere around the difference that we see for local requests.

In Figure 6.2 we can see that indeed the latency is now higher than the original response time. Especially the first requests are high, even local requests are now surpassing the formerly slowest response time. The first remote request has a response time of around 290 ms, and a first local request is roughly half as long ($\tilde{1}$50 ms). The average requests are also roughly doubled, from 26 ms for local request to 48 ms for remote requests. Originally, remote requests were seven times slower than local requests on average.

For a human user the difference should not be notable for requests on average, as at worst it is only 10 ms slower than previous remote requests. Local requests are still faster than the original remote requests.
So, if the majority of the services do not use a policy, then the latency is still in manageable value ranges. If however the majority implements a single privacy policy, the latency would be noticeable. In section 6.3 we analyze how different policy conditions alter the behavior.

When comparing the latency for the first request to the original latency for first requests, 145 ms are added to the latency for local requests, or 200 ms for remote requests. On average, this is 21 ms for local requests, and 14 ms for remote requests. It is surprising that local requests on average introduce more latency to the system than remote requests. This could either be a measuring error, or due to the big difference in latency between remote and local requests.
For first requests, the latency becomes noticeable, as the latency surpasses 100 ms. This

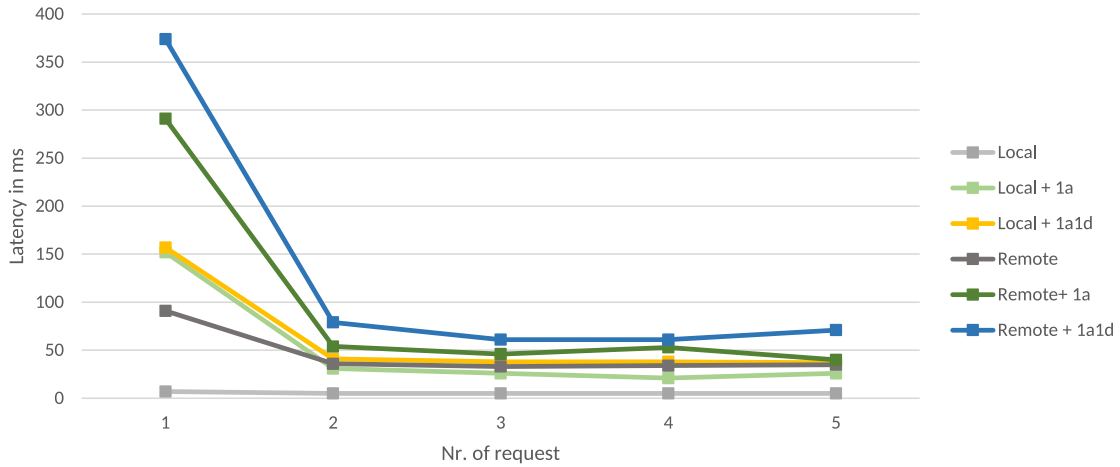

Figure 6.3: Comparing a policy set that first denies, then grants access (`1d1a`), to a directly access granting policy (`1a`).
Original request latencies are included for comparison.

contradicts our goal to stay below 100 ms of delay, but since this high latency is only reached by first requests, it is tolerable.

So, as conclusion, requesting a result from a search provider introduces a latency between 14 ms and 21 ms on average. This is still an acceptable latency. The latency for a first request to a search provider is high, but only occurs once, so this is still functional.

For another evaluation, we analyze the latency that one additional policy introduces, by using a policy set where the first policy requires that the requester is in a room (returns an access denial for our setup), and the second policy is the same as in the previous measurement (returns an access grant). In the charts, this is abbreviated to `1d1a`.

We expect that the latency is further increased, only not as much as the previous measurement added to the response time, since only another request to the location search provider is performed.

We compare these results with the previous tests, so the original setup, and the policy that directly grants access (`1a`) in Figure 6.3.

It shows the `1d1a` measurements in yellow and blue, and the previous results in green. The difference between the local requests which include a policy is still low, especially for the first request. However, on average, the local request is now for the first time slower than the original remote request, being at 39 ms, compared to 35 ms for a simple remote request. This means that the difference becomes more and more noticeable by users, especially for remote requests that have a policy.

We see that the additional deny policy, that has to query for another room, introduces a latency of 5 ms for local, and 83 ms for remote first requests, compared to the previous measurements (green). For the remaining requests, having a deny policy increases the latency by 13 ms for local, and by 20 ms for remote requests. This is a small latency, and won't slow the system down significantly on it's own.
However we cannot directly compare the values from the previous measurement with this one, as a certain base of processing time is part of the previous request. In section 6.3 we, beneath other factors, look at the latency that is introduced by one single policy.

6.2.3 Obfuscation

In order to evaluate the latency the obfuscation mechanism introduces, we use the policy set of the previous measurement, and extend it to include a simple obfuscation rule. The obfuscation functionality we use for our setup takes a node value, checks the value, and returns an obfuscated node value. The obfuscation service also runs on the local agent.

We expect that the obfuscation functionality introduces some latency, as another request to a service is sent with the corresponding obfuscation rules and the value. The obfuscation service has to be discovered, which also introduces some latency. The additional latency will therefore probably be somewhere around the latency introduced in the first measurement (21 ms on average for local requests, and 14 ms for remote requests).

Figure 6.4 shows the measurement results. It is surprising that the first local request using obfuscation comparatively takes longer, compared to the first remote requests, only 41 ms latency is introduced for first remote requests, whereas the local first request is 130 ms slower when using obfuscation. This difference is quite high and must therefore be a measuring error, since the remote request involves the local agent, and therefore a remote requester gains at least the latency that the local agent experiences. It therefore should be lower, or the value of the remote request should be higher.

The average values are more useful for us, and they show that the difference between **obfuscation** and no obfuscation is on average either **27 ms** for local agents, and **26 ms** for remote agents, so roughly the same. This means that we can take this value as the latency that is introduced to policies on average, when they use obfuscation.

This is slightly higher than the expected latency, but it is still within a low range.

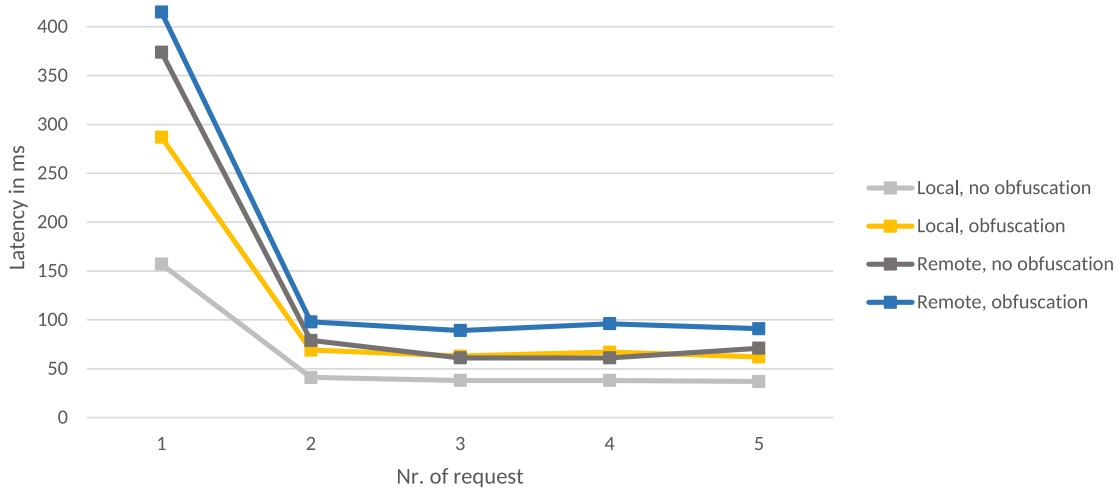

Figure 6.4: Latency introduced through obfuscation, by evaluating a policy set that first denies, then grants access (`1d1a`), either with or without obfuscation rules in the policy.

6.2.4 Caching

As a countermeasure to high latency, we introduce caching of previous obfuscation results. We measure the advantage of caching by again applying a policy that first denies, and then grants access (`1d1a`). We also compare it to the original response times, in order to get an overview over the total latency.

We expect that the response time is further increased for first requests, compared to the previous obfuscation policy results. This increase should be only a slight increase though, as the only difference is that the result from the obfuscation service has to be cached. After the value has been cached, all further requests with this setup should be significantly faster, at least coming close to the performance we have when we are only processing policies, no obfuscation rules.

The results are split into two charts, because the time difference between the first requests, and the requests that come after it, is too big and the effect becomes clearer by looking at the results separately. Figure 6.5 therefore shows the effect of caching for the first requests sent by either a local, or a remote agent. Figure 6.6 shows the same, but for an average over all policies coming after the first one.

In Figure 6.5 we see that for first requests, the latency is already improving. This is unexpected and might indicate a measurement inaccuracy. For the remote requests it is even faster than the functionality without any obfuscation rules involved, so this is an unexpected result. Even for first requests, the latency is reduced by 5 ms for local, and by 20 ms for remote requests. This difference is not too big, so we consider it as

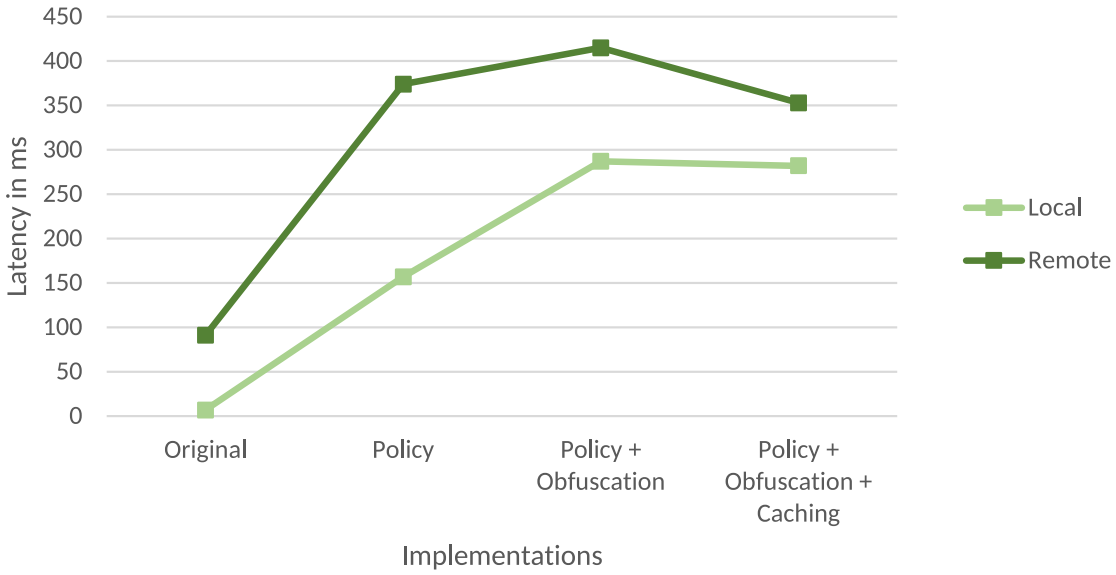

Figure 6.5: Latency on first request, introduced by the different implementations.
Testing for either a local or remote request with a policy set that first denies, then grants access (`1d1a`).

being slightly off.
As we state in the expected outcome, the response time for first requests should at least be as long as the previous requests without caching.

Over all, this setup has a latency of 282 ms for first requests locally, compared to 7 ms in the original setup. For remote requests, it is even higher, at 353 ms compared to 91 ms. The difference to the original latency is therefore 275 ms for local, and 262 ms for remote requests.
This latency for first requests is high, the remaining requests should bring significant improvement if we want to keep the system functional. Therefore we look at Figure 6.6 now.

We see that for local requests, the latency is 45 ms, thereby approaching the performance as it is without obfuscation rules (39 ms). This is how we expected it to be, reducing the latency by **20 ms** compared to the previous measurements.
For remote requests, the value is even lower, which is surprising, but since the difference is 3 ms, this can be disregarded. For remote requests, the performance is increased at a higher rate, since caching the obfuscation result locally also saves the request to the remote obfuscator and thereby further communication delay. We save **28 ms** by caching, compared to the previous measurements.
Therefore caching proves to be highly useful for the setup, at least for all requests after the first ones.

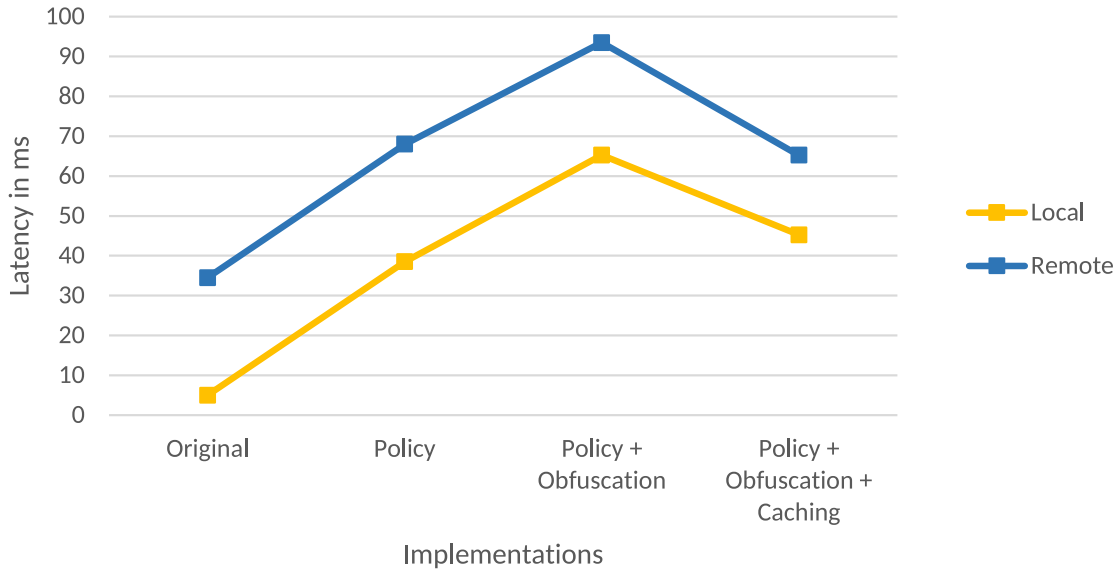

Figure 6.6: Latency on average, introduced by the different implementations.
Testing for either a local or remote request with a policy set that first denies, then grants access (`1d1a`).

6.3 Policy characteristics

This section looks at how the characteristics of a policy set change the performance of the system. We want to know whether a large policy set with simple policies introduces higher latency, than a policy set with few, but long policies. For this we measure the average latency of different policy sets that use obfuscation. The results are shown in Figure 6.7.

In subsection 6.3.1 we analyze the latency introduced by long policy rules. A long policy in our test setup consists of five requests to a location search provider, and two requests to a time evaluating functionality. The time evaluation is done on the node itself, since this is not dependant on any external logic. Therefore the majority of latency that is introduced comes from the location search provider.

This is compared to the number of policies in the policy set in subsection 6.3.2. For our evaluation setup, a policy set that contains many policies consists of 10 access denying policies and one access granting rule, all of which have the same obfuscation rules as in the previous measurements.

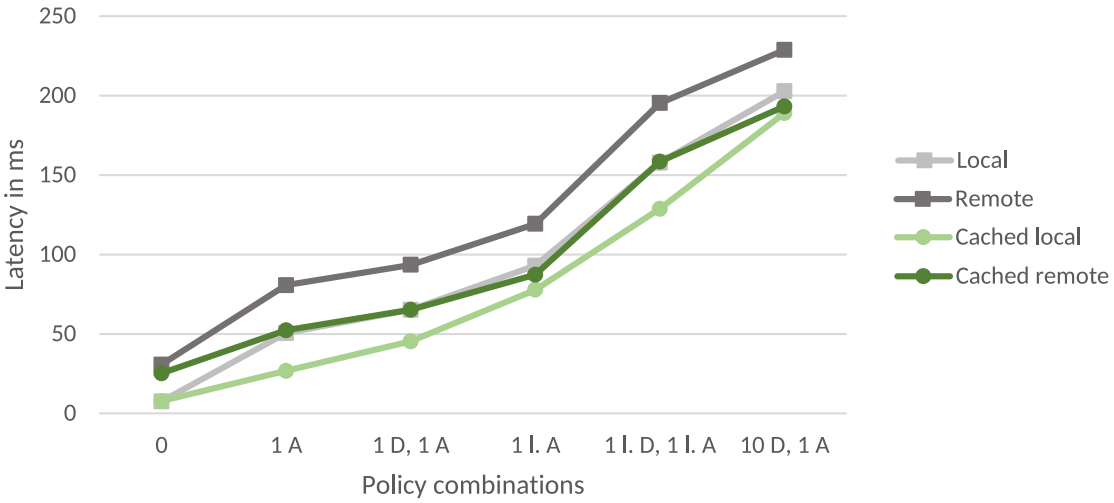

Figure 6.7: Latency introduced by the complexity of the policy set, analyzed for local and remote requests, and considering the usage of caching.

6.3.1 Policy length

A long policy should take longer to evaluate than the previous policy sets, because the different nested logical connections of the policy have to be evaluated and then combined to return an access decision.

We expect that this increase is minimal, introducing around the same latency as previous additional policies did (see subsection 6.2.2).

Figure 6.7 shows how the complexity is indeed increased, and that for example the difference between a short access granting, and a long access granting policy (`1a`) is increased notably.

We also see how caching makes the data points for local and remote policies almost meet for 10 deny rules, with one following access granting rule (rightmost light and dark green data points). The difference is only 4 ms. Since the 10 deny policies are requesting the same value, this makes sense, as caching is effective for repetitive requests.

For the requests that use no caching (grey lines), it can be seen how the latency difference remains the same throughout all policy sets. On average, they are 29 ms apart.
For the cached requests, the latency differs throughout the policy sets. It is highest for the long `1a1d` rules, which is plausible, since they involve two different requests for multiple search provider results. The difference is lowest for the last data point, as mentioned above.

So, if we have long, complex rules, it is better to have caching enabled, since it reduces the latency for remote requests.
We want to see the true cost of a long request which consists of five location restrictions, compared to a short request, consisting of one location restriction. For this, we subtract the latency of the long access grant (`1l.A`) policy evaluation from the long deny-allow (`1l.D, 1l.A`) evaluation, and divide this result by the number of search provider queries involved in the policy evaluation, so five. This gives us **14.25 ms** (159 ms minus 87 ms, divided by 5) for remote, and **10.2 ms** (129 ms minus 78 ms, divided by 5) for local **requests per search provider involved in long rules**.

6.3.2 Number of policies

Figure 6.7 we see that the latency is highest for a policy set with multiple policies. This is no clear indicator of what influences the latency the most, as these large policy sets (`10d1a`) consist of 11 requests to a search provider, and the small policy set that has long policies (`1l.D,1lA`) has nine requests to a search provider, but combines the results with logical expressions that have to be evaluated. So in order to get an understanding for the latency that a big policy set introduces by first evaluating multiple denying rules, we subtract the latency caused by one deny - one allow policy set (`1d1a`) from ten deny - one allow policy set (`10d1a`) and divide this result by nine, in order to get the latency of one deny policy on average.
In numbers, this is 193 ms minus 65 ms, divided by 9, resulting in **14.2 ms** delay for one remote **request to a search provider in complex policy sets**. For local requests, this is 189 ms minus 45 ms, divided by 9, gives as a result about **16 ms** for local requests to a search provider.
It is surprising that local requests to a search provider on average take longer, but since the values are close to another, this can be seen as the range of delay for this type of request.

Comparing those values to the results of the previous section, we see that a request to a search provider on average costs 10 to 14 ms in long rules, and between 14 and 16 ms for complex policy sets. Therefore it can be said that a higher number of policies in a policy set makes the response time about 3 ms slower than a single long policy rule does. However this difference is minimal, and therefore the evaluation of long policies does not have a significant impact on the evaluation time. The increase in latency is mainly based on the number of search providers that have to be queried.

6.4 Conclusion

Now we know that especially the first requests to a service are slowed down significantly compared to the original response time. The highest latency we measure is for remote requests with a policy set that contains many deny policies, this can be up to 461 ms additional latency. By using caching, we can reduce the additional latency to around around 400 ms. This is still a high value, since it is slower then 100 ms we aimed at in subsection 2.4.1, therefore it causes noticeable latency [26].
However, the high latency is only produced by first requests, the additional latency produced by later requests is at most around 200 ms (large policy set, local requests without caching). With *caching*, this additional latency is at least reduced to 184 ms. This is still high, but should not be a normal use case, as a policy set with many policies also is less understandable for a user, and therefore is less likely.

A more realistic use case is one where two policies are active and applicable, and the first policy denies access, followed by a policy that grants access (`1d1a`). For this, the response time is around 40 ms slower as in the original setup for local requests, and about 31 ms slower for remote ones. This is a comparatively low number, but especially for local requests, this is nine times slower than the original request. Therefore this additional latency will be noticeable once more services in the network have applicable policies in place.

The latency we add to a request by passing its value to an *obfuscator* is around 26 ms, as we discuss in subsection 6.2.3. On its own, this is not a high latency, but combined with other time intensive rules, this causes a longer response time. Therefore obfuscation is a factor that the user should consider in regards to if the performance loss is at balance with the privacy protection it offers.

As we see in subsection 6.3.2, the latency introduced by one request to a *search provider* is around 14 ms. Therefore the response time of a service using policies depends strongly on the number of search providers involved.

We notice a major improvement in the response time when obfuscation results are cached at the nodes, for local requests caching decreases the response time by 20 ms, for remote requests it is even more, about 28 ms. So far, cached results are only a fallback in case of an unreachable obfuscation service. In order to increase performance in the future, the obfuscation results could be cached and used even if the obfuscation service is available. In case the policy changes, and therefore the obfuscation rule might have been changed, a new query to the obfuscation service has to be performed.

The latency introduced through querying search providers is even more relevant, as we expect this to be a standard part of the privacy policy. Using cached results of the search provider is not advisable since for example the location of a device can change often. Unless the search provider becomes unavailable we want to have the most recent result from the search provider.

As a conclusion it can be said that for first requests, we do not meet our goal of staying below the 100 ms delay, as we have measured a delay of 488 ms with our test setup. We meet the lax goal of staying below a delay of 1 second.
For the requests after the first, we in most cases meet our goal of staying below the 100 ms delay, therefore keeping the impression up that the reaction is instantaneous. We measure latencies higher than 100 ms only for policy sets that included calls to nine or ten search providers. Since the delay introduced by search provider requests is about 14 ms, we can estimate that a maximum of five to six search providers should be requested by the policy for the most common use cases, if the reaction time should stay below 100 ms.

In chapter 8 we list some possibilities that can reduce the latency observed in this chapter.

Chapter 7

Conclusion

This thesis approached the aspects of increasing the privacy protection in smart environments, at the example of the DS2OS implementations. Access control is usually done based on group membership or roles, which has the disadvantage that a high number of roles is accumulated over time if more fine-grained access control is needed. The goal of this thesis was to extend this access control to include other aspects into the access decision which might more accurately represent the users understanding of privacy.

In analyzing related research and other implementations, we found that Attribute-based Access Control (ABAC) is the most fine-grained and adaptable access control. The attributes are a semantic description of the environment and are assigned to the entities. A policy can use this semantic description to express a rule for a certain attribute, e. g. the location of an entity.
The advantage of implementing an attribute based access control at the example of the DS2OS is that a semantic description of the entities already exists, and context is semantically searchable through search provider services. Therefore we can include any attributes that are discoverable through search providers in the system.
We therefore solved the problem of creating a fine-grained access control. The individual attributes are linked through logical expressions, which are known to the user. Since the search providers can be designed to discover context at any level of detail, the policy becomes expressive and adaptable to diverse use cases.

An additional privacy mechanism that is not part of ABAC and which has been used in few other cases is data obfuscation, which removes or alters certain sensitive data. This way, sensitive data is kept privacy but can still be processed, at a level of detail that is accepted by the user, and that might still be useful enough for the service. This also

helps to maintain a good system performance.
By combining the attribute based privacy policy with an obfuscation mechanism, the users can not only decide who has access based on more factors, but can also alter the exposed data itself based on the same fine-grained attributes. This further extends the expressiveness of the policy.
The understandability of the policies is good enough for the prototype, where the programmers implement and maintain the policies. By creating the possibility for default policies, the understandability is further increased. An ideal implementation could improve this by offering a GUI, which we elaborate in chapter 8.

The design developed for this thesis extends the existing access control functionality of the DS2OS with an attribute based access policy, which reuses the existing semantic description of context. An interface to obfuscation services was created, and both the attribute based access policies, and the obfuscation rules are declared in a policy set. The policy set is defined per service and enforces the privacy rules for any requests to this service.

The evaluation of the performance of the prototype shows us that the response time of requests is slowed down by around 14 ms per search provider involved, and about 26 ms latency is caused by the query to the obfuscation service. Caching proved to be a good approach to reduce the latency, as it can save up to 28 ms of response time. Therefore the performance of the system is slowed down, but for use cases that do not involve more than five to six search providers the latency stays below 100 ms, which was the requirement for this thesis. The latency can be reduced even further when caching is also used for the search provider queries.
The autonomy of the VSL was maintained, since the policies are part of the service manifest, stored in the NLSM. The policies are therefore locally available and the processing of the rules is also done autonomously. The KAs only have to involve external entities when a search provider is requested, which might not be available on the local node. By caching previous results from search providers, the autonomy could be further increased, but at the expense of an absolute privacy protection.

Ultimately, the owner of a pervasive computing environment has to decide how important privacy protection is in their setup, and weighing this against the performance decrease that is introduced by checking the privacy conformity of an access request. The increased latency of the privacy policies in common cases does not slow the response time down to values higher than 100 ms, but even this might be too slow for time-critical operations. By enabling diverse caching possibilities, the owner can find a good compromise between privacy and performance. Chapter 8 discusses some further caching opportunities.

Chapter 8

Future Work

As the previous chapter summarized, the majority of our requirements for an expressive and understandable privacy policy are implemented in the current prototype. This chapter explores features that would further improve the privacy policy functionality, which we did not realize in the prototype.

In pervasive environments, the large number of devices and the high connectivity of the devices is the defining property. Therefore the communication between the entities has to be fast. The latency of the prototype is still unnoticeable in regular cases for the user. However with an increased number of policies, or a bigger network where a search provider might not be as quickly addressable as in the presented test setup, the latency will become too high. This could lead to frustration with the privacy functionality and would make the owner of a network disable the privacy protection mechanism. This is not a desirable outcome, therefore the utmost priority for further improvement of the prototype is to reduce the latency of policy processing.

One major factor that influences the performance of the policy evaluation is the querying of different search providers to determine the parameters that are relevant to a policy. In the prototype, the search providers are queried for every element of the policy anew, since for example the location of an entity can change between different access requests. This of course introduces latency, since every query to a search provider takes around 14 ms (see chapter 6). In future implementations it might be one solution to offer the possibility to the programmer of a service to define a desired behavior for querying search providers. Either they should be queried at a regular interval and between those intervals, cached values are used, which would return results faster, but is less protective than newly requested context. Alternatively the desired behaviour could be the current

implementation, that new values should be queried for every request. It should be kept in mind that the first behavior would cause a constant traffic in the background.

Another factor that introduces latency is that the address of the corresponding search provider is looked up via a type search at every requests. An improvement to this could be to cache the address of for example the location search provider once it was discovered. If the search provider at the cached address becomes unavailable, a new type search for another search provider of that type can be done.

The requests to the obfuscation service introduce some latency, and caching proved to be a viable way to improve the response time. As it is explained in chapter 6, the cached value is only used when the obfuscation service becomes unavailable. The obfuscation will not change with changing environment attributes, therefore the cached obfuscated values can be used, as long as the passed values stay the same.

Another minor but easy way to improve the latency would be to cache the policies of the service on the node, instead of reading them every time from the manifest file. Since a changed policy set would also mean that the manifest changed, we can rely on the policies we read at startup of a service.

An aspect that is considered in the design process is to allow the programmer to set a preferred functionality of the policy evaluation. The programmer could be offered a way to specify if privacy or performance are of greater interest, and according to this choice, the evaluation either uses caching extensively, not at all, or at the balance that is proposed in this thesis.

A useful extension of the policy evaluation presented by this thesis would be to have policies that are valid for all contexts of the service. In our prototype, the first matching rule is evaluated. Therefore if another rule would grant access to a bigger data set, it is not reached in our prototype.
However this can require the user to define repetitive policies for all context. To avoid listing repetitive rules, an advanced evaluation mechanism could state generally valid rules in addition to the context-specific rules.

Aspects which are mentioned in the literature, but which we didn't implement in the prototype are a Graphical User Interface (GUI), and policy conflicts. With a GUI, the policy declaration becomes far more developer friendly, since suggestions about available search providers can be presented. Furthermore, common privacy policies for the context that is processed by the service could be suggested. A GUI can help the programmer understand the policies better, and could possibly help them in activating rules, or removing unwanted ones.

An overview over the activated policies could be given to the users in the VSL, which helps them to understand what policies are in place, and how they possibly restrict their access.
Policy conflicts are currently unlikely, since the hierarchy of the services is not important to the privacy enforcement, as policies are only valid for the specific service they are declared for, not for any other services in the VSL.

Chapter A

Appendix

	Original		Policy		Policy & Obfuscation		Policy & Obfuscation & Caching	
	First	Average	First	Average	First	Average	First	Average
0 policies	7.00	5.00	7.00	7.50	7.00	7.50	4.00	7.75
Short policies								
1 allow			152.00	26.00	281.00	50.50	291.00	26.75
1 deny, 1 allow			157.00	38.50	287.00	65.25	282.00	45.25
10 deny, 1 allow			302.00	184.00	467.00	202.75	455.00	189.00
Long policies								
1 allow			198.00	70.75	344.00	93.00	200.00	77.75
1 deny, 1 allow			282.00	135.00	419.00	157.75	258.00	128.75

Table A.1: Measurement results from sending **local** requests. Testing the original system against different implementation setups, using different policy sets (in milliseconds).

	Original		Policy		Policy & Obfuscation		Policy & Obfuscation & Caching	
	First	Average	First	Average	First	Average	First	Average
0 policies	91.00	34.50	69.00	30.75	69.00	30.75	57.00	25.25
Short policies								
1 allow			291.00	48.25	509.00	80.75	369.00	52.25
1 deny, 1 allow			374.00	68.00	415.00	93.50	353.00	65.25
10 deny, 1 allow			363.00	195.25	552.00	228.75	488.00	193.25
Long policies								
1 allow			367.00	91.00	550.00	119.25	218.00	87.25
1 deny, 1 allow			356.00	157.75	507.00	195.25	494.00	158.50

TABLE A.2: Measurement results from sending **remote** requests. Testing the original system against different implementation setups, using different policy sets (in milliseconds).

Chapter B

List of acronyms

ABAC	Attribute-based Access Control.
CA	Certificate Authority.
CMR	Context Model Repository.
DAC	Discretionary Access Control.
DAML	DARPA Agent Markup Language.
DS2OS	Distributed Smart Space Orchestration System.
FOAF	Friend of a Friend.
GDPR	General Data Protection Regulation.
GUI	Graphical User Interface.
IoT	Internet of Things.
JSON	JavaScript Object Notation.
KA	Knowledge Agent.
KOR	Knowledge Object Repository.
MAC	Mandatory Access Control.
NLSM	Node Local Service Manager.
ORSD	Ontology Requirements Specification Document.
OS	Operating System.
OWL	Web Ontology Language.
P2P	Peer-to-Peer.
PDP	Policy Decision Point.

PEP	Policy Enforcement Point.
RBAC	Role-based Access Control.
RCC	Region Connection Calculus.
RDF	Resource Description Framework.
RR	Request Router.
S2S	Smart Space Service management.
S2Store	Smart Space Store.
SLCA	Site-Local Certificate Authority.
SLSM	Site-Local Service Manager.
SML	Service Management Layer.
SOUPA	Standard Ontology for Ubiquitous and Pervasive Applications.
VSL	Virtual State Layer.
XACML	eXtensible Access Control Markup Language.
XML	Extensible Markup Language.

Bibliography

[1] Luigi Atzori, Antonio Iera, and Giacomo Morabito.
„The internet of things: A survey“.
In: *Computer networks* 54.15 (2010), pp. 2787–2805. URL: https://www.sciencedirect.com/science/article/pii/S1389128610001568.

[2] Axiomatics. *How Can I Use Date in a XACML Policy?* July 2016. URL: https://www.axiomatics.com/blog/how-can-i-use-date-in-a-xacml-policy/.

[3] Axiomatics. *Understanding XACML combining algorithms*. July 2014.
URL: https://www.axiomatics.com/blog/understanding-xacml-combining-algorithms/.

[4] Dan Brickley and Libby Miller. *FOAF Vocabulary Specification 0.99*. 2014.
URL: http://xmlns.com/foaf/spec/#sec-crossref.

[5] Fang Chen and Ravi S Sandhu. „Constraints for role-based access control“.
In: *Proceedings of the first ACM Workshop on Role-based access control*.
ACM. 1996, p. 14. URL: https://dl.acm.org/citation.cfm?id=270177.

[6] Harry Chen et al.
„Soupa: Standard ontology for ubiquitous and pervasive applications“.
In: *Mobile and Ubiquitous Systems: Networking and Services, 2004. MOBIQUITOUS 2004. The First Annual International Conference on*.
IEEE. 2004, pp. 258–267.
URL: https://ieeexplore.ieee.org/abstract/document/1331732/.

[7] Harry Chen, Tim Finin, and Anupam Joshi.
„The SOUPA ontology for pervasive computing“.
In: *Ontologies for agents: Theory and experiences*. Springer, 2005, pp. 233–258.
URL: https://link.springer.com/chapter/10.1007/3-7643-7361-X_10.

[8] Delphine Christin et al.
„A survey on privacy in mobile participatory sensing applications“.
In: *Journal of systems and software* 84.11 (2011), pp. 1928–1946.
URL: ftp://130.83.198.178/papers/CRKH11.pdf.

[9] Open Geospatial Consortium.
GeoSPARQL - A Geographic Query Language for RDF Data.
https://portal.opengeospatial.org/files/?artifact_id=47664. 2011.
URL: http://www.opengeospatial.org/standards/geosparql.

[10] Nigel Davies et al. „Privacy mediators: Helping IoT cross the chasm“.
In: *Proceedings of the 17th International Workshop on Mobile Computing Systems and Applications.* ACM. 2016, pp. 39–44.
URL: https://dl.acm.org/citation.cfm?id=2873600.

[11] Bernhard Debatin et al. „Facebook and online privacy: Attitudes, behaviors, and unintended consequences“.
In: *Journal of Computer-Mediated Communication* 15.1 (2009), pp. 83–108.
URL: https://onlinelibrary.wiley.com/doi/full/10.1111/j.1083-6101.2009.01494.x.

[12] Anind K Dey, Gregory D Abowd, and Daniel Salber.
„A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications“.
In: *Human-computer interaction* 16.2 (2001), pp. 97–166.
URL: https://dl.acm.org/citation.cfm?id=1463110.

[13] Colin Dixon et al. „An Operating System for the Home“. In: (2016).
URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/homeos.pdf.

[14] Lorenzo Donini.
„Autonomous Certificate Management for Microservices in Smart Spaces“.
MA thesis. Technische Universität München, Aug. 2018.

[15] Qiang Ge et al.
„The Application of SWRL Based Ontology Inference for Privacy Protection.“
In: *JSW* 9.5 (2014), pp. 1217–1222.
URL: http://www.jsoftware.us/vol9/jsw0905-21.pdf.

[16] Daniel Giusto et al.
The internet of things: 20th Tyrrhenian workshop on digital communications.
Springer Science & Business Media, 2010.

[17] Martin Henze et al.
„A comprehensive approach to privacy in the cloud-based Internet of Things“.
In: *Future Generation Computer Systems* 56 (2016), pp. 701–718. URL: https://www.sciencedirect.com/science/article/pii/S0167739X15002964.

[18] Jerry R. Hobbs. *A DAML Ontology of Time.* 2003.
URL: http://www.cs.rochester.edu/~ferguson/daml/.

[19] J Hong and M Langheinrich. „Privacy challenges in pervasive computing“. In: *Computing Now* 7.6 (2014). URL: https://www.computer.org/web/computingnow/archive/june2014?lf1=267870609d109616043122e23422218.

[20] James Hong et al. „Don't Talk Unless I Say So! Securing the Internet of Things With Default-Off Networking“. In: (2018). URL: https://sing.stanford.edu/site/publications/bark-iotdi18.pdf.

[21] Chung Tong Hu et al. *Guide to Attribute Based Access Control (ABAC) Definition and Considerations*. Tech. rep. 2014. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-162.pdf.

[22] Suzana Iacob and Antonis Bikakis. „Evaluation of Semantic Web Ontologies for Privacy Modelling in Smart Home Environments.“ In: *PrivOn@ ISWC*. 2016. URL: https://pdfs.semanticscholar.org/cd84/9b2f3a7d7df22f1f54a6a461110cedce71a0.pdf.

[23] Xin Jin, Ram Krishnan, and Ravi Sandhu. „A unified attribute-based access control model covering DAC, MAC and RBAC“. In: *IFIP Annual Conference on Data and Applications Security and Privacy*. Springer. 2012, pp. 41–55. URL: https://link.springer.com/content/pdf/10.1007/978-3-642-31540-4_4.pdf.

[24] Lalana Kagal, Tim Finin, and Anupam Joshi. „A policy language for a pervasive computing environment“. In: *Policies for Distributed Systems and Networks, 2003. Proceedings. POLICY 2003. IEEE 4th International Workshop on*. IEEE. 2003, pp. 63–74. URL: http://ieeexplore.ieee.org/abstract/document/1206958/.

[25] Michelle Kwasny et al. „Privacy and technology: folk definitions and perspectives“. In: *CHI'08 Extended Abstracts on Human Factors in Computing Systems*. ACM. 2008, pp. 3291–3296. URL: https://dl.acm.org/citation.cfm?id=1358846.

[26] Jakob Nielsen. *Usability engineering*. Elsevier, 1994. URL: https://www.nngroup.com/articles/response-times-3-important-limits/.

[27] Information Commissioner's Office. *Privacy regulators study finds Internet of Things shortfalls*. Sept. 2016. URL: https://ico.org.uk/about-the-ico/news-and-events/news-and-blogs/2016/09/privacy-regulators-study-finds-internet-of-things-shortfalls/.

[28] Marc-Oliver Pahl. „Distributed Smart Space Orchestration“.
PhD thesis. Technische Universität München, 2014.

[29] Ioannis Panagiotopoulos et al. „Proact: An ontology-based model of privacy policies in ambient intelligence environments“.
In: *Informatics (PCI), 2010 14th Panhellenic Conference on*. IEEE. 2010, pp. 124–129.
URL: http://ieeexplore.ieee.org/abstract/document/5600452/.

[30] Paul B Patrick. *System and method for dynamic data redaction*.
US Patent 7,748,027. 2010.
URL: https://patents.google.com/patent/US7748027B2/en.

[31] Maria Poveda Villalon et al. „A context ontology for mobile environments“.
In: (2010). URL: http://oa.upm.es/id/eprint/5414.

[32] Ravi Sandhu. „Attribute-Based Access Control Models and Beyond.“
In: *ASIACCS*. 2015, p. 677.
URL: http://www.profsandhu.com/miscppt/kth_abac_141029.pdf.

[33] Bill N Schilit et al.
„Challenge: Ubiquitous location-aware computing and the place lab initiative“.
In: *Proceedings of the 1st ACM international workshop on Wireless mobile applications and services on WLAN hotspots*. ACM. 2003, pp. 29–35.
URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.1013.1390&rep=rep1&type=pdf.

[34] Kalpana Shankar. „Pervasive computing and an aging populace: methodological challenges for understanding privacy implications“. In: *Journal of Information, Communication and Ethics in Society* 8.3 (2010), pp. 236–248. URL: http://www.emeraldinsight.com/doi/abs/10.1108/14779961011071051.

[35] Organization for the Advancement of Structured Information Standards (OASIS).
eXtensible Access Control Markup Language (XACML) Version 3.0. July 2010.
URL: http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-cd-04-en.html.

[36] Inc.2 Sun Microsystems. *A Brief Introduction to XACML*. Mar. 2003.
URL: https://www.oasis-open.org/committees/download.php/2713/Brief_Introduction_to_XACML.html.

[37] Karen P Tang et al. „Rethinking location sharing: exploring the implications of social-driven vs. purpose-driven location sharing“. In: *Proceedings of the 12th ACM international conference on Ubiquitous computing*. ACM. 2010, pp. 85–94.
URL: http://www.cs.cmu.edu/~jasonh/publications/ubicomp2010-socialsharing-final.pdf.

[38] Samuel D Warren and Louis D Brandeis. „The right to privacy“.
In: *Harvard law review* (1890), pp. 193–220.
URL: http://www.jstor.org/stable/1321160.

[39] Zachary Wemlinger and Lawrence Holder.
„The cose ontology: Bringing the semantic web to smart environments“.
In: *International Conference on Smart Homes and Health Telematics*.
Springer. 2011, pp. 205–209.
URL: https://link.springer.com/chapter/10.1007/978-3-642-21535-3_27.

[40] Alan F Westin. „Privacy and freedom“.
In: *Washington and Lee Law Review* 25.1 (1968), p. 166.
URL: https://scholarlycommons.law.wlu.edu/cgi/viewcontent.cgi?article=3659&context=wlulr.

[41] Juan Ye et al. „Ontology-based models in pervasive computing systems“.
In: *The Knowledge Engineering Review* 22.4 (2007), pp. 315–347.
URL: https://www.cambridge.org/core/journals/knowledge-engineering-review/article/ontologybased-models-in-pervasive-computing-systems/37531A29B7B354DB78400DA9E2A1CEEE.