A CONTEXT-AWARE ARCHITECTURE FOR SMART APPLICATIONS WITH ENABLED ADAPTATION AND REASONING CAPABILITIES

ZAKI CHAMMAA

A THESIS
IN
THE DEPARTMENT
OF
COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

MAY 2019

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: Zaki Chammaa

Entitled: A Context-Aware Architecture for Smart Applications with Enabled Adaptation and Reasoning Capabilities

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

_____ Chair
Dr. Sabine Bergler

_____ Examiner
Dr. Tristan Glatard

_____ Examiner
Dr. Juergen Rilling

_____ Co-supervisor
Dr. Vangalur Alagar

_____ Co-supervisor
Dr. Nematollaah Shiri

Approved by _____
Chair of Department or Graduate Program Director

_____
Dr. Amir Asif, Dean
Faculty of Engineering and Computer Science

Date _____

Abstract

A Context-Aware Architecture for Smart Applications with Enabled Adaptation and Reasoning Capabilities

Zaki Chammaa

The term “smart city” refers to an instrumented, interconnected, and intelligent city built by leveraging Information and Communication Technologies (ICT). In such a city, a combination of embedded hardware and software involving sensors, actuators, and a host of mobile devices and wearables that are connected to the Internet of Things (IoT) networks will sense data in different contexts and automatically drive desired adaptations through actuators. Through adaptations, city planners, professionals, and researchers aim to optimize resource consumption and cost of providing services while improving the quality of life for the ever increasing urban population. To fully realize this goal, a context-aware and data-centric inference is a necessity. A system is said to be context-aware if it is able to adapt its operations to the current context without explicit user intervention. This thesis proposes a generic context-aware system architecture for development of smart city applications. The proposed architecture puts special emphasis on privacy and security, incorporating mechanisms to protect the system and sensitive information at each layer of the architecture. Furthermore, this architecture integrates with a reasoning component, whose inference engine can be driven by logic or other formalisms. A prototype implementation and a case study done in this thesis indicate the practical merits of the proposed architecture and provide a proof of concept.

Acknowledgements

Many people have helped me over the past couple of years and I am truly grateful for their support. I would like to start by thanking my supervisors, Drs. Vangalur Alagar and Nematollaah Shiri, for their continuous support, guidance, and teaching throughout my Master studies. This work would not have been possible without them, and I will be eternally grateful for what they have done for me. I would also like to thank my parents for their love and support throughout my whole life, and without whom I would not be where I am today. Moreover, I would like to give a special thanks to my siblings for helping take my mind off of my work when I needed to relax. Last but not least, I would like to thank my wonderful wife Moushomee for being there for me throughout my studies. Her love and support were like a light in the darkness that kept me going during this challenging process.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

All around the world, more people migrate from rural areas to cities in hopes of finding better jobs, easier access to healthcare and an overall better quality of life. Consequently, cities are becoming overpopulated. In the US, the percentage of individuals living in urban areas rose significantly over the last two centuries, going from 5.1% in 1790 to over 75% in 1990 [ABD15], as shown in Figure 1. The US is not the only country with a rise in urban population. In fact, 2008 was the year where more than 50% of the world population (3.3 billion people) lived in cities. This number is expected to go up to 70% in 2050 as per UN estimates [CBN11b]. As a consequence of this migration, cities are consuming most of the energy (60%-80%) around the world, which results in unprecedented increase in pollution levels [ABD15]. As more people continue to migrate from rural areas to cities, government officials and city planners have to come up with efficient solutions to deal with challenges such as traffic management and transportation systems, waste management, healthcare, air pollution and overall energy consumption.

In response to this new phenomenon, researchers and professionals came up with the “smart city” concept. The term “smart city” can have a wide range of interpretations, so defining it can sometimes be a daunting task. The challenge in finding an exact definition for this concept is that it spans multiple disciplines. Many researchers try to define what a smart city is with respect to their area of research. When compiling the many definitions available in the literature, one can start to see a pattern developing and one can begin to understand what a smart city is and what the requirements are for a city to be considered smart.

Going back to its roots, the term “smart city” was first used in the 1990s. Back then, the interest was to find ways to leverage Information and Communication Technologies (ICTs) to make a city smart [AANC$^+$12]. Since then, many researchers and professionals have enriched the definition of smart cities, some distancing themselves from the technological aspects and focusing more on the social and urban aspect of it. In [NP11], the authors have tried to define what a smart city is by examining the "meaning of smartness" in the context of urban living. When comparing the

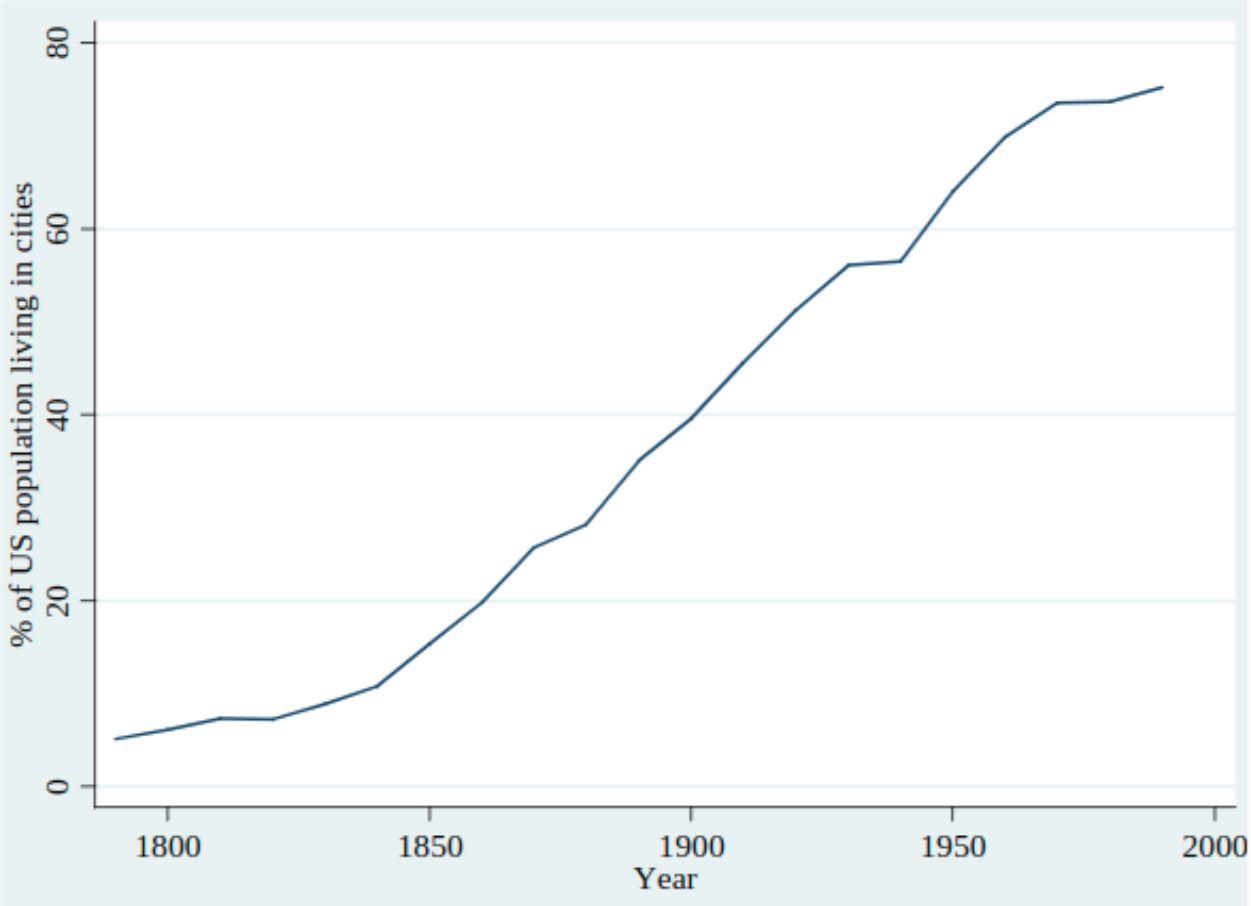

Figure 1: Percentage of US population living in urban areas 1790-1990 [CBN11a]

words "smart" and "intelligent", they found that the latter is usually restricted to having a quick mind and being responsive to feedback, whereas the former implies being intelligent and adapting to dynamically changing situations.

In [HEH+10], the authors define "smart city" as an "instrumented, interconnected and intelligent city." The term "instrumented" refers to "leveraging sensors, wearables, and other such devices to gather live real-world data and integrate them for future adaptations". The term "interconnected" refers to "the ability of having the gathered data available to city services through local and cyber networks". The term "intelligent" refers to "the ability to perform complex analytics on the gathered data in order to make informed decisions and provide timely services".

From a technological point of view, ICT plays an important role in the critical infrastructure components and services of a city [WSB+10]. ICT enables these services to have sufficient intelligence to adapt to users' constant demands [KK08]. A prime example of ICT role in smart city design is in the construction of "smart buildings" [GDB+13]. Smart building design integrates one or more wireless sensor network with actuators in different levels and locations of the building

in order to sense, monitor and regulate elements such as temperature, humidity and lighting under varying contextual constraints.

Another definition given in [CBN11b] states that a city is smart when investments in human and social capital and traditional (transport) and modern (ICT) communication infrastructure fuel sustainable economic growth and a high quality of life, with a wise management of natural resources through participatory governance. Furthermore, it is specified in [Che10] that smart cities take advantage of communications and sensor capabilities sewn into the cities' infrastructures to optimize electrical, transportation, and other logistical operations supporting daily life, thereby improving the quality of life for everyone.

From the many perspectives presented above, one can see that smart cities need to leverage sensors, actuators, and a host of mobile devices to sense live data and adapt to changes in the environment through actuators. As illustrated in Figure 2, smart city projects span across multiple application domains, some of which may need to collaborate in achieving the common goal of improving various aspects of the city as well as providing a better quality of life to its citizens. Interaction between devices are *context-dependent*, and collaboration between applications must be based on *reasoning* and *adaptation*. Motivated by this, in this thesis we pin the concept of smartness on *context-awareness* and *reasoning*, because adaptation is part of context-aware reactions.

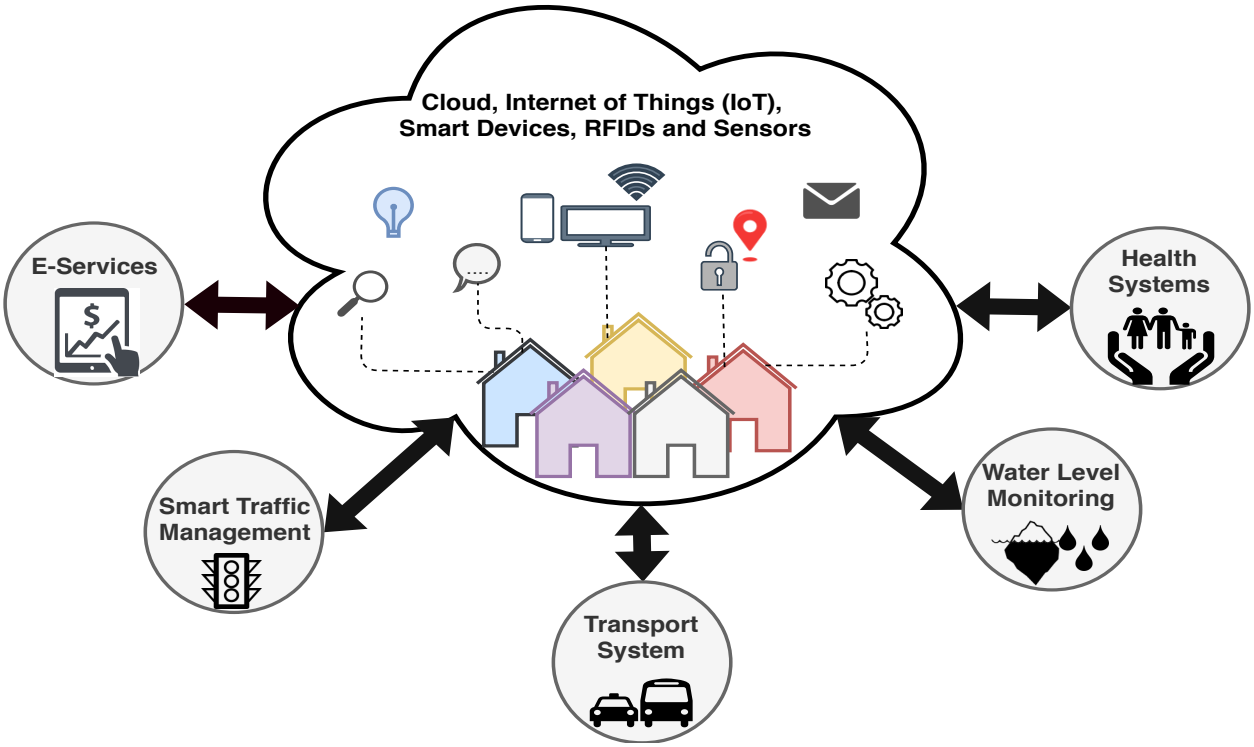

Figure 2: Smart City Applications

1.1 Context-Awareness and Reasoning

Now that the concept of "smart cities" has been explained, we describe the notions of context-awareness and reasoning in this section.

A system is said to be context-aware if it is able to adapt its operations to the current context without explicit user intervention, thereby increasing usability and effectiveness by taking environmental context into account [BDR07]. Research into context-aware systems has progressed significantly over the last few years, and this is due in part to advances in mobile computing and Internet of Things (IoT). The design and development of reliable context-aware systems is crucial to emerging smart city applications. The following are examples of different applications that use context-awareness.

- *Smart homes and buildings*: The type of data collected in buildings varies greatly depending on the type and application of the building, but generally includes security, sound, motion, video, temperature, humidity, smoke and gas concentrations, and electrical consumption. This data can improve response times to emergencies since decisions can be derived and sent to first responders in a timely manner. The data can also help monitor and optimize energy consumption by turning off lights and HVAC systems in areas where no one is present.

- *Mobile applications*: Mobile devices come equipped with many sensors such as location sensor, proximity sensor, ambient light sensor, and temperature sensor. It is important that programs and services react specifically to their current location, time and other environment attributes and adapt their behaviour according to the changing circumstances as context information changes rapidly in mobile computing [BDR07].

- *Critical systems*: Critical systems are highly reliable systems that have very low tolerance for failure. Having an effective context-aware system that can adapt and react in time is of the utmost importance for such systems since failures may have catastrophic consequences such as massive security breaches, loss of business or even death.

An important aspect of context-aware systems is reasoning. In this thesis, context reasoning is the process of deriving new facts based on existing facts, context information, and adaptation rules. This is an important feature of context-aware architectures for smart city applications since it supports the creation of new adaptations. The adaptation capability is crucial for smart systems in order to generate proper reactions that would directly affect the environment. In our proposed architecture, context reasoning will occur after context instances are built based on the sensor readings obtained from the environment. The design in our work is not tightly coupled to any one specific method of inference, allowing any rigorous methodology for context reasoning within the inference engine.

1.2 Thesis Contributions

The main contributions of this thesis are: (1) defining a set of comprehensive requirements to govern the design and implementation of a Wireless Sensor Network in smart city applications, (2) introducing a robust and generic architecture to support context-aware systems that incorporates privacy and security at every layer, (3) interfacing the proposed architecture with an inference engine, and (4) developing a running prototype system that shows the practical merits of the proposed architecture for smart city applications.

1.3 Thesis Outline

The rest of the thesis is organized as follows. The requirements for an intelligent wireless sensor network will be discussed in Chapter 2. In Chapter 3, we discuss some definitions of context, present the problem studied in this thesis, and review relevant context-aware architectures in the literature. Chapter 4 presents our context-aware architecture and compares it to those reviewed in Chapter 3. A detailed design will be presented in Chapter 5, and each component in the architecture will be described in more detail. Chapter 6 provides implementation details on the prototype system developed in this thesis. A case study on "Smart Room Configuration Control" is presented in Chapter 7, along with possible extensions to show the practical merits of the proposed architecture. Concluding remarks and a brief discussion on future work and challenges are provided in Chapter 8.

Chapter 2

A Background of Wireless Sensor Networks for Smart City Applications

In this chapter, we review the current status of sensor network technology for management of smart buildings. The goal is to identify the essential sensor types, and their fault tolerance, privacy and security requirements which need to be considered in the design of smart city applications such as smart buildings.

According to [Tra16], a smart building is a structure that uses sensors and actuators in order to collect data and manage it according to a set of rules and policies defined and required by the building management. The sensors and actuators in the building form a complex ecosystem of interconnected devices that belong to different groups such as external sensors, wearable sensors and indoor monitoring sensors. These sensors help in gathering useful data and activities from inside and outside the building. Within each group of sensors mentioned, there are different types of sensors. For instance, the indoor sensors group has motion detectors, light sensors, floor sensors, door sensors, humidity sensor, thermometers and window sensors. As for the wearable sensors group, it includes, for instance, smartphones and smart watches, and the external sensors group includes cameras and environmental sensors. When these sensors are networked it will enable the building to be smart, in the sense that it can automatically and effectively monitor the interior and exterior of the building continuously and prepare for appropriate reactions to meet the situations that can arise in all contexts.

To make the most out of the information provided through the sensors and to exploit it for best performance, one must organize and deploy these devices strategically throughout the environment. A solution that can be used in such a scenario is a Wireless Sensor Network. A Wireless Sensor Network (WSN) is a group of wirelessly connected nodes that sense data from their environment and transmit it to a sink node where it is aggregated and made available for further

processing. These nodes are generally equipped with a processor, a transceiver, a sensing unit and a power unit [OH13]. The main advantages of using WSNs is that they are (1) scalable, (2) capable of withstanding extreme weather conditions, and (3) suitable for remote locations such as mountains, oceans, and rural regions [HAAK+17].

2.1 Requirements

Deploying a WSN in a smart city application generally requires extensive design and planning in choosing the right sensor types and in constructing an efficient WSN. To ensure robustness and reliability of the WSN that is being deployed, the following requirements must be met.

2.1.1 Fault Tolerance

In order to be considered fault tolerant, the sensor network must be robust against node failure [PSC09]. In other words, the WSN should be able to perform its tasks as expected even when there are node failures. There are different reasons why sensor nodes may fail, and these include: lack of power, physical damage, hardware issues or environmental disruption [BTR15].

The goal of fault detection is to verify that the services being provided are functioning properly, and in some cases to predict if they will continue to function properly in the near future [BTR15]. In general, nodes in WSNs are prone to failures because of the reasons stated above, and this is why robust fault detection and recovery algorithms are absolutely necessary. This is particularly important for commercial buildings as so many day-to-day activities heavily depend on it. For instance, if the motion sensor that controls the lights in a particular room stops working and the lights stay on unnecessarily, it would result in increased energy costs. In more serious cases, if a camera or an electronic lock becomes defective, it can lead to a security breach that could result in unauthorized access to restricted areas of the building.

There are different methods to detect node failures in a WSN. In [BTR15], the authors propose three ways to detect faults in the network. The first method to determine if there are node failures is through self-diagnosis, where a node would use the measurements of accelerometers to decide if there is a problem that could potentially result in hardware malfunction. An example of self-diagnosis is when the sensor hardware allows monitoring of the battery voltage. In this scenario, the battery life (time of failure) can be estimated by monitoring the discharge rate of the battery, allowing the replacement of the battery in a timely fashion to avoid any interruptions. The second method used in node failure detection is called group detection, which assumes that sensors from the same region have similar values. Therefore, measurements from neighbours of a node are taken and the result is used to calculate the probability that the node is faulty. The third fault detection

method proposed in [BTR15] is called hierarchical detection, which uses a detection tree in order to make the fault detection method scalable. The fault detection results at a node are forwarded to the parent nodes, which aggregate the results from all the child nodes and forward them to the sink node. Although this method works well with any network size, it consumes more resources from the network.

There are different solutions to ensure that WSNs remain functional in the event of node failures. For instance, in [BTR15], they propose two recovery techniques to ensure fault tolerance. Active replication is a popular solution that can be used in cases where many nodes provide the same functionality. An example of that is a scenario where environmental data sensors are deployed throughout a city, and nodes are placed in each district. If the data of neighbouring districts is aggregated at the base station, node failures will not affect much the aggregated data, hence allowing the system to remain functional. The second method is called passive replication, in which the requests are sent to the primary replica for processing. Three steps are required for the passive replica method: fault detection, primary selection, and service distribution.

In the case of a commercial smart building, it is very important to have a good fault tolerance mechanism in place. Generally, the size of the network can be estimated before deployment and is unlikely to change much afterwards. For this reason, scalability of the fault tolerance mechanism is not necessary. For this particular application, a mix of self-diagnosis and group detection can be used to identify failures in the network. Generally, a sensor within a smart building uses electricity rather than battery as it uses power from the building itself. Therefore, hardware and software failures would be the most common reasons for failure in this application. For cases where the sensor stops working completely, a "heartbeat" can be sent to the base station from each sensor, and when the base station stops receiving a heartbeat from a particular sensor, it detects that this sensor has stopped working. For the case where the sensor still works but not as expected, a correlation between the data sent by similar neighbour sensors can be computed to determine validity of each sensor in that group. By combining the above two methods, one can try to minimize the risk of undetected faults in the network.

As for the recovery mechanism, several solutions can be used in smart buildings depending on the sensors, their locations, and their importance. For instance, a thermometer in an employee's office would not be treated the same way as an electronic lock guarding a restricted area. For this reason, a scale of importance should be implemented and each sensor should be assigned a number that determines its importance. For less critical sensors, active replication can be used, and data from similar neighbouring sensors can be aggregated to estimate the missing data from the faulty sensor node. However, for more critical sensors, there should be some sort of redundancy such as a back-up sensor ready to take its place. The back-up sensor can either be an idle sensor on stand-by or an existing sensor in the network that can perform the tasks of the faulty sensor until it is fixed

and back in-use.

2.1.2 Privacy

According to the International Association of Privacy Professionals (IAPP), information privacy is the right of an individual in having some control over how her personal information is collected and used. Often, people think that privacy and security are the same thing. While the two concepts share some aspects, they are different. Data privacy is focused on the use and governance of personal data, with guidelines involving the definition of policies to ensure that users' personal information is being collected, used and shared in appropriate ways. Security on the other hand focuses more on protecting data from malicious attacks and the exploitation of stolen data for profit. While security is an important element in data protection, it is not enough in general for addressing privacy concerns [oPP]. However, for a system to be considered "privacy-preserving", the security and safety mechanisms in the system must fulfill the privacy requirements of all the actors in the system [PAW17].

Privacy is an important feature when it comes to WSNs in smart buildings because of the nature of the information collected and stored by the sensors. This information includes, but is not restricted to, user identities, user locations, user activities and interactions with other users. The collection of such information may sometimes be essential to ensure convenience and security, but it also raises some privacy concerns for users of the building. For instance, Wi-Fi connections are usually monitored and information relating to that connection is logged. This information includes the device's MAC address, the connection access point as well as the time stamp. Such information can be used to detect the actual location of a user [PDY+17]. Privacy policies must be put in place to avoid unauthorized access to data and restrict access and use of the device to authorized users only.

In [PDY+17], the authors describe two sets of rules that govern smart building management systems (BMS). The first set of rules includes building policies, which are requirements that ensure that the building functions properly from a management point of view. For example, the BMS may store the identities and locations of all residents, and the special needs of physically challenged elders to prepare for emergency evacuation. Another example of a building policy is to require that the residents present some identification to access common areas and rooms in the building. These policies will be used to enforce proper installation and functionality of the sensors deployed in the building. This means that residents in the building are willing to share a minimum amount of their personal information with the management, while the management assures the residents to keep that information secure.

The second set of rules presented in [PDY+17] is concerned with user preferences. These

rules ensure that users have some measure of control over their private information. For instance, a user may specify that their information must not be shared with anyone outside the building management. Another user may allow their activity data to be shared with a third-party, as long as their personal information is removed from the data. Additionally, users should know exactly how their data is handled by the BMS. They should also be allowed to modify their preferences at any time. For example, a resident may not want another resident to know the apartment in which she lives, the rent she pays, and her personal phone number. However, a resident may allow her name to be displayed on the occupants list at the building entrance, which is associated with a button that is linked to her TV screen. Thus, unintrusively the resident gets the right to know who is asking her permission to visit her in the building, while the visitor will have the limited privilege to share a code to the apartment.

In the architecture presented in this thesis, the data collected is made available to the residents of the building so that they can view what information is being collected by the sensors. Users should be able to specify what information is collected by sensors, so long as it does not pose any safety or security threats to the building. In the case that a specific type of information is required by the BMS, the user should be made aware that this information will be collected. However, the user should still have some control over how this information is used or shared, and should be allowed to refuse sharing of this information with anyone outside of building management.

As technology becomes cheaper and more compact, privacy concerns will become even more important as it will be extremely easy for an individual or an organization to spy on people. Technology alone will not be able to solve privacy problems. In fact, social awareness and new laws and regulations are extremely important to ensure a certain level of privacy in WSNs [PSW04]. Governments should issue privacy policies and regulations in order to minimize such behaviour, and people found guilty of breaking these regulations should be prosecuted. Furthermore, transparency from organizations (building management in this case) will increase the public trust in pervasive computing and allow growth of such technology. Only by following these guidelines will we get closer to achieving harmony between the public and the fast evolving technology.

2.1.3 Security

Wireless network security consists of designing and implementing solutions to protect systems and information within a network. The implemented measures are there to protect both hardware and software from attacks. Furthermore, network security also ensures proper access control inside and outside the network. A good security strategy for WSNs ensures protection against a variety of internal, external, and system threats. Examples of internal threats include, for instance, weak access control, privilege abuse, and data exfiltration. Types of external threats include Denial-

of-Service (DOS), eavesdropping, data breaches, and malware. Finally, the list of system threats includes hardware failure, software failure, and power failure.

WSNs present interesting security challenges because node failures are fairly common in WSNs, and these failures become vulnerabilities that can be used by a malicious individual. For this reason, security is an important requirement in WSN systems because of possible consequences faced in case of a compromise. For example, if an attacker performs a DOS attack on a portion of the network, all the sensors in an area would either stop working or become extremely slow. A situation like this would allow an intruder to enter the building area without being noticed. This can have very negative consequences as the intruder can steal, vandalize, or hurt someone. In order to maximize security in a system, it must be integrated in each component of that system. Failing to do so would result in non-secure components that can become points of vulnerability and attack to the system [PSW04]. In the case of a WSN, every sensor node should be secure to prevent unauthorized access and malicious attacks that would compromise the entire system [AHSC12].

In order for a network to be considered secure, security properties such as access control, confidentiality, data integrity and availability must be incorporated at each node. These properties are described in more details in the following subsections.

Access Control

Access control is an important security requirement for WSNs because it ensures protection of sensitive data from unauthorized access. The basic idea of access control is to put in place mechanisms that allow access to resources only with correct credentials and refuse access otherwise. In this thesis, we will be using a context-aware role-based access control (CA-RBAC), which is a variant of the role-based access control (RBAC) model.

The main idea of the RBAC model is that each role has a set of associated permissions, and any user associated with that role will have those permissions. Accordingly, if a role is removed from a user, that user will lose all the permissions associated with that role. Also, if the permissions are modified on the role itself, it will affect every user attached to that role. This method of access control is very useful for WSNs because of its simplicity, but it has some limitations since it can only allow or deny access to a resource. For the WSN presented in this thesis, we may need more flexibility for access control.

Context-aware Role-based Access Control (CA-RBAC) model is discussed in [GMW10b, APW17]. In this method, context is combined with the RBAC model in order to control access to resources in the network. The goal of this model is to enrich the RBAC model by making it context-aware. This way, access to a resource is not limited to allowing or denying access to it, and may be differ-

ent depending on the changing context. In [GMW10a], the authors argue that the RBAC model is not good enough to provide a complete access control solution for WSN. They define three context situations that can affect access control decisions in the CA-RBAC model, namely critical, emergency, and normal. In our example of the smart commercial building, access to camera footage in normal context is only granted to members of building management with a high clearance. If a suspicious individual enters the building and the context changes to critical, then access to camera footage can also be granted to the authorities to ensure safety of the individuals inside the building. The incorporation of context to the RBAC model will make it more flexible, and will ensure that the right access is given depending on the situation.

Confidentiality

Data confidentiality ensures that information is only accessed by authorized users. One method of achieving this is through access control, which has been discussed in the previous subsection. Another method that is used to ensure confidentiality in WSNs is encryption. There are two types of encryption: asymmetric and symmetric. In asymmetric encryption, two keys are used, one for encryption and one for decryption. In this type of encryption, a unique private key can decrypt a message encrypted using the corresponding public key [MCM14]. As for symmetric encryption, only one secret key is used for both encryption and decryption when there is a communication between two entities, with the key being known only to those two entities [MCM14].

When these two methods are compared in the scope of WSNs, symmetric encryption is usually the one that is more suitable for devices like sensor nodes because of its low overhead [MCM14]. Generally, asymmetric encryption is not suitable for WSNs because of the big size of the code, message and data, the long processing time, and the high power consumption [MCM14]. In this thesis, we will use the access control algorithm presented in the previous subsection in order to ensure confidentiality.

Data Integrity

Data integrity is the process of ensuring that data is unchanged by either an attacker or by unintentional damage and that compromised data is detected by the system. In this thesis, data integrity will be ensured using the access control method mentioned above. Basically, the data will be protected from damage by the access control policies defined in the system.

Availability

Although fault tolerance ensures that nodes in a WSN are available at all times, there are attacks such as Denial-of-Service that can target nodes in a network and make them unavailable for long

periods of time if no measures are put in place to detect and prevent such attacks. A very popular solution that is used against these attacks is intrusion detection. This method will be used in the WSN proposed in this thesis along with the fault tolerance methods described in a previous subsection to ensure the availability of the nodes at all times.

Generally, intrusion detection systems have two components to detect intrusions and attacks. The first component performs signature based intrusion detection, using a compiled database of known threats on WSNs to detect attacks. This method is very effective against known attacks, but will generally not detect zero-day threats. The second component performs a behaviour based intrusion detection. In order to do so, a training period ranging from a few days to a few weeks is necessary. During that period, the intrusion detection system learns the normal behaviour of the network and builds a model using a machine learning algorithm. After the training period is complete, all the incoming and internal traffic in the WSN is compared to the normal behaviour of the network, and alerts are raised when suspicious activities are detected. The training of the model needs to be performed on a regular interval in order to account for changes in the network.

In this thesis, we only use the signature based method because the behaviour based method requires the design and implementation of an efficient machine learning algorithm, which is out of the scope of this work. Furthermore, behaviour based models are computationally expensive to build and optimize.

2.2 Summary

In this chapter, we introduced the requirements for the Wireless Sensor Network that will be used in our smart city application. Features such as fault tolerance, privacy, and security were discussed, and existing solutions were studied to address each of those features. In the next chapter, we will present a literature review of existing context-aware architectures in order to subsequently showcase our contributions.

Chapter 3

Context-Aware Systems

In this chapter, context-aware systems will be discussed in more detail. In Section 3.1, different perspectives of context notion are presented. The problem that is studied in this thesis is presented in Section 3.2. We then review existing context-aware architectures in Section 3.3, and highlight their advantages and disadvantages.

3.1 Definition of Context

The notion of “context” is crucial to describe, specify, and evaluate context-aware systems. For this reason, many researchers [ADB+99, Dey01, Wan06] have attempted to provide a working definition of context, even though it is a rich concept to define. The Oxford English Dictionary defines context as "the circumstances that form the setting for an event, statement, or idea, and in terms of which it can be fully understood." In this definition, circumstances can be information such as the time of day, the day of week, the location or the weather outside. If we know that an employee is at home at 2 AM on Tuesday, we may infer that this person is resting or sleeping. On the other hand, if that individual is on the road at 7 AM on Wednesday, we may infer that this person is on their way to work.

Context has been defined and used in different fields, including computer science. The authors in [HNBR97] describe context as "the aspects of the current situation". This definition is a simple one, and basically states that context is any information found in a specific situation. Another definition provided in [ADB+99] states that context is "any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves." In this definition, the authors extend the definition of context by adding the interaction between the user and the application. In fact, by considering only the information of all entities that are

relevant to the interaction between the user and the application, one can get a much better picture of the situation.

In the dataflow programming language Lucid [WA85], context cannot be defined although it is inherently assumed. In order to improve the expressive power of Lucid and make it usable in different context-aware applications, [Wan06] formalized context as a relation over "dimension, tag" pairs and gave semantics for its use in intensional programming. Recently, [AAS18] has given a comprehensive review of context literature to show the richness of context concept and proposed a generic formal definition of it. Although many different researchers have used context only with ad-hoc notations, we follow the notation introduced in [Wan06, AAS18] for smart city applications because reasoned inference requires well-defined syntax and semantics for contexts.

3.2 Problem Statement

Over the last two decades, research in the area of context-aware computing has progressed significantly. This research includes presentation of architectures and frameworks that define and exploit contexts in different application systems. A context-aware system can be defined as a real-time reactive system that senses its environment round the clock, adapting and reacting to changes without explicit user intervention. Generally, context-aware systems leverage sensors deployed in the environment to gather monitoring data. The data is then processed, which generates adaptations and reactions that are sent to actuators, which in turn perform actions that directly affect the environment.

While context-aware systems are important for smart city applications, their implementations bring new research challenges. Often, context-aware systems are designed and implemented with only one application in mind. The main advantage of this type of solution is that the architecture will be specific to that particular application, which means it will not be usable for other applications. When discussing smart cities, the need for a generic architecture that can be used for multiple applications becomes important. Having such an architecture would simplify the development and maintenance of applications, enabling the advancement of smart cities. Furthermore, smart city applications often generate a large amount of unstructured data from multiple sources. This brings about challenges for efficient and scalable data acquisition, processing and storage because many of these applications must react in real-time. Finally, many context-aware systems ignore the notion of privacy and security. This is a weakness when dealing with contextual data in a smart city application because a lot of useful information collected by sensors is sensitive by nature.

The next section will elaborate on existing context-aware framework solutions in order to motivate the context-aware architecture for smart city applications developed for this thesis.

3.3 Context-Aware Framework Solutions

In this section, we review some published work related to solutions for context-aware computing and systems. Each solution will be analyzed with respect to architecture, context model, method of reasoning, security mechanisms and privacy policies.

3.3.1 Context-Awareness Sub-Structure (CASS)

The context-aware system [FC04], called Context-Awareness Sub-Structure (CASS), discusses a centralized middleware aimed mainly at mobile applications. Figure 3 shows a high-level architecture of the CASS system. As can be seen from this figure, there are three main components that make up the CASS architecture, namely a sensor node, the CASS middleware, and a hand-held computing device. Here, sensor nodes are computers with one or more sensors attached to them. These nodes sense and collect data from the environment and send it to the middleware. The CASS middleware receives this data, stores the context, and uses the data to infer new information. Finally, the hand-held computer is the device used to run applications that use the CASS middleware.

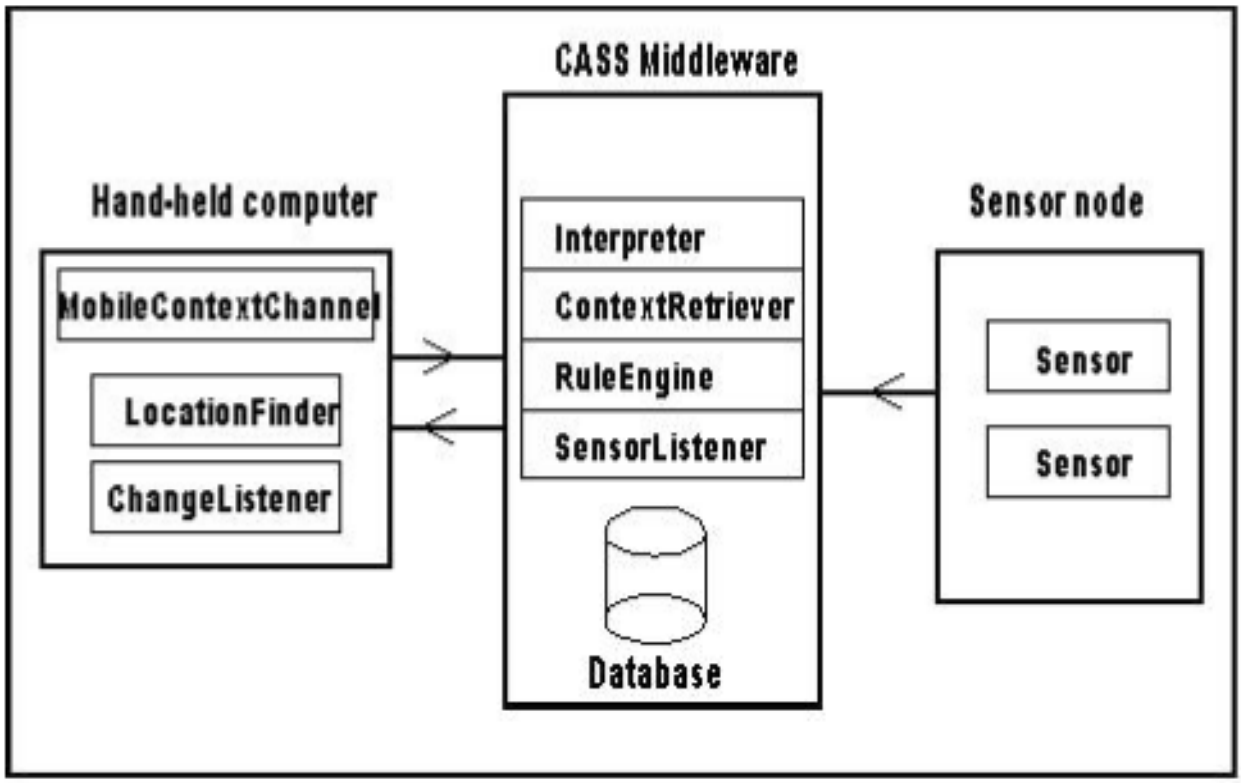

Figure 3: CASS Architecture [FC04]

The middleware is the main component of this architecture, and contains the following four main classes: an Interpreter, a ContextRetriever, a RuleEngine, and a SensorListener. The SensorListener class listens for new data from sensors and stores them as context information in a relational database. The ContextRetriever class fetches the stored context when it is needed. The Interpreter class is used to transform data into a format expected by a class. Both the SensorListener and ContextRetriever classes can use the Interpreter class. The RuleEngine class is central to the design of the CASS middleware. This class uses the rules in the knowledge base in collaboration with an inference engine to find a matching goal or goals when a change in context is detected. In CASS, the knowledge base is separated from the inference engine for two reasons. First, it makes it possible to represent knowledge in a natural way, making it easier for domain experts to express rules. Second, this separation makes it simpler and more flexible to change the knowledge base without affecting the inference engine.

In CASS, contexts are stored in a relational database, as the authors argue that such databases are good for handling large volumes of data. Another advantage of storing context in a relational database is that all data interactions will be made in the Structured Query Language (SQL), which is the most widely used language to deal with relational data. While relational databases are robust and mature, they do not perform as well with non-structured data. In smart city applications, the number of sensors in the physical environment could vary from a few hundred to hundreds of thousand sensors, making scalability of the database an important challenge. Furthermore, the variety of the sensors deployed in such applications makes it hard to predict the structure of the data. Therefore, using a relational database may not be suitable for a context-aware architecture in a smart city application.

One of the shortcomings of the CASS architecture is that it does not address privacy and security issues. This is an important aspect of context-aware systems in smart city applications which needs to be looked into in order to protect personal and sensitive user information.

3.3.2 Context Broker Architecture (CoBrA)

Figure 4 shows the agent-based context-aware architecture [Che04], called Context Broker Architecture (CoBrA). This architecture is aimed at "intelligent spaces", which are physical spaces such as offices, bedrooms, and vehicles that contain devices that offer users pervasive computing services [Che04]. As can be seen in Figure 4, the context broker is the heart of the CoBrA architecture, since it provides a centralized model of context for all components in the space, obtains the context from the devices and agents, reasons over the acquired context, and protects privacy by enforcing user defined policies.

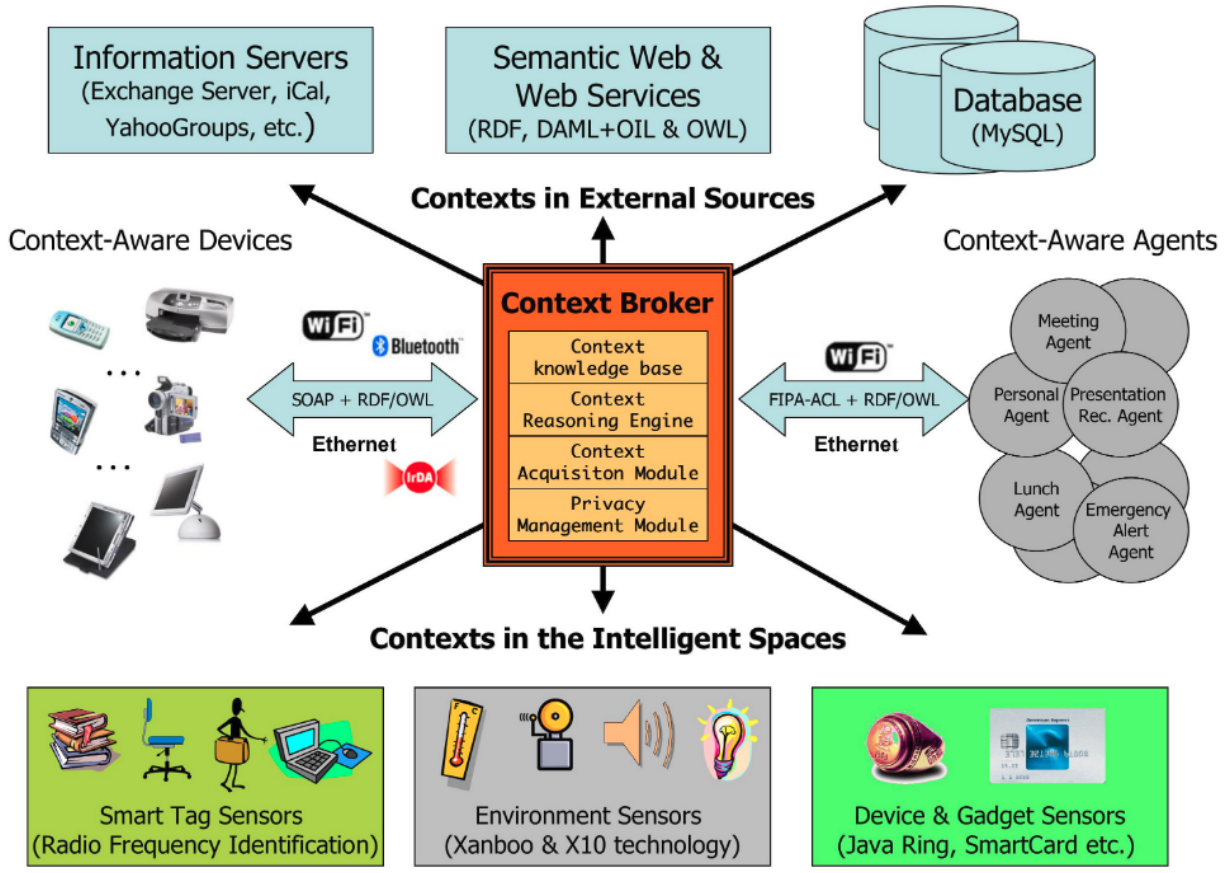

Figure 4: CoBrA Architecture [Che04]

The context broker contains four components, which are the context knowledge base, the context reasoning engine, the context acquisition module, and the privacy management module. The context knowledge base manages the storage of the knowledge within the context broker. This knowledge includes the ontologies describing various types of contexts, the ontology instance data of the acquired context, and the metadata for describing the storage structure of the represented knowledge. The context reasoning engine is a logical inference engine for reasoning over the acquired context. This engine interprets context based on the collected data, aggregating the contextual information from multiple sources using ontologies and domain heuristics and detecting and resolving inconsistencies within the acquired information. The main purpose of the context acquisition module is to obtain contextual information from sensors, agents, and the Web. It is designed in a way to improve reusability of the context sensing procedures. This is achieved by hiding the low-level context sensing implementations from the high-level functional components. The privacy management module manages the users' defined privacy policies and controls sharing of their private information. It is responsible for enforcing these policies when the context broker performs actions on user information.

CoBrA uses the Web Ontology Language (OWL) for context modeling, context reasoning, and knowledge sharing. The OWL language is a Semantic Web language that is used by computer applications that need to process the content of information instead of just presenting information to humans [MvH]. This language is a knowledge representation language for defining and instantiating ontologies. An ontology is a formal explicit description of classes (concepts) in a domain of discourse, with the properties of each class describing various features and attributes of the class and providing role restrictions [NM01]. CoBrA uses its own OWL-based ontology approach called COBRA-ONT. An example of this ontology can be seen in Figure 5.

```
<loc:LocationContext>
  <rdf:type rdf:resource="&tme;InstantThing"/>
  <loc:locationContextOf>
    <per:Person>
      <per:name rdf:datatype="&xsd;string">
        Harry Chen
      </per:name>
    </per:Person>
  </loc:locationContextOf>
  <loc:boundedWithin rdf:resource="&ebgeo;Japan"/>
  <tme:at rdf:datatype="&xsd;dateTime">
      2004-02-23T11:23:00
  </tme:at>
</loc:LocationContext>
```

Figure 5: COBRA-ONT Example [Che04]

Because of the nature of the data found in pervasive computing, the context broker within the CoBrA architecture has the responsibility to enforce policies to protect this sensitive information. CoBrA adopts a policy-based approach to protect privacy. According to the author [Che04], a policy is a set of rules that is specified by a user or a computing entity to restrict or guide the execution of actions. For instance, a user may create a policy to not allow sharing of his personal information. In CoBrA, policies are defined using the SOUPA policy ontology [CPFJ04]. The SOUPA ontology is a standard ontology for supporting pervasive and ubiquitous computing applications.

Using the SOUPA policy ontology, vocabularies for privacy and security policies are defined. These vocabularies are based on the Rei policy language [KFJ03]. With SOUPA, CoBrA enables a user to control the type and amount of information that the context broker can share with other agents. The author chose the SOUPA policy ontology because it supports policy reasoning using

description logic. Even though the SOUPA policy ontology has many advantages, many trade-offs in policy expressiveness arise. In fact, the ontology does not support the definition of logical expressions in the form of variables and rule conditions, resulting in users not being able to define conditional policy rules.

3.3.3 Context Toolkit

In [SDA99], the authors introduced a context toolkit that uses the concept of "context widget", based on GUI widgets. The authors justify this choice by arguing that GUI widgets have three main benefits. First, these widgets hide specifics of physical interaction devices from the application programmers. This way, those devices can change with minimal impact on applications. The second benefit of GUI widgets is that they manage details of the interaction to provide applications with relevant results of user actions. Finally, they provide reusable building blocks of presentation to be defined only once. This means that these blocks can be reused, combined, and tailored for use in many applications.

Similarly, a context widget is a software component that provides applications with access to context information from their operating environment. Just as GUI widgets hide implementation details and shield applications from the back-end, context widgets protect the application from context acquisition concerns. Some of the advantages of context widgets described are as follows:

- Add a level of abstraction between the sensors and the application. In other words, the application should not care if the presence of individuals in a physical space was detected by a motion sensor, a camera, or a combination of both.

- Filter contextual information to provide applications with what they need. For instance, a widget that tracks the location of a user within a building will only send context information to the application if the user changes rooms or leaves the building. That way, the data sent by the sensors can be reduced by a great amount, increasing the performance of the system without affecting its accuracy.

- Provide reusable and customizable building blocks of context sensing. A widget that monitors air quality can be used by many applications such as buildings, chemical plants, or parks.

As shown in Figure 6, the Context Toolkit contains the components BaseObject, Widgets, Services, Aggregators, Discoverer and Interpreter. The BaseObject class is a superclass which offers generic communication abilities to ease communication between components and simplify their creation. The Widget component is a reusable block that provides a level of abstraction between

the sensors and the application by hiding how context information is collected by the sensors. The Service component is a sub-component of Widget and it is responsible for dealing with actuators. The Aggregator is responsible for composing context of particular entities by subscribing to relevant widgets. As the name suggests, the Interpreter transforms raw collected context to something more meaningful. Finally, the Discoverer detects the components that are interested in any given context changes. In terms of context modeling, the Context Toolkit handles context in simple attribute-value-tuples, which are encoded using XML for transmission. According to the authors in [BDR07], the key-value model is the simplest method to represent context.

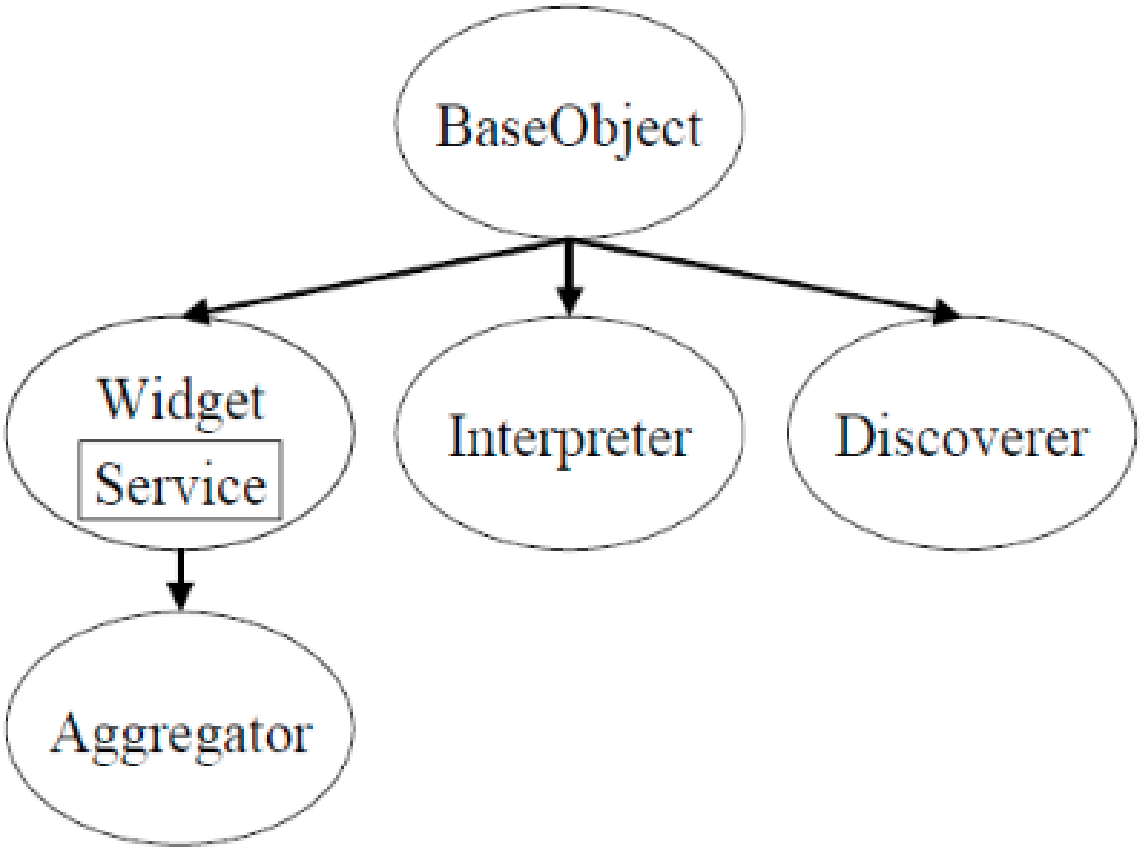

Figure 6: The Context Toolkit Architecture [SDA99]

In order to deal with the privacy issues found in context-aware systems, the Context Toolkit introduces the concept of context ownership. In this model, the user is the owner of the sensed context data relating to him, and has control over access to that context information. The components involved in that access control process are the Mediated Widgets, Owner Permissions, BaseObject and Authenticators. The role of the MediatedWidget component is to determine the owner of the sensed data. The Owner Permission component uses stored situations to allow or

deny permission queries that it receives. The stored situations include information such as authorized users and time of access. The BaseObject class contains an identification mechanism so that applications and components have to provide their identity when they initiate a request. The last component used for this access control mechanism is the Authenticator. This component uses a public-key infrastructure to validate the identity of the component sending the request.

3.3.4 Context-Aware Framework (CAF)

Figure 7 shows the architecture for the Context-Aware Framework (CAF) discussed in [Hna11]. This architecture contains four main modules, which are the Sensor Mechanism, Context Mechanism, Adaptation Mechanism, and Reactivity Mechanism. The Sensor Mechanism is responsible for receiving the collected sensor data, ensuring its validity and transforming it. Furthermore, this module is also responsible for aggregating data from multiple sensors and sending the latest contextual information to the Context Mechanism. The Context Mechanism is responsible for defining context and context situation, translating contexts and situations from different context theories and evaluating contexts and reasoning about situations. The Adaptation Mechanism is responsible for analyzing the collected knowledge about the environment and triggering the appropriate reactions. Finally, the Reactivity Mechanism is responsible for sending the reactions to the appropriate actuators.

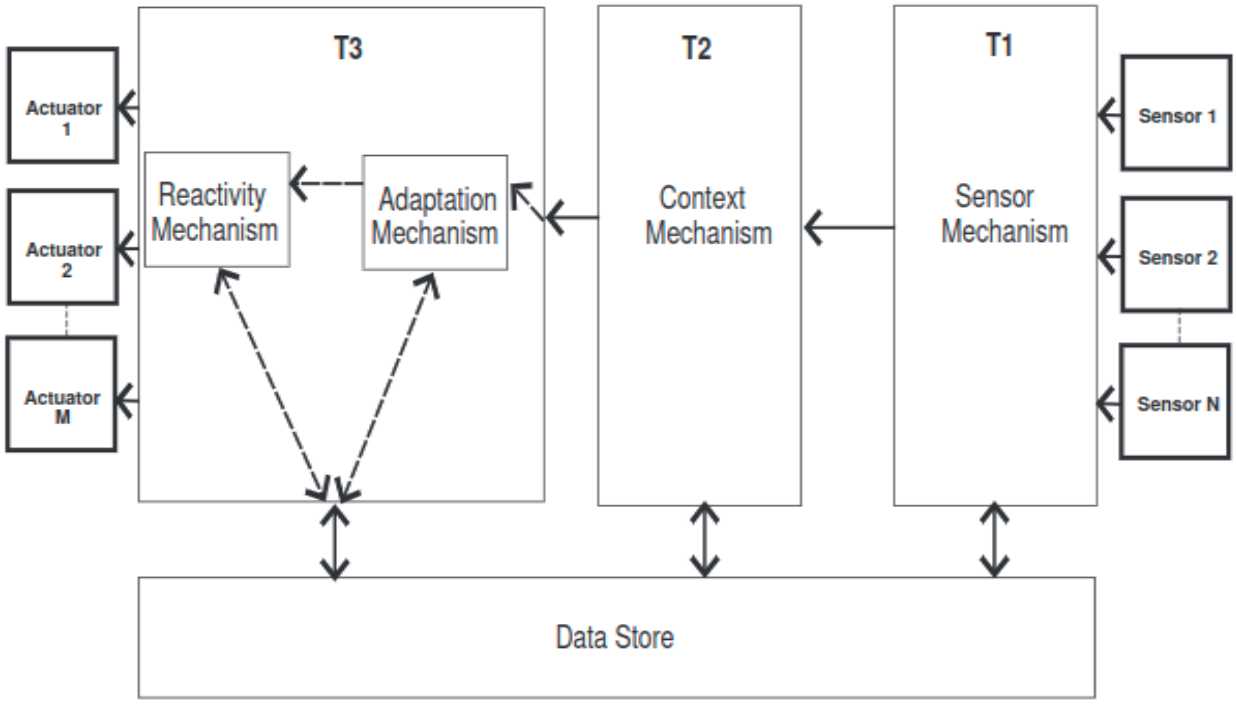

Figure 7: The Context-Aware Framework (CAF) Architecture [Hna11]

The concept of context situation introduced in their work represents semantic information based on context atomic properties. A context situation is a custom state that occurs when some predefined environment conditions are met. Situations are represented as expressions evaluated against contexts. The context is represented as a set of key-value pairs, where the key is the dimension and the value is the tag. In the scope of CAF, a situation is a state of interest to an application. For instance, Cold situation occurs when the temperature in degrees drops below a certain value.

As with the CASS architecture, CAF does not consider privacy and security issues. As mentioned above, these are important aspects of context-aware computing that should always be considered and incorporated in such systems because of the type of information processed. Furthermore, as can be seen in Figure 7, the author includes a data store in his architecture. The data store includes information such as associations between sensors and dimensions, situation expressions, associations between situations and adaptations, and associations between reactions and actuators. There is no mention of which schema is used for this data or how the database is managed. In the architecture proposed in this thesis, we define what type of information is stored in the data store since it plays a crucial role in the development of smart city application systems.

3.3.5 Agent-Based Context-Aware Architecture for a Smart Living Room

In [MEeAT16], the authors propose a multi-agent software architecture for building a smart living room. A smart living room is considered to be a smart space, which is "is able to acquire and apply knowledge about its environment and adapt to its inhabitants in order to improve their experience in that environment". Smart spaces must be capable of changing their behaviour dynamically based on users' activities and the environment. The authors tackle in depth the context-awareness aspect by focusing on context definition and modeling, and propose an architecture that contains the essential modules needed in a smart space.

Multi-agent systems are comprised of components called agents that are able to interact, usually by passing messages. The authors argue that using a multi-agent system is beneficial for their architecture because multi-agent systems offer a decentralized architecture while keeping the autonomy and proactivity of agents. They claim that such characteristics enhance the architecture modularity and fit requirements of appliances and equipment in smart spaces to provide adapted services to inhabitants in a proactive manner according to the current context.

As shown in Figure 8, there are three groups of agents in this architecture, namely the sensor multi-agent system, the core agent, and the actuator multi-agent system. The sensor central agent embeds two modules. These modules perform (1) context gathering, which re-assembles all received context data from sensor agents and builds the global current context vector, and (2) context interpretation, which has the task of interpreting each raw context data into useful information. In

this component of the architecture, there are some intelligent actions taken by the sensor central agent, namely in the context interpretation activities. That being said, the authors don't mention how the context interpretation happens.

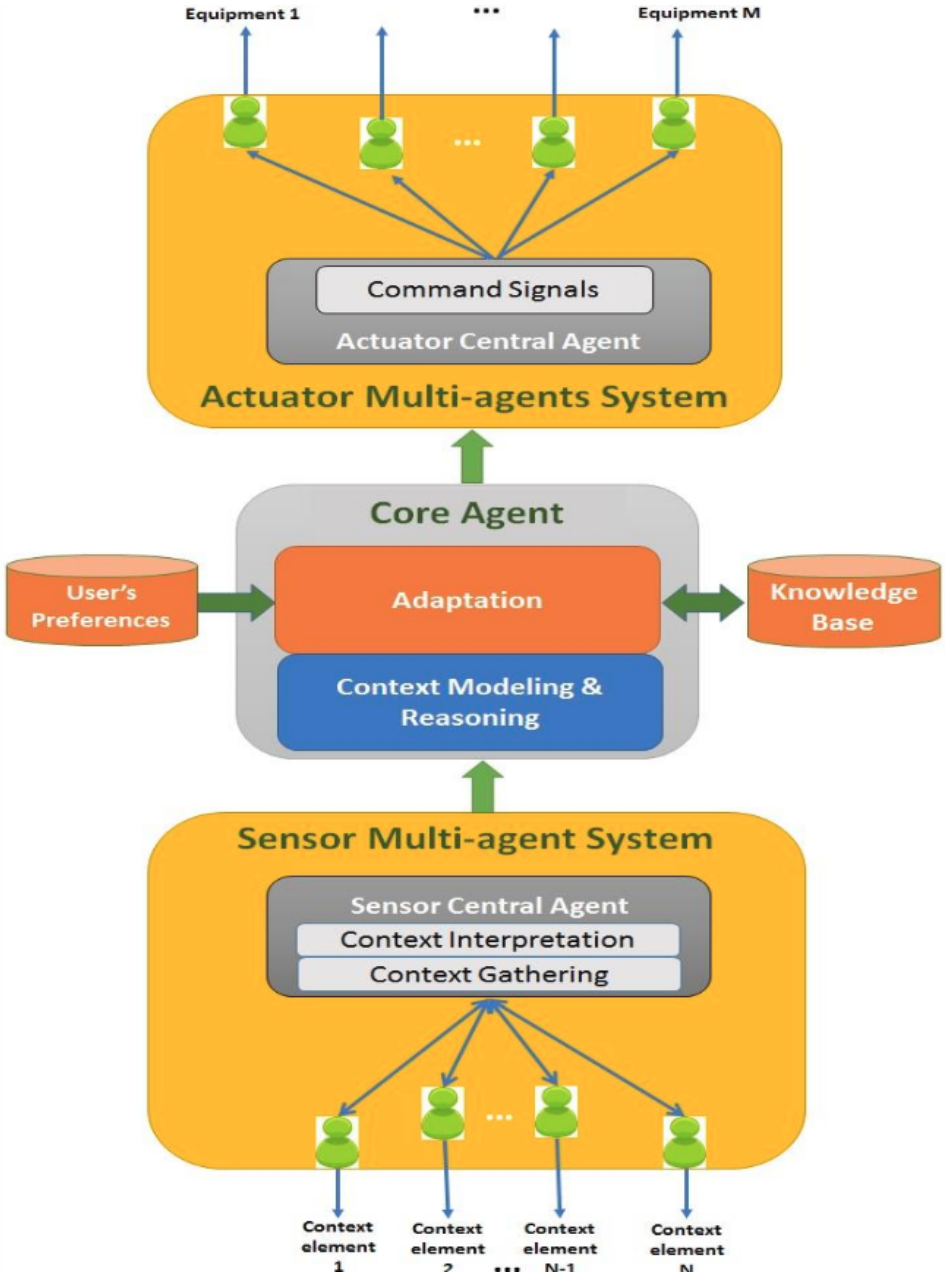

Figure 8: Agent-Based Architecture for the Smart Living Room [MEeAT16]

The second component is the core agent, which is the main component of the architecture. The core agent ensures context-awareness in the architecture and includes two modules: (1) context modeling and reasoning, and (2) context-aware services adaptation. Context modeling and reasoning are important aspects for achieving intelligence. The aim of context modeling is to provide an

abstraction of context information from the technical details of context sensing that allows an easy context management and more flexible sharing among devices and appliances.

The authors use ontology to model context, as illustrated in Figure 9. They give the following justifications for using ontology:

- Ontology is a powerful representation model for knowledge sharing, reuse, and expression of complex situations compared to other data modeling techniques.
- Ontology provides the possibility to perform logical reasoning. It permits the system to deduce relevant new contextual information that is not explicitly provided by sensors embedded in the living room.
- Ontology reasoning permits the system to check context inconsistency and conflicts caused by imperfect sensing of contextual information.
- Ontology reasoning allows the system to verify whether concepts are consistent.
- Ontology reasoning enables the system to find subsumption relationships between classes and instances.

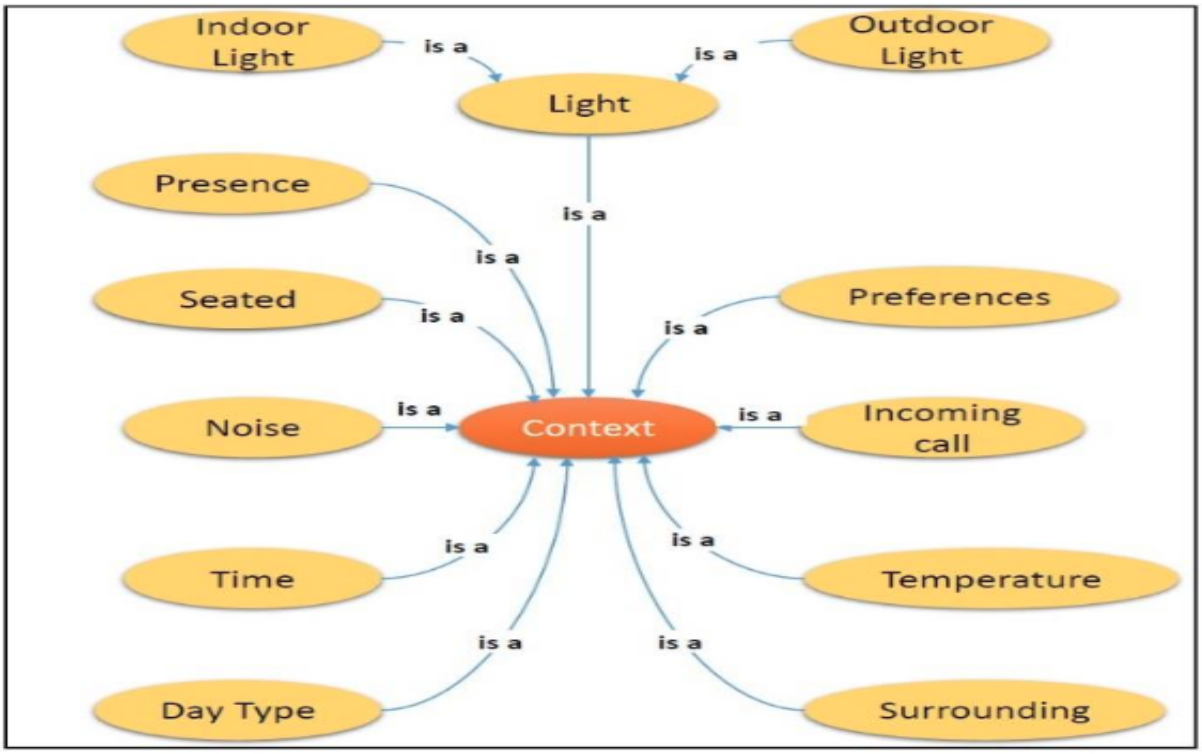

Figure 9: Ontology-Based Context Modeling [MEeAT16]

Though the authors mention the advantages of using ontology to model and reason with contexts, they provide no proof for these claims. Once context has been modeled and reasoned with, adaptations are generated based on the current context and the user preferences stored in the database.

The process of generating adaptations is an important aspect of intelligence, though the use case they provided is somewhat simplistic, described as follows. For each equipment, the user should provide the preferred form of service according to some context values. For example, the user could prefer to watch the news channel on his TV from 8 PM to 10 PM when he is alone. The authors mention that they use machine learning to achieve the adaptation task, but they do not provide any detail.

The final component is the actuator multi-agent system, which is composed of an actuator central agent (server) and a set of agents (clients), where each equipment of the smart living room is assigned to an agent. The actuator central agent contains a unique module, called command signals, that receives the service configuration vector from the adaptation module of the core agent and sends command signals to each equipment agent. This module stores the state of each actuator in a vector so as to minimize the communication needed. This can be considered an intelligent behaviour, since the module is optimizing its performance and preserving resources. Another intelligent aspect of this module is the fact that the command signal translates the adaptations to corresponding signals and chooses the appropriate devices to send those signals to. No details are provided to describe how this is done.

In addition to the three groups of agents discussed above, there are two databases present in this architecture, namely user's preferences and knowledge base. For the user's preferences database, the authors mention that the user must fill out his/her preferences before smart living operation. Although they provide examples of user preferences, they do not to mention which schema is used, which database is used, and how the data is managed and by whom. As for the knowledge base database, no details are provided in the paper, and it is mainly used as a black box.

Though the authors mention the possibility of having multiple users, there is no evidence in their work to support this claim. For instance, the authors mention that the devices found in the smart space should provide a set of services to the user or users in the smart space. However, they have not explained in their architecture and case study how different users can specify different preferences. In fact, it is assumed in the paper that preferences are from one user, while the other users are treated as triggers for some device events.

3.3.6 An Architecture for Interactive Context-Aware Applications

In [RSC07], the authors introduce a new architecture derived from the Model-View-Controller paradigm [KP88] that models physical and logical context in the front-end, to help users better understand application behaviour. They argue that by modeling context in the user interface, developers can represent the application's inferences visually for users.

In this architecture, the application domain is represented using a set of models. Some of these models represent physical objects, while others represent abstract data structures that the application uses. Using the MVC pattern as shown in Figure 10, the authors are able to represent a model in multiple ways by attaching multiple views. There is no report on implementation of these aspects in their work, but they say that it might be helpful for applications such as augmented reality (AR) goggles and projectors.

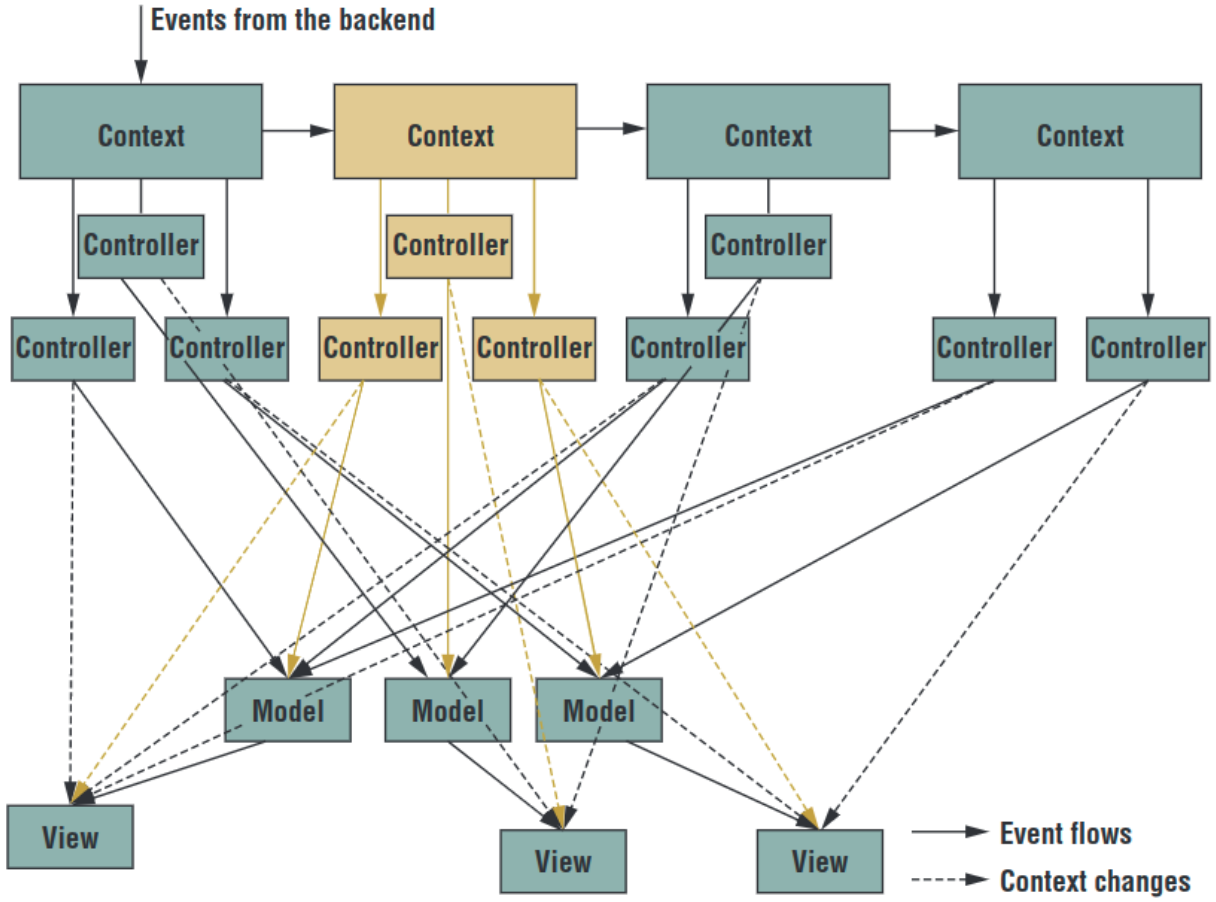

Figure 10: MVC Architecture [RSC07]

As shown in Figure 10, the architecture is designed in a way such that there is one controller per model for each context in the application. These controllers encapsulate how the model reacts to interaction in each context. The controllers are then attached to the corresponding context components. The key is that each controller is only activated when its context becomes active. Context handling is separated from interaction handling because many controllers are used. In doing so, the controllers will not contain context evaluation code.

In the system proposed in their paper, as shown in Figure 11, users interact with the application while performing their daily tasks. The application is responsive to the user's context and interprets all interactions against the current context. In other words, when a user changes context, the application changes its frame of reference in order to interpret the user's actions. There are two types of interactions that a user can perform, namely implicit and explicit interactions. Implicit interaction is an interaction not directly targeted at the application. For example, events detected by sensors are considered to be implicit interactions because the user is not changing the context directly. Explicit interactions happen when the user changes the context directly, with actions such as setting up a connection or sending media.

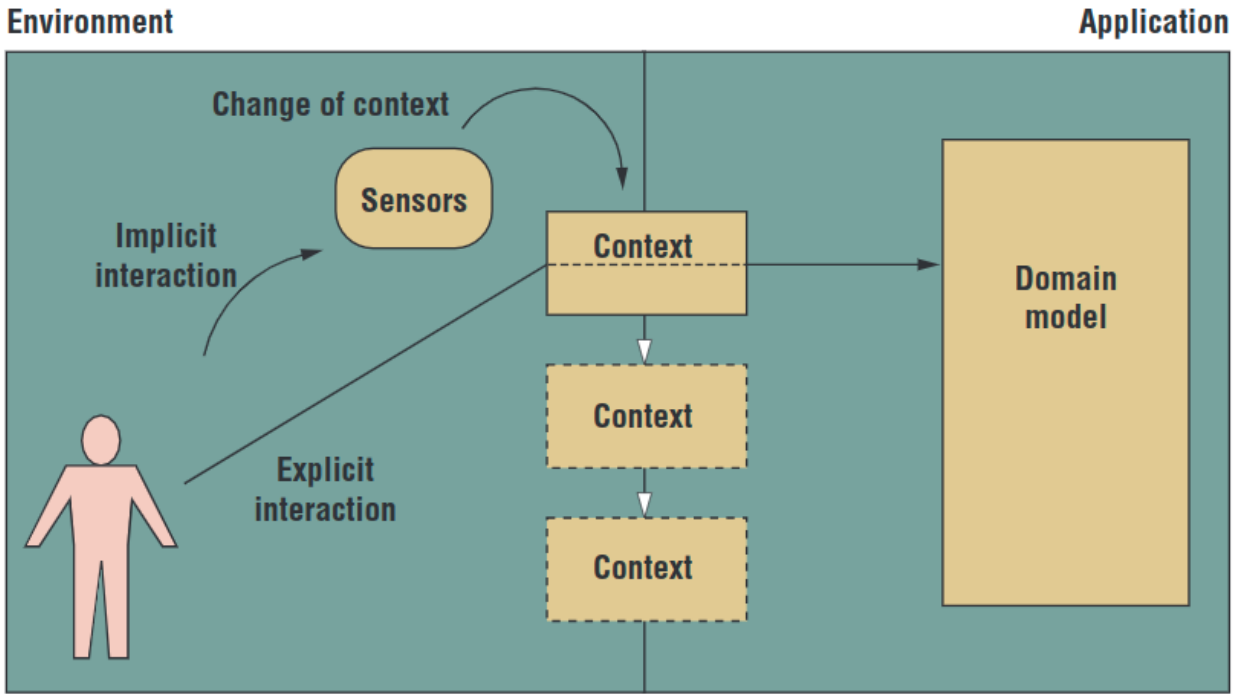

Figure 11: Interaction Model [RSC07]

When discussing context definition and modeling, the authors mention that they have only modeled the user context because according to them, real world objects don't have context. They claim that this is a phenomenological view of context. The phenomenological view emphasizes that peo-

ple create and maintain context during interactions, while the more popular positivist/engineering view is more concerned with representing context. In other words, the authors believe that humans perceive context as a property of interaction, rather than of objects or people.

The authors argue that one of the main benefits of their architecture is that it attempts to make the communication between the user and the application more intelligible. The authors present the following points to show how their architecture improves the communication between users and the application:

- *The designer's model is communicated*: MVC helps designers elicit the model they are using for the application by forcing them to specify the application using model objects. Second, it helps them communicate the model by requiring that view objects remain faithful representations of model objects at all times.

- *Context is not a piece of information, but rather a frame of reference*: In this architecture, context is used to interpret interactions. Each time the context changes, the framework switches the set of active controllers, whose function is to interpret user interaction.

- *The current context is always visible*: In the proposed framework, the controllers' activation and deactivation functions were designed to change the view when the context changes.

- *The system state is always visible and instant feedback is always provided*: All application entities are represented as models. Because each model has a view, the system state is always visible. Also, the system instantly updates the view as the system state changes.

As can be seen from the points above, the authors achieve their initial goal of improving the communication between users and the application in context-aware applications. However, there are several limitations in the proposed framework. The first limitation is that the system expects a context component for each context-variable combination to determine the contextual state. This can increase the number of context components exponentially as the number of context variables grows. It is difficult to predetermine all contexts that might arise in smart city applications, and we must have a way to model context dynamically. Another limitation is that it assumes only one user interaction. This is actually a limitation of the interaction model as the authors have looked at conversation as a one-to-one activity. This would not be realistic for smart city applications, in which we deal with many users. Finally, the paper does not address privacy and security issues found in context-aware systems. This is an important aspect of context-aware systems, particularly in smart city applications because of the nature of the information processed.

3.4 Summary

In this chapter, we reviewed the notion of context and context-aware computing. Then, the problem that is central to this thesis was explained, motivating the need for a generic, scalable architecture that takes into account privacy and security. Finally, a critical review of published work on context modeling and context-aware architecture was given.

Chapter 4

Context-Aware Architecture for Smart City Development

The goal of this chapter is to present a context-aware architecture for development of smart cities. The architecture has a number of components that interact through intelligent controllers which ensure secure sharing of data while enforcing privacy. The functionality of each component in the architecture is discussed in details, and its smart interactions with other components are explained. The architecture is compared to the other architectures discussed in Chapter 3 to establish its merits.

4.1 Proposed Architecture

In this section, our proposed architecture will be discussed in more details. The global architecture will be presented first, followed by a more detailed view of each component. Privacy and security issues and solutions will be discussed at each step of the architecture. Finally, we study and analyze the various aspects of the proposed architecture.

From the definitions presented in Chapter 1, smartness in cities can be seen as leveraging sensors and other input devices to gather live data pertaining to various applications, process it, and react with the inferred actions through actuators. In a software architecture, these steps can be divided into three logical steps, namely input, process, and output.

Figure 12 shows a high level view of the architecture of the context-aware system proposed in this thesis. There are four components in this architecture, namely the sensor component, context component, inference component, and adaptation component. In terms of smartness steps, the sensor component can be placed in the "input" step, the context and inference components in the "process" step, and the adaptation component in the "output" step. Furthermore, there are two data

stores available. The general data store is used to store sensor and actuator information as well as different policies. The type of information stored in the general data store must be general to all applications, as the name suggests. The other data store is the application data store. It stores facts, rules, and context information related to a specific application. Facts are generally static and can include information such as user information, user preferences, and privacy policies. Rules express predefined adaptation processes used together with context to determine the appropriate adaptations. Finally, the context information stored in the application data store is a dynamic entity which gets instantiated and affected by the data coming from the sensors and other input devices in the environment.

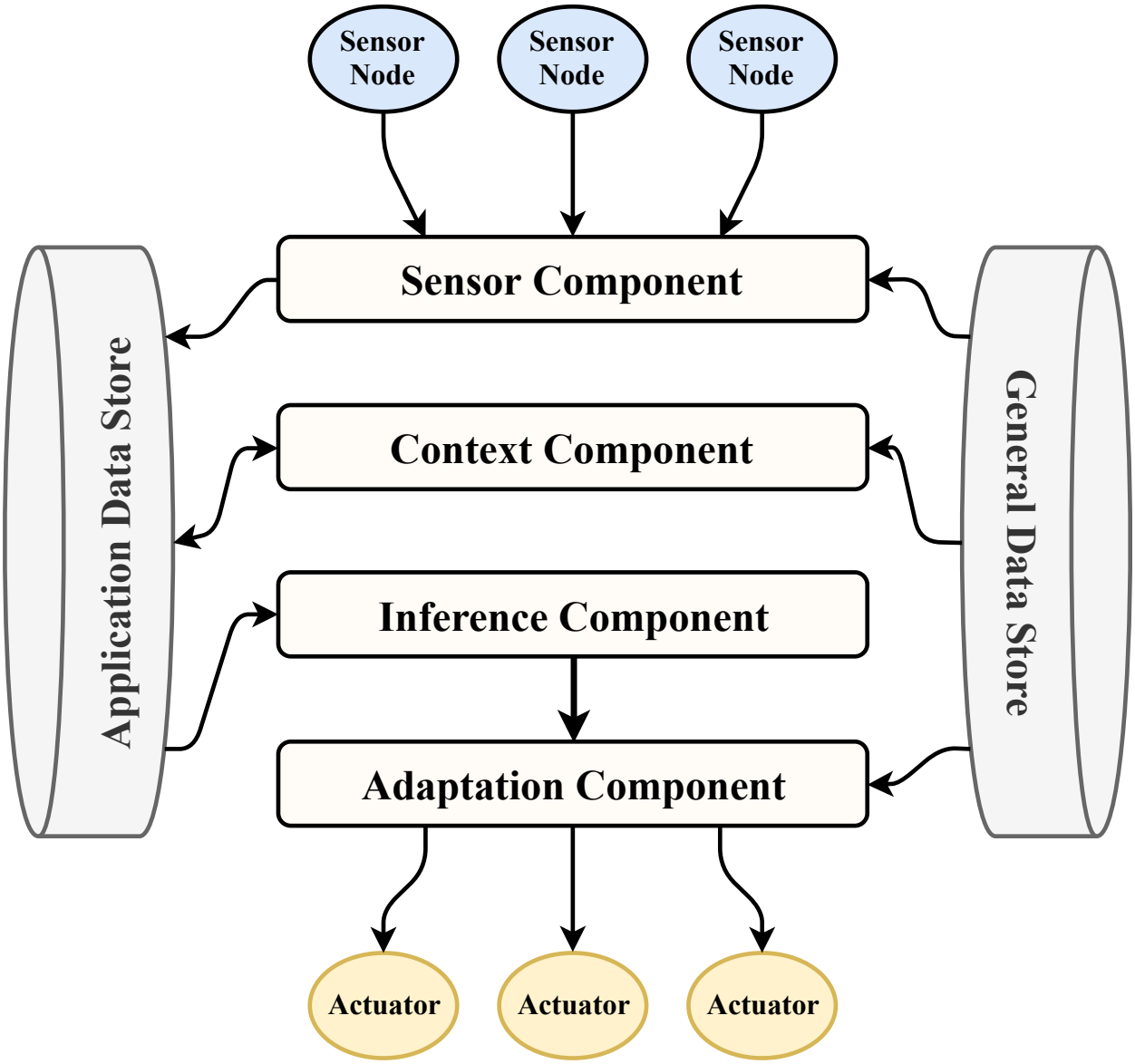

Figure 12: Context-Aware Architecture for Smart Cities

The objective of the sensor component is to receive raw sensed data from the sensors deployed in the environment, transform it, and store it in the application data store. This may trigger the context component, which will receive the new data and validate it. It will then build a new context instance and store it in the application data store. When a new context is stored, a trigger is sent to the inference component, which then proceeds to fetch the facts, rules and context from the data store, and runs the inference engine. The result of the inference engine is then sent to the adaptation component, which in turn determines which actuators to interact with and sends the appropriate command if a change in state is required.

The components in this architecture are designed to be as independent as possible from one another and to be as generic as possible. The only component that depends on another is the adaptation component, which depends on the inference component. Put together, this architecture enables sensing data in real-time, building and validating context, inferring new knowledge, and reacting on the environment through actuators. The subsequent sections will explain in more details how each of these components functions and how the components interact with each other.

4.1.1 Sensor Component

The sensor component is the point of entry to the context-aware system and the "input" step in the smartness steps. As shown in Figure 13, this component receives the sensor data collected directly from the environment, processes and validates it using information and policies from the general data store, and stores it in the application data store. This component is comprised of a controller, a transformer, a validator, a data synchronizer, and a data store connector. The controller is the main sub-component in this component. Its main objectives are to subscribe to sensor data streams, fetch sensor information and policies from the general data store, call the transformer and the validator for each sensor reading, use the data synchronizer to aggregate and normalize the readings, and call the data store connector to write the new sensor data in the application data store. The controller listens to the sensor data streams and is triggered whenever a new reading enters the stream. The second sub-component is the transformer, which uses the stored sensor information and policies to transform the data. For instance, a temperature sensor may have information related to its unit and data range stored in the general data store. It could also have a policy which specifies its data retention period. In this scenario, the transformer would check that the unit of the sensor matches the unit specified for this application, and transforms the data if required.

The third sub-component is the validation unit, which ensures the validity of the data using the information fetched from the general data store. Using the temperature sensor example, the role of the validator is to check if the new reading is within the specified data range for that sensor. It would also check to see if the data is still valid according to the data retention policy. If any of the

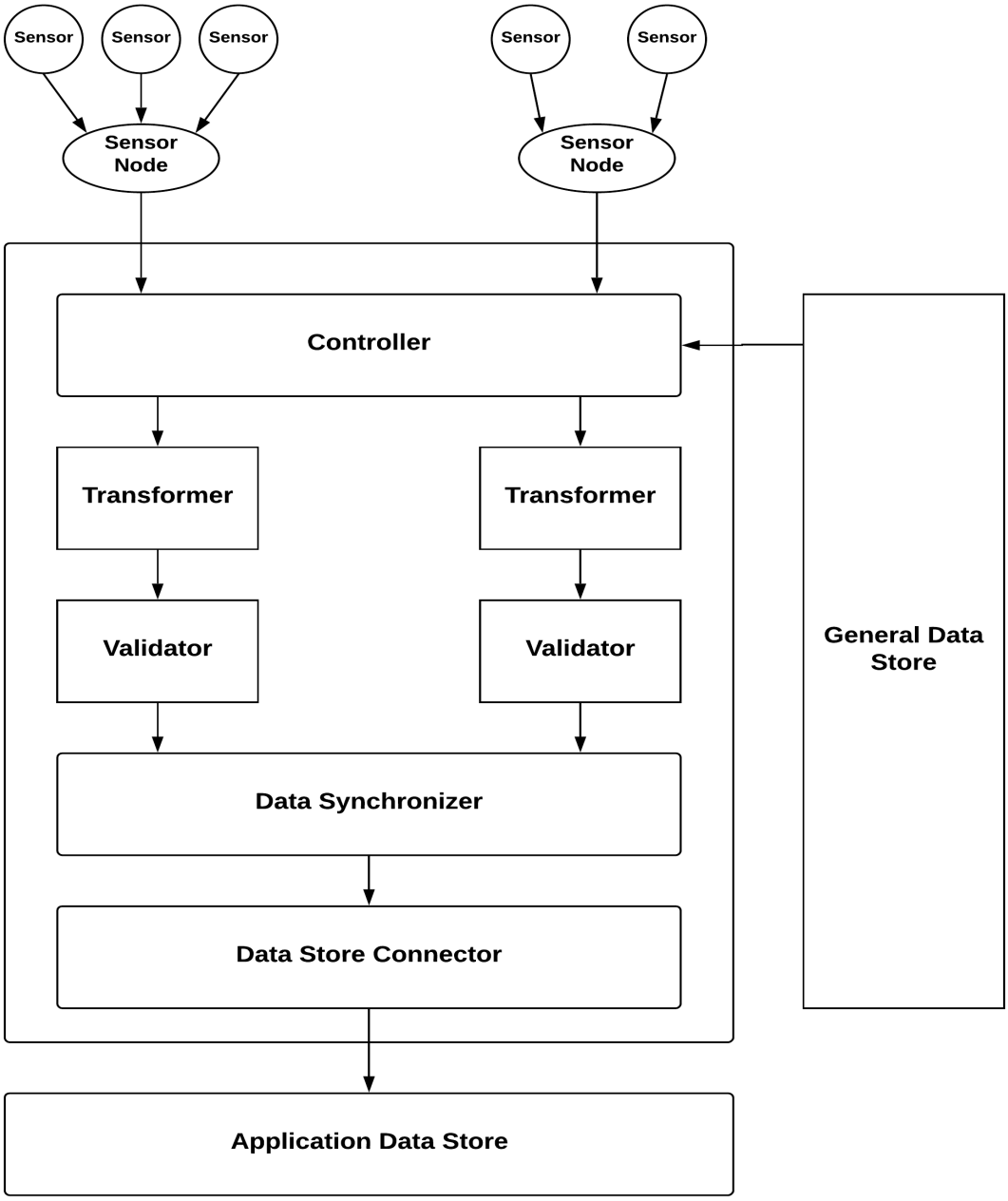

Figure 13: Sensor Component

checks fails, then the new sensor reading is not stored in the application data store. This ensures validity of the sensor data in the data store. Additional policies such as security and privacy policies may be added in the data store to provide additional protection to the system. The fourth unit is the data synchronizer, which aggregates data coming from sensors given a criteria such as type or area. It also normalizes the data before it is stored in the application data store. The last unit in the sensor component is the data store connector. This sub-component has specialized methods to

connect to the defined data stores. It receives the data that is transformed and validated, reformats it according to the data store used, and records it in the appropriate location.

In terms of privacy and security measures for this component, we use a role-based access control mechanism (RBAC) to ensure that communication and access is only allowed with specific entities in the system. For the sensor component, the controller will be granted access to receive data from registered sensor devices in the environment, read sensor information from the general data store, and write data to the sensor readings table in the application data store. It is important to note that the controller is the only unit in the sensor component that is allowed to communicate with other components in the architecture. Another method that could be implemented within the sensor component is intrusion detection, which could prevent some of the common security attacks such as a denial-of-service attack (DoS attack), malware, or SQL injection. We have not considered intrusion detection in this thesis. All in all, setting up security measures helps ensure that the communication between components is secure and the that user information stored in the databases is safe.

Overall, the sensor component subscribes to sensor data streams, listens for new data in the stream, transforms and validates the new data, synchronizes it, and stores it in the appropriate location in the application data store. This component is designed in a way that allows the definition of any type of sensor. This capability is needed because of the wide variety of sensors found in smart cities. In this component, sensors share basic methods and actions, with the possibility of defining more specialized methods if a sensor requires it.

4.1.2 Context Component

The second component in the proposed architecture is the context component. It is the first component in the "process" step used to achieve smartness. As shown in Figure 14, this component listens to changes in the sensor readings data in the application data store, and receives the processed sensor readings as soon as they are added. It then determines whether to store the new sensor reading as context in the application data store. If it decides to store it, it builds the context using syntax policies stored in the general data store and stores the context instance in the application data store.

The sub-components in this component are the controller, context manager, context builder and data store connector. The controller has a listener that monitors the sensor readings data in the application data store. When new data is added, an event is sent to the controller, forcing it to run and fetch the new data from the application data store. The controller then calls the context manager and the context builder for the new readings. If the new context data has to be stored in the application data store, the controller calls the data store connector to store it. The context manager manages the incoming sensor readings as well as the stored context. This unit keeps the

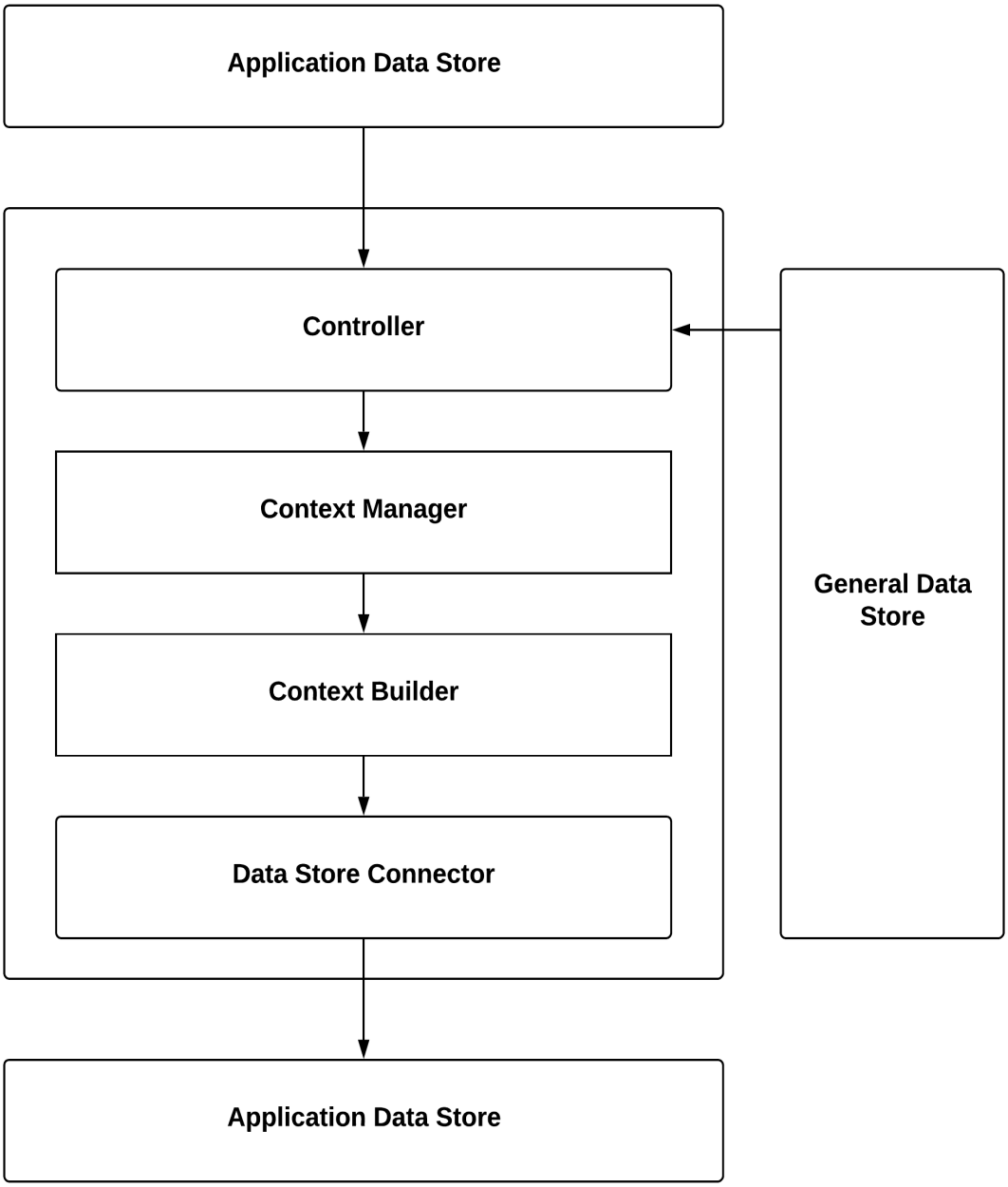

Figure 14: Context Component

stored context data up to date, decides whether to store new data as context, requests new data if needed, and removes old context instances based on data retention policies. The context manager is important because it controls the context information for the application, thus controlling the efficiency and accuracy of the system.

The context builder uses syntax policies stored in the general data store to ensure that the context is built in a syntactically correct way. In this architecture, the context is defined using key-

value pairs of dimensions and attributes. This unit is only called if the context manager decides that the new sensor reading is to be stored as context in the application data store. Finally, the last sub-component is the data store connector, which is the same one used in the sensor component. This unit connects to the application data store and stores the new context data in the appropriate location.

Similar to the sensor component, the context component uses a RBAC in order to control access between the components. The controller is assigned a role which allows it to fetch context policies from the general data store and write context instances in the context table in the application data store. The controller is the only unit that is allowed to communicate with other entities in the system.

In summary, the context component listens to changes in the sensor reading data store, receives the new data when it enters the application data store, determines whether or not to store it as context, ensures that the context data is valid and up to date, builds new context based on policies stored in the general data store, and stores the new context information in the application data store.

4.1.3 Inference Component

The inference component shown in Figure 15 is the third component in the context-aware architecture and the second component in the "process" step. It is the main component for smartness of the architecture as it receives context information based on sensor data from the application data store and uses adaptation rules and facts stored in the application data store to produce adaptations. This component contains a controller, an inference engine helper, and an inference engine. The controller continuously listens to changes in the context information in the application data store. Upon receiving an event indicating that one or more context instances have been added to the application data store, the controller fetches the new context information. It then calls the inference engine helper unit to prepare the input for the inference engine, which includes adaptation rules, facts, and context instances. It then runs the inference engine and sends the inferred adaptations to the adaptation component.

The inference engine helper's main purpose is to prepare the context instances, adaptation rules, and facts so that they meet the requirements of the inference engine. The first thing this sub-component does is load the adaptation rules and facts for the current application. Then, it performs necessary checks to ensure that the context instances, adaptation rules, and facts conform to the requirements of the inference engine. These requirements may include syntactic conformance and the ordering of the input given to the inference engine. For the scope of this thesis, it is assumed that syntactic conformance is sufficient to run the inference engine. After all the requirements mentioned above are satisfied, the inference engine helper indicates to the controller that it can

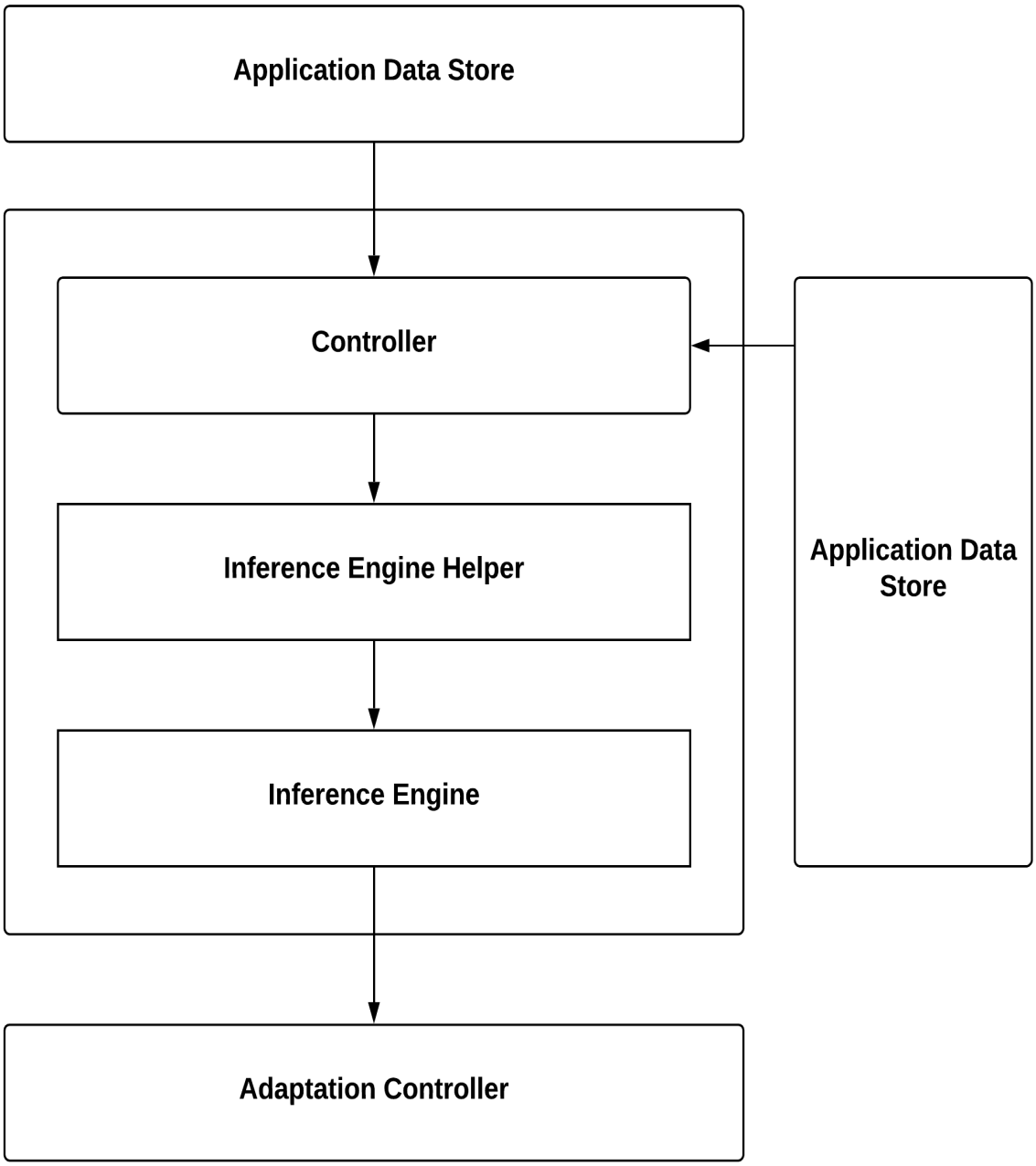

Figure 15: Inference Component

run the inference engine with the input. The role of the inference engine is to infer new facts and knowledge based on context information, stored policies, adaptation rules, and facts from the the data stores. As soon as the inference engine completes its run, its output is formatted by the helper in order to have a uniform format regardless of the inference engine. Finally, the formatted output is sent to the adaptation component. Thus, the inference engine component in the architecture is quite

generic, in the sense that it can be made to work for any rule-based inference engine. Consequently, the architecture provides the flexibility for different designs to plug-in different inference engines to achieve their goals.

In order to guarantee that the context-aware system functions at all times, there should be a way for the inference component to request any missing information from other components. For instance, if there is an adaptation rule in the application data store that requires temperature and humidity readings, but the inference component only receives context information containing a temperature reading, then this component must be able to communicate with the context component and request the required information. The context manager within the context component will then check to see if there is a valid humidity reading stored in the application data store. If the information is available, the context component controller will send it to the inference component. If it is not available, then the context component will request it from the sensor component, which will then fetch it directly from the humidity sensor in the environment. Upon receiving the humidity reading, the sensor component will then follow the regular steps of storing the reading in the application data store, which will then be processed by the context component, and then stored as context in the application data store. When this happens, the inference component will receive the new context information that it needed.

The inference component is a key component in the architecture, and as such, proper security measures need to be put in place to ensure its protection. In order to do so, a role must be assigned to the inference component controller, granting it permissions to perform its tasks. These permissions include reading adaptation rules, facts, and context instances from the application data store. The facts include information such as static context instances, user information, user preferences, and privacy policies. The user preferences and privacy policies can be used by the inference engine in order to protect user data. The controller is also allowed to request additional context information from the context component. However, it is not allowed to receive the information directly from the context component as the context instances need to be stored in the application data store first. After this is done, the controller can fetch the new context information with the permissions granted to it. Finally, the controller is granted a permission that allows it to trigger the adaptation component controller and send the new inferred data.

All in all, the main responsibilities of the inference component include continuously listening for new context information from the application data store, loading the adaptation rules and facts for the current application from the application data store, preparing the adaptation rules, facts, and context instances to meet the syntactic requirements of the inference engine, running the inference engine, and sending the new inferred knowledge to the adaptation component.

4.1.4 Adaptation Component

The adaptation component shown in Figure 16 is the fourth and final component in the proposed architecture, and it makes up the "output" step in the smartness steps. This component receives a structure from the inference component that contains inferred adaptations as well as time and location metadata. Using this information, the adaptation component determines which actuators to communicate with, transforms the actions into actuator commands, and sends the commands to the actuators. The sub-components included in this component are the controller, actuator, and actuator connector. When the inference component executes, the inferred knowledge it generates is sent to the adaptation component. The controller continuously listens for the arrival of new inferred knowledge. As soon as it receives one, the actuator unit determines which actuators need to be activated based on the metadata loaded by the controller from the general data store. This metadata includes possible actions, location, and a mapping of actuator commands and actions. After the actuator is selected, the actuator unit transforms the inferred action into a meaningful actuator command using the metadata loaded from the general data store. Finally, the controller uses the actuator connector to locate the appropriate actuators, connect to them, and send the appropriate commands to them.

In order to better understand how the adaptation component works, Table 1 shows an example of sensor reading ranges coming from a temperature sensor in a room along with the associated statuses and actions. In our context-aware architecture, the information displayed below would be considered static context information, and would be treated as facts. If the temperature sensor were to send a reading indicating that the temperature is 19 °C, then this reading would be transformed into a context instance by the context component and stored in the application data store. The inference component would then run the inference engine. Using the information from Table 1 as well as some predefined adaptation rules, it would determine that the status of the room is "cold" and the action required is "Start Heater". The inference component would then send the action, the timestamp, and in some cases the location of the sensor to the controller of the adaptation component. Each actuator would have its associated metadata stored in the general data store. The metadata would include possible actions by the actuators, actuator command per action, and actuator locations. Using this information, the actuator unit within the adaptation component would determine which actuator to select, and the actuator connector would send it the appropriate command.

As with the other components in this architecture, the adaptation component uses RBAC to manage communication and access between components. The controller is the only unit that is allowed to communicate with other components in the system, and as such, it is assigned a role with limited permissions. This role has permissions that allow the controller to receive the inferred

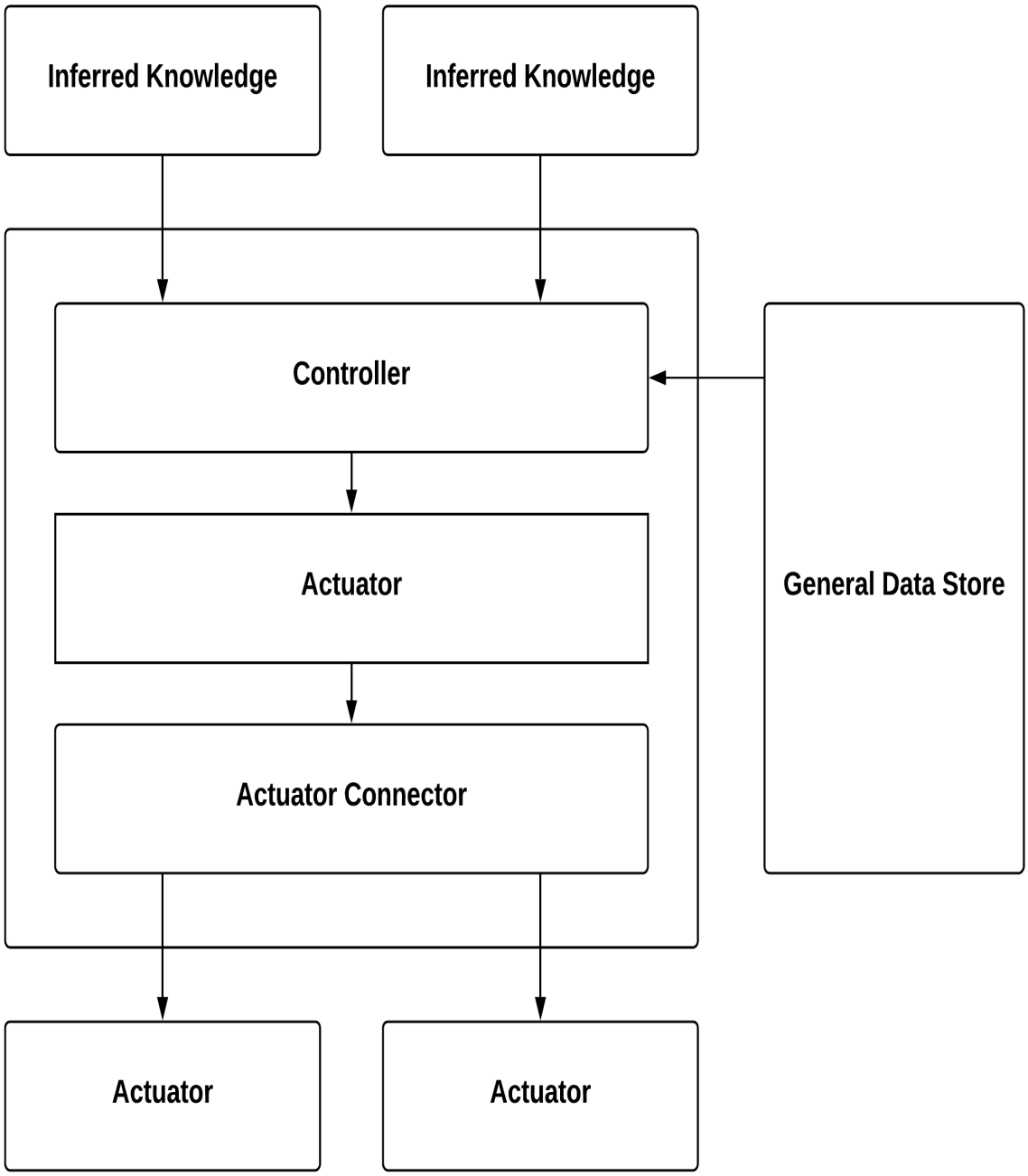

Figure 16: Adaptation Component

knowledge from the inference component, load actuator metadata from the general data store, and communicate with the actuators that are registered for the specific application. The communication between the controller and the actuators includes sending commands to the actuators and receiving responses and actuator statuses from the actuators.

Temperature (°C)	Status	Action
Less than 20	Cold	Start Heater
20-25	Normal	Nothing
More than 25	Hot	Start Cooler

Table 1: Temperature Adaptation Example

In summary, the adaptation component receives the inferred knowledge from the inference component and determines the appropriate actuators and commands using the new knowledge and the information stored in the general data store. Afterwards, it transforms the adaptation into an actuator command using the information loaded from the general data store, and connects and sends the command to the appropriate actuators.

4.1.5 Data Stores

In the context-aware architecture shown in Figure 12, there are two data stores available, namely the general data store and the application data store. The general data store is used to store information that can be used by more than one application. The information included in this data store includes sensor and actuator metadata as well as policies that are not restricted to a single application. The application data store is used to store data related to a specific application. It stores facts, adaptation rules and context information related to that application. The adaptation rules in the application data store are used together with context to determine the appropriate adaptations and reactions to be taken on the environment. The context information stored in this data store is always changing since it comes from the sensors and other input devices in the environment.

The facts stored in the application data store include information such as a user information, user preferences, and privacy policies. As mentioned in Chapter 2 when discussing privacy of Wireless Sensor Networks (WSNs), the same two sets of rules described can be used to provide users within a smart city application with privacy. The first set of rules is application policies. These policies are used to ensure that the application functions properly. For example, a home intrusion detection system that detects unwanted windows and doors openings may need to know the number of residents in a home, whether or not the residents are at home, and contact information to use in case of an intrusion. This type of information is required to ensure that the home intrusion detection system functions properly. In order to protect this sensitive data, the second set of rules is put in place. These rules are user preferences, which are policies chosen by the users to control how their data is used. In the home intrusion detection system example, the resident may have a policy that allows the system to share his information with the authorities in case of an intrusion, but not in other cases. The two sets of policies described are stored in the application specific data

store.

An additional action that can be taken in order to protect user data and ensure its confidentiality is to encrypt the data stores in the architecture. In Chapter 2, two encryption methods were introduced, namely symmetric and asymmetric encryption. These two methods were discussed in details, and the conclusion was that symmetric encryption is faster and more efficient since it only uses one key to encrypt and decrypt the data. Asymmetric encryption is a more recent and complex technique. It uses two keys: a public key to encrypt the data and a private key to decrypt it. While this method is more secure, it is slower and more complex than symmetric encryption. The symmetric encryption was chosen for WSNs in order to optimize resource utilization, whereas asymmetric encryption is used to encrypt the data stores since there are more computing resources available and the data stored are very sensitive.

Another method of ensuring data confidentiality is through access control. In this method, various roles and permissions are created and assigned to the components in order to control the access of the components in the architecture to the data stores. Minimal access is granted to the components in order to ensure that they function properly without compromising the security of the system. These two security methods help protect the user's data and allow the application to be compliant to various policies and regulations enforced by governments and users.

4.2 Analysis and Discussion

So far, we focused more on presenting the context-aware architecture developed in this thesis. In this section, our architecture will be compared those presented in Chapter 3, and the merits and contributions of our architecture will be discussed in more details. Table 2 compares the architectures surveyed in Chapter 3 to the one proposed in this thesis.

4.2.1 Sensing

In [FC04], sensor nodes are used to gather data from multiple sensors in the environment. In the architecture, the authors use a SensorListener class in order to get the data from the sensor nodes. In [Che04], sensor data is fetched using the context acquisition module. The author defines three types of context acquisition methods, namely directly from sensors, using a middleware infrastructure, and from a context server. The authors in [SDA99] present context widgets in order to handle different data sources. Using this method, they separate the context acquisition concerns from the application since the widgets hide the complexity of fetching data from various sensors. Additionally, since the widgets are encapsulated software software components, they can easily be reused. In [Hna11], the author uses a sensor mechanism to connect to sensors, listen to

Paper	Sensing	Context Model	Context Processing	Security and Privacy
[FC04]	Sensor listener	Relational data model	CASS Inference engine & relational knowledge base	Not available
[Che04]	Context acquisition module	Ontologies (OWL)	Context reasoning engine & context knowledge base	SOUPA policy language
[SDA99]	Context widgets	Attribute-value tuples	Context interpretation & aggregation	Context ownership
[Hna11]	Sensor mechanism	Key-value pairs	Context situations	Not available
[MEeAT16]	Sensor agents	Ontologies	Pellet ontology reasoner	Not available
[RSC07]	Sensors	MVC	MVC	Not available
Proposed Architecture	Sensor component	Key-value pairs	Any rule-based inference engine & context knowledge base	Application & user policies, RBAC, encryption

Table 2: Comparison between proposed and surveyed architectures

incoming data, and process it. The listeners provide a layer of abstraction between the sensors and the context-aware application, and the connectors enable the communication between the sensors and the application. In terms of data processing, the sensor mechanism transforms, validates, and aggregates the data coming from the sensors before sending in the next component in the system. The architecture presented in [MEeAT16] uses a sensor multi-agent system to gather data from the environment. It has the responsibility of gathering raw contextual information using a sensor network. In this architecture, each sensor is assigned an agent that sends the readings to a sensor central agent. Finally, the work presented in [RSC07] mentions that sensors can detect implicit user interactions. However, this paper fails to provide details on how this is done, since the authors are more preoccupied with representing context in the user interface.

In the architecture presented in this thesis, a sensor component is used to interact with the sensors in the environment. Each instance of the sensor component is tailored to fit the requirements of each sensor in the environment. The sensor component has methods to connect to the sensors, receive their readings, transform and validate the readings, aggregate and normalize them, and store them in the application data store. Overall, various aspects from [FC04, Che04, Hna11] have been incorporated in the architecture presented in this thesis. The work presented in [RSC07] does not discuss sensing in details. In [SDA99], the overall concept of widgets is an interesting one, especially for context acquisition. The main advantage of using their proposed method is that it

provides a level of abstraction between the sensors and the application. Although widgets are not used in the work presented in this thesis, there is a level of abstraction between the sensors and the application. In our architecture, an important feature of the sensor component is to be as generic as possible while allowing the possibility of being extended to answer the specific needs of some sensors. This way, most sensors can be deployed to the environment and used without the need of any changes to the sensor component.

4.2.2 Context Modeling

In [FC04], the context is modeled using a relational data model, which is known to be robust and mature. The main advantage of using a relational model is that all data interactions will be made in the Structured Query Language (SQL), which is the most widely used language to deal with structured data. The problem with using relational databases is that they don't scale well and they are not flexible, which can be problematic in smart city applications since the number of sensors in the physical environment could vary from a few hundred to hundreds of thousand sensors.

The works presented in [Che04, MEeAT16] use ontology to model context. In [Che04], an OWL-based ontology called COBRA-ONT is introduced to model context. In [MEeAT16], the authors mention that the context is modeled using ontology but do not provide details. Although an ontology-based representation allows for the writing of explicit, formal conceptualizations of contextual information, it can also be inefficient and difficult to work with [AvH04]. According to [AAS18], most ontological approaches use ontology concepts but mainly through ad-hoc structures. In fact, the author believes that Formal Concept Analysis (FCA) is the only formal way for formalizing context. As a formal approach, FCA is credited for its theoretical model, well-defined operations and expressiveness. However, the goal of FCA is to formally capture relationships between entities. FCA provides no representation to context.

The work presented in [SDA99] handles context in simple attribute-value-tuples encoded using XML for transmission. Key-value pairs are used in [Hna11] to model context. This is an interesting choice because a key-value pair is a simple data structure that is strongly typed, which means it is more implementable. Finally, in the work presented in [RSC07], only the user context is modeled because according to the authors, real world objects don't have context. This is a very different approach from our work and those surveyed in Chapter 3. The authors claim that this is a phenomenological view of context, whereas our work follows the positivist/engineering view. The phenomenological view says that people create and maintain context during interaction, while the positivist/engineering view is more concerned with representing context.

In our work, context is modeled as dimensions and attributes. This method has a well-defined representation with the help of the lattice-based formalism used in building context structures. It

also has context operators to manipulate context and makes it usable for real-world applications. Finally, it supports context structures (e.g. hierarchical and tree structures).

4.2.3 Context Processing

The CASS inference engine is used in [FC04] for context reasoning and processing. This is an interesting choice because it enables the system to generate new facts based on rules and context. Similar to the CASS architecture, the architecture in [Che04] uses a rule-based inference engine for context reasoning. Both of these approaches are similar to the approach used in our thesis. The work presented in [SDA99] offers facilities for both context aggregation and context interpretation. The context aggregators are responsible for composing context of particular entities by subscribing to relevant widgets, while context interpreters provide the possibility of transforming context. In [Hna11], context situations are used for context processing. A context situation is a custom state that occurs when predefined environment conditions are met. Situations are represented as expressions evaluated against context dimensions, and they are defined using logical conditions over contexts. In [MEeAT16], the authors mention that they use the Pellet ontology reasoner without providing much details about it. In the [RSC07], the authors do not discuss any methods for context reasoning, as they assume that it is already done. Their main focus is to communicate changes in context and inferences to users through a user interface.

In this thesis, we can interface our architecture with any inference engine provided the input and output of the engine are known. That being said, a context-based, rule-based inference engine is used in the proposed architecture to reason with context and infer new knowledge. This inference engine has formal basis, simple syntax, and declarative semantics which makes it theoretically sound and practically extendable.

4.2.4 Security and Privacy

In [Che04], the author presents a policy-based approach (SOUPA policy language) to deal with privacy protection. In the work presented in [SDA99], the authors consider a component called "Authenticator" that validates the source of the information sent to another component using a public-key infrastructure. This paper also uses the concept of context ownership to preserve user privacy. Using this method, individuals get assigned context data related to them, and they can control who can access their data. In [Hna11], the author introduces the Workflow and Policy Expression Language to define adaptations. This language supports the execution of actions with respect to a set of policies. Although this can be used to enforce privacy policies, there are no mentions of this language being used to protect privacy in the paper. The works presented in

[FC04, RSC07, MEeAT16] do not discuss security and privacy, both of which are crucial to smart city application design.

In this thesis, application and user policies are used to ensure protection of user information. These policies are defined using predicate logic, in order to have the same framework and syntax for all the rules in the system. Furthermore, roles with different levels of permissions are used to control the communication between components. This ensures that a component can only read or write from another component when the role allows it. In order for an architecture to be used in a smart city, privacy and security measures must be considered to protect user information and gain trust among citizens.

4.3 Summary

In this chapter, the context-aware architecture developed in this thesis was presented and discussed. The sensor, context, inference, and adaptation components were explained in details, and the interactions between the different components was highlighted. A special focus was on privacy and security, as they were discussed at each level of the architecture. Finally, a comparison between the architecture presented in this thesis and those reviewed in Chapter 3 was done, and the main contributions of our architecture were highlighted. This architecture is designed to be generic enough to be used by different smart city applications. Furthermore, context is modeled using a well-defined representation of dimensions and attributes, making it suitable for real-world applications. The architecture also interfaces with any rule-based inference engine, provided the input and output of that engine are known. RBAC is used to control the communication between components, and symmetric encryption is used to protect sensitive information in the data stores. In the next chapter, a detailed design of the architecture will be presented to better illustrate how the components work and how they interact with each other.

Chapter 5

Detailed Design

In this chapter, a detailed design of the architecture and its components are given. For each component, we describe the interface structure, internal component functionality, and interactions with other components.

5.1 Design Pattern

In this architecture, the factory design pattern was used to accommodate the requirements of smart cities. This pattern is used in the *Sensor* component, the *DataStore* component, and the *Adaptation* component. We use this pattern in order to create objects without having to expose the logic to their respective controllers. In the *Sensor* component, we have a base *Sensor* class, and more specialized classes for different sensor types. The base class contains the attributes and method definitions that are common to all the sensor types, whereas the specialized classes override some of the methods defined in the base class to meet the requirements of the sensor in question.

In the case of the *DataStore* component, most databases share the same basic functionalities such read, write, and delete. Therefore, the factory pattern can be used to have one *DataStore* class and more specialized classes that implement the functionalities that are specific to a certain database. As for the *Adaptation* component, many actuators will have similar implementations. Therefore, using this design pattern would allow us not to duplicate the implementations. The factory pattern also provides some flexibility needed when dealing with classes that share some similarities but have some differences as well. It provides a level of abstraction which helps hide the implementation of the sensor classes, since other components only need to call one class and specify which sensor instance they wish to instantiate.

5.2 Detailed Component Description

As explained in Chapter 4, the context-aware architecture in this thesis is composed of a sensor, a context, an inference, and an adaptation component. The general and application data stores are used to store contexts, facts, and rules. The application data store is also used for the communications between some of the components. The following subsections will delve more into details about the components of our proposed architecture.

5.2.1 Sensor Component

Figure 17 shows the sensor component, which contains a *SensorFactory* class, a *Sensor* class, a *DataSynchronizer* class, and a *Sensor* controller. The controller is responsible for calling and instantiating other classes in this component. When the controller receives data from the physical sensors, it reads the sensors' types and IDs and instantiates the appropriate *Sensor* objects using the *SensorFactory* class. Since there can be many sensors sending data at the same time, we use the *DataSynchronizer* class to aggregate data coming from similar sensors in the same area and to normalize the data before it is stored in the application data store. Once the data is ready, the controller uses the *DataStore* component to write it to the database. It does so by instantiating a *DataStore* object using the *DataStoreFactory* class and the appropriate data store type.

Figure 18 shows the sequence diagram for the interactions between the classes in the sensor component as well as the interactions of the sensor component with other architectural components. As soon as sensor data is received by the controller, a *Sensor* object is created using the Factory pattern. Then, the data read by the sensor is transformed and validated. As soon as all the sensor readings have been processed, the data is sent to the *DataSynchronizer* in order to be aggregated and normalized. Finally, an instance of *DataStore* is created and the data is stored in the application database.

SensorFactory Class

The *SensorFactory* class is used to instantiate the correct sensor object using the type of the sensor. It uses the Factory design pattern discussed in Section 5.1. It contains a static method, called *new_sensor*, which takes the sensor type and ID as parameters, as described in Table 3.

Method	Description
new_sensor	This method takes the sensor_id and sensor_type as parameters and returns the appropriate *Sensor* class.

Table 3: *SensorFactory* Methods Description

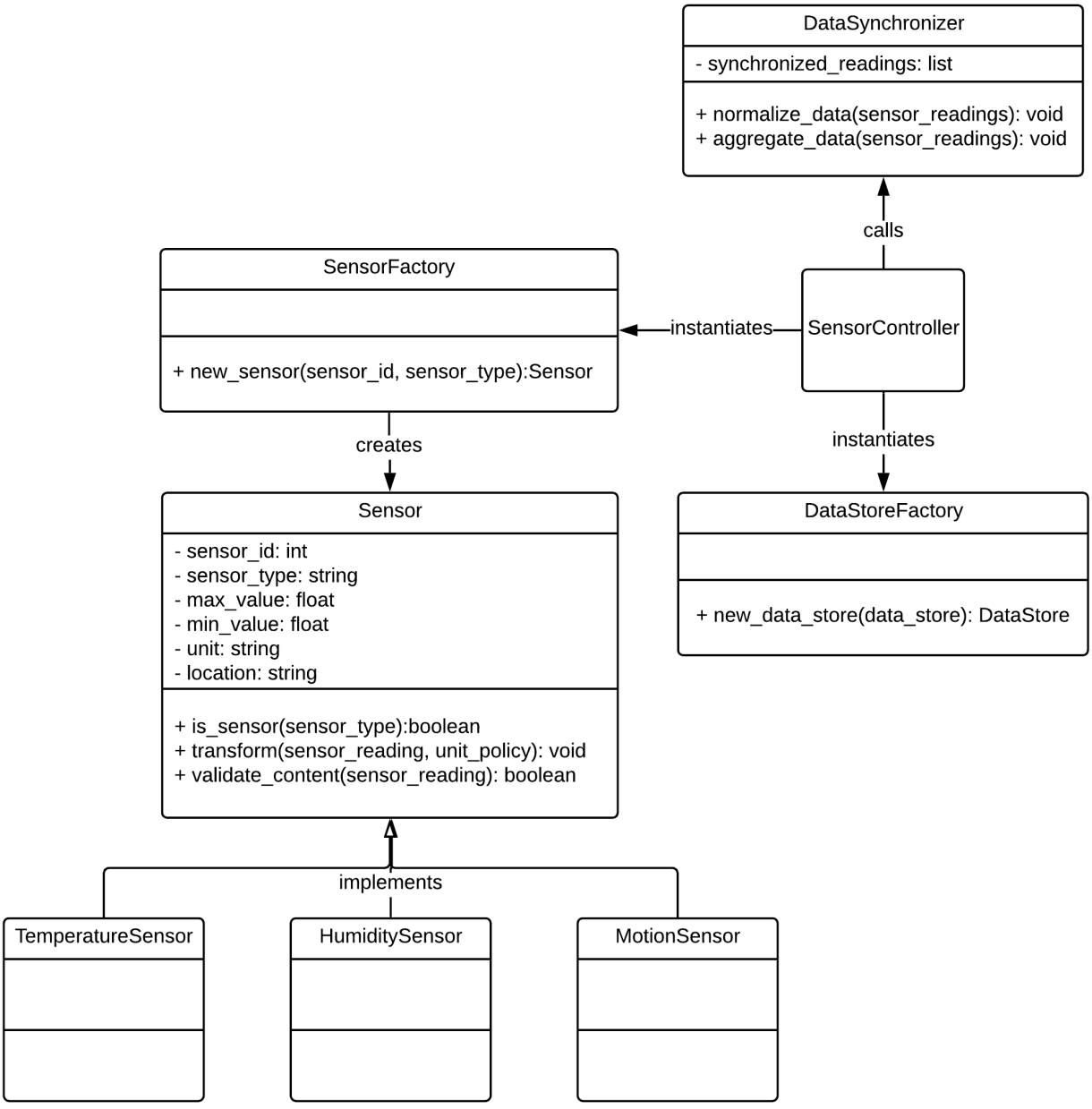

Figure 17: Sensor Component Class Diagram

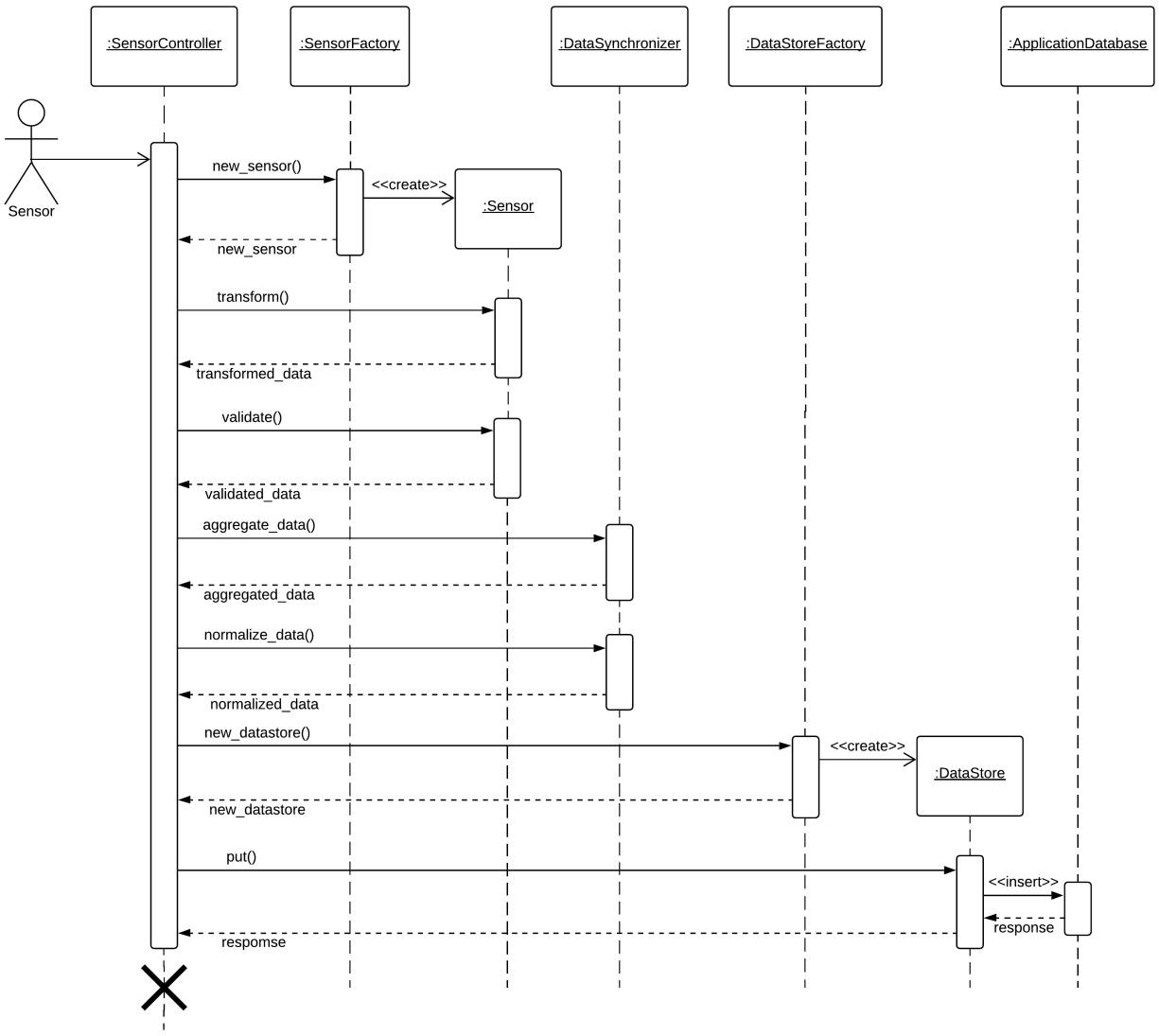

Figure 18: Sensor Component Sequence Diagram

Sensor Class

The *Sensor* class is a general class that contains variables and methods that are common to most sensors, as described in Tables 4 and 5. There are also specialized sensor classes that inherit from the *Sensor* class. Examples of such classes include *TemperatureSensor*, *HumiditySensor*, and *MotionSensor* illustrated in Figure 17. The *SensorFactory* class is used to determine which sensor objects to instantiate.

Variable	Description
sensor_id	ID of the sensor device.
sensor_type	Type of the sensor device.
max_value	Maximum value that can be read by the sensor device.
min_value	Minimum value that can be read by the sensor device.
unit	Unit of value sent by sensor device.
location	Location of the sensor device.

Table 4: *Sensor* Variables Description

Method	Description
is_sensor	Static method that takes sensor type and returns true if the sensor type parameter is equal to sensor type in the class, otherwise it returns false. This method is called by the *SensorFactory* class.
transform	Method that takes a sensor reading and a policy and returns the value transformed to conform to the policy.
validate	Method that takes a value as a parameter and ensures that it is valid. This method returns true or false.

Table 5: *Sensor* Methods Description

DataSynchronizer Class

The *DataSynchronizer* class is responsible for ensuring that the sensor data that is most recent, normalized, and aggregated. The variables and methods of this class are described in Tables 6 and 7.

Variable	Description
synchronized_readings	List containing the synchronized readings.

Table 6: *DataSynchronizer* Variables Description

Method	Description
aggregate_data	Method that takes the normalized sensor readings and aggregates the readings so that data can be grouped by criteria such as location and sensor type.
normalize_data	Method that takes the sensor readings and populates the synchronized_data list with the normalized readings.

Table 7: *DataSynchronizer* Methods Description

Sensor Controller

The *Sensor* controller is the component that acts as an entry and exit point to the sensor component. When the controller receives new sensor data, it instantiates a new *Sensor* object using the *SensorFactory* class and the sensor type it receives from the sensor reading. After the data is processed and is deemed fit to be stored, the controller instantiates a *DataStore* object in order to write the sensor reading in the appropriate table of the application database.

Another important aspect of the *Sensor* controller is that it can reject incoming data if there is a security compromise. One way to do this is by checking the signature of the sender. If it does not match a list of allowed sources, it can reject the data it is sending. Another way the controller can reject data is by verifying that the sender is actually allowed to send data according to some security and privacy policies stored in the database.

5.2.2 Context Component

The *Context* component is responsible for creating context instances out of sensor readings, storing the context instances in the context table in the application database, and managing the context instances in the application database. The *Context* component contains a *Context* class and a *Context* controller, as shown in Figure 19.

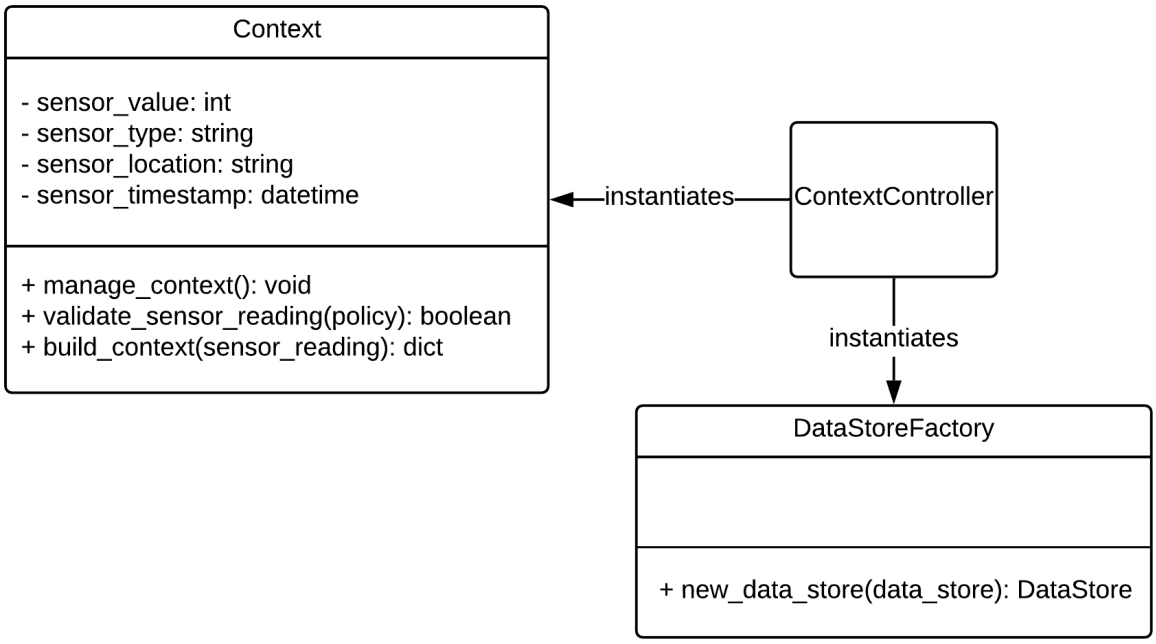

Figure 19: Context Component Class Diagram

Figure 20 shows the sequence diagram for the *Context* component. Upon receiving an event from the sensor table in the application database, the controller validates the reading using the *validate_sensor_reading()* method and checks to see if the context needs to be stored using the *manage_context()* method. If it does, the context instance is built using the *build_context()* method. Finally, the appropriate *DataStore* object is used to store the context in the application database so that it can be used by the *Inference* component.

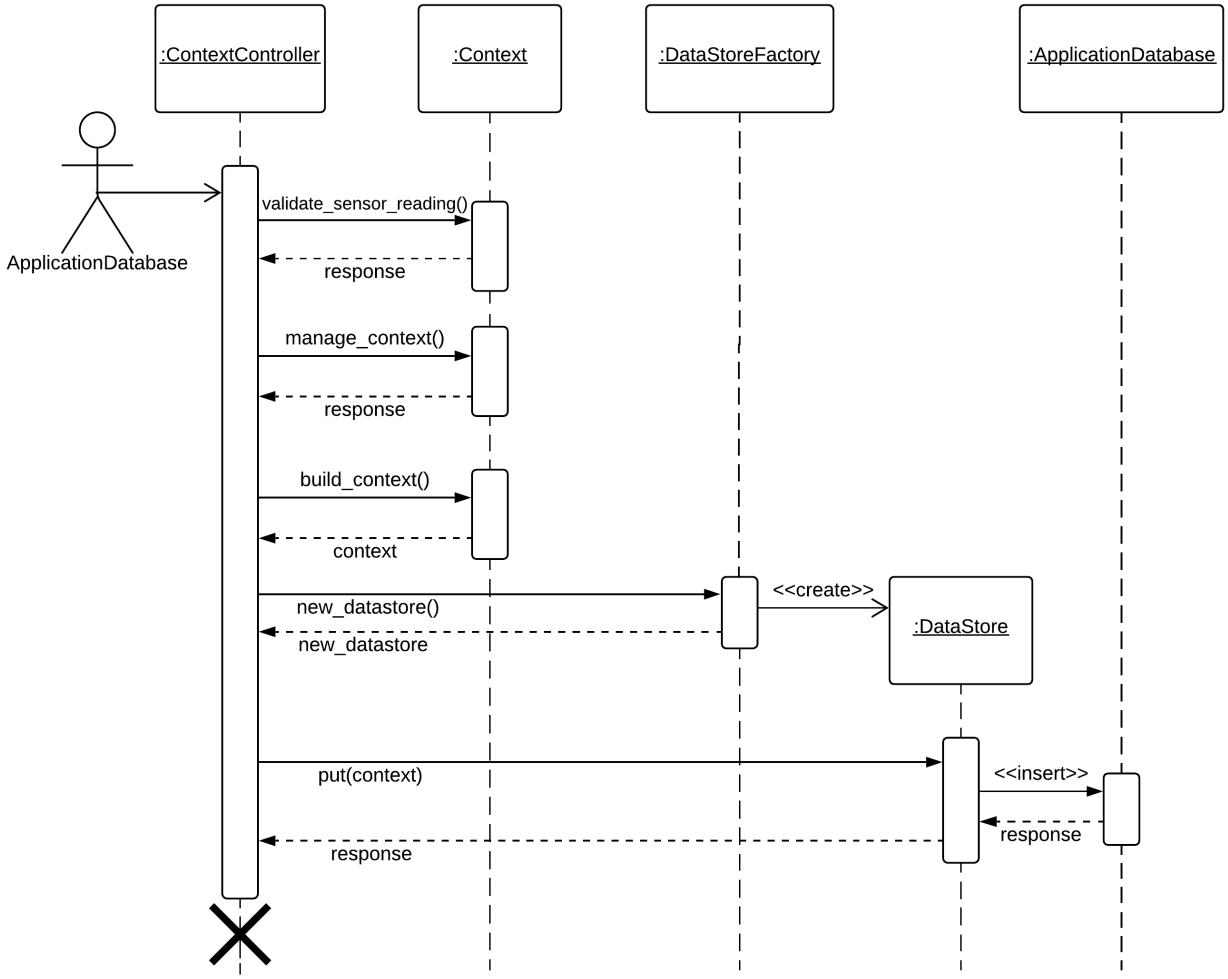

Figure 20: Context Component Sequence Diagram

Context Class

The *Context* class contains the variables and methods used to manipulate context instances. Tables 8 and 9 list these variables and methods along with a short description.

Variable	Description
sensor_value	Value received from the sensor device.
sensor_type	Type of the sensor device.
sensor_timestamp	Timestamp of value received from sensor device.
sensor_location	Location of the sensor device.

Table 8: *Context* Variables Description

Method	Description
manage_context	Method that manages the stored context instances to determine if they are still valid and whether to store new context instances.
build_context	Method that builds a context instance from the sensor reading while respecting the syntax guidelines and requirements for context instances.
validate_sensor_reading	Method that validates a sensor reading based on stored policies. An example of such policies can be a "Time to live" (TTL) policy.

Table 9: *Context* Methods Description

Context Controller

The *Context* controller is responsible for receiving sensor readings from the application database as soon as they are stored. A *Context* object is then created in order to validate the incoming data, determine whether to store this new information in the application database, and write the built context instance in the appropriate table of the application database. Similar to the way the *Sensor* controller applies privacy and security measures, the *Context* controller can reject incoming data in the event of a compromise.

5.2.3 Inference Component

The *Inference* component is responsible for operating the inference engine. It does so with the help of a few methods that prepare the input to the engine and run the engine if necessary. In our work, the architecture is not dependent on the inference engine, and so we can plug any inference engine in order to perform reasoning. This component contains an *InferenceEngine* class as well as a *InferenceEngine* controller. Figure 21 shows the class diagram for this component.

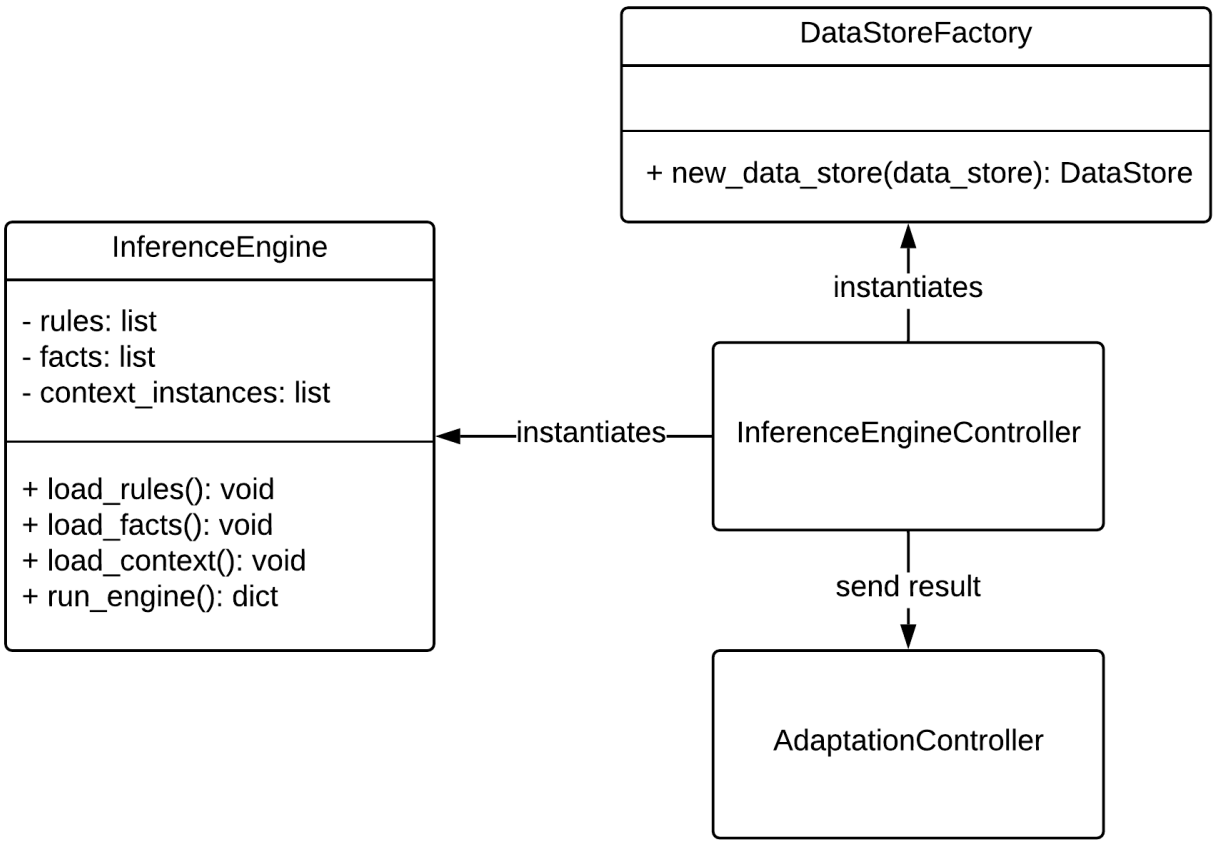

Figure 21: Inference Component Class Diagram

Figure 22 shows the sequence diagram for the *Inference* component. The first thing the controller does upon receiving an event from the context table in the application database is load adaptation rules, application facts, and most recent context instances. Once this is done, it runs the inference engine with the rules, facts, and context instances as parameters. Finally, it sends the inferred data to the *Adaptation* controller.

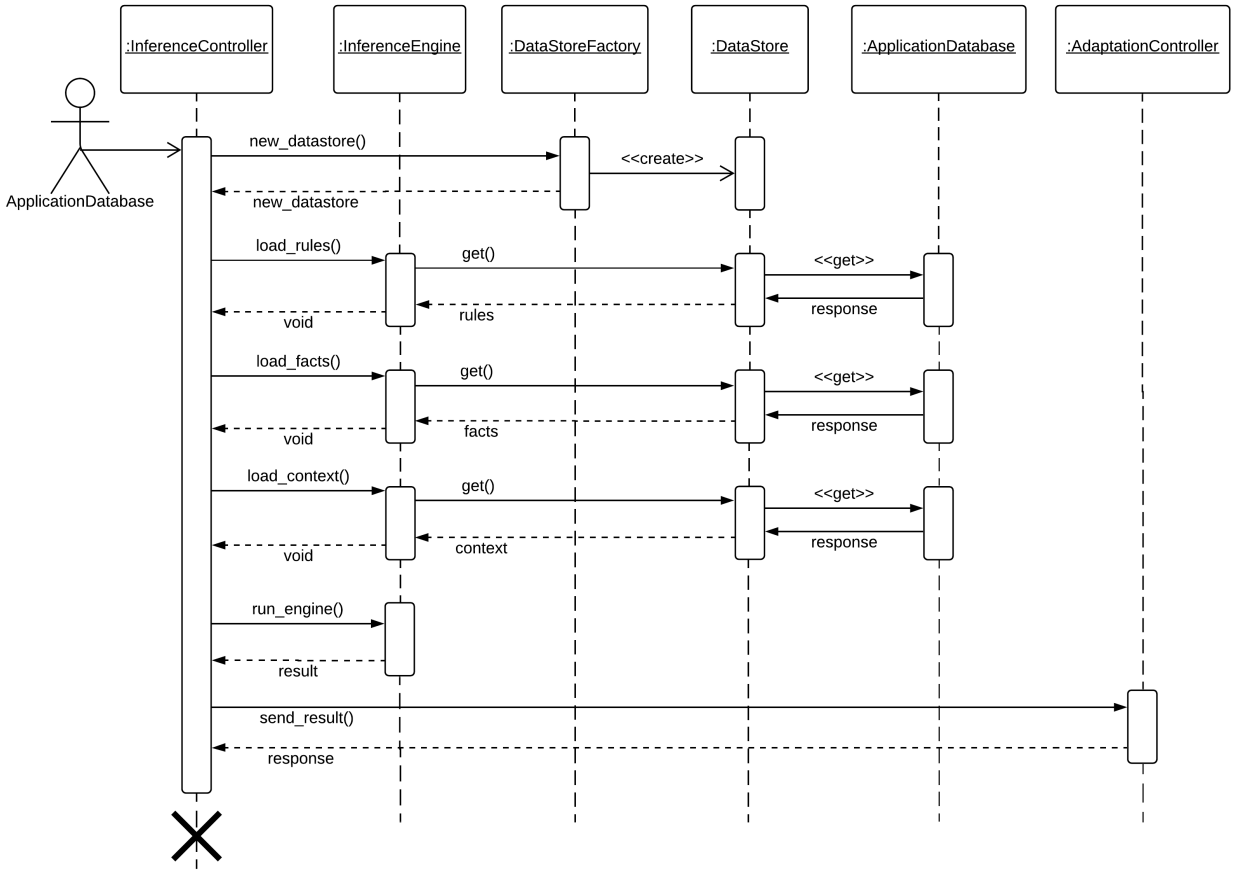

Figure 22: Inference Component Sequence Diagram

InferenceEngine Class

The *InferenceEngine* class is responsible for preparing the input to the inference engine and running it. Preparation steps include loading adaptation rules, application facts, and context instances from the databases and ensuring that syntactic requirements are met. Tables 10 and 11 list these variables and methods along with a short description.

Variable	Description
rules	List of rules loaded from the application database.
facts	List of facts loaded from the databases.
context_instances	List of context instances loaded from the application database.

Table 10: *InferenceEngine* Variables Description

Method	Description
load_rules	Method that fetches the rules from the application database and populates the rules list.
load_facts	Method that fetches the facts from the application database and populates the facts list.
load_context	Method that fetches the context instances from the application database and populates the context_instances list.
run_engine	Method that runs the inference engine using the rules, facts, and context instances loaded from the application database. It returns the inferred data.

Table 11: *InferenceEngine* Methods Description

InferenceEngine Controller

The *InferenceEngine* controller is triggered whenever a new context instance is added to the application database. As soon as it is triggered, the controller will make the privacy and security checks mentioned for the sensor and context controllers. Then, it will instantiate a *Datastore* object in order to fetch context instances, facts, and rules from the application database. Following that, it will instantiate an *InferenceEngine* object in order to load rules, facts, and context instances and run the inference engine. If the engine runs successfully, the controller will send a trigger to the *Adaptation* component in order to proceed with the inferred data.

5.2.4 Adaptation Component

The *Adaptation* component ensures creation of the appropriate actuator commands based on the inferred data from the inference engine, selection of actuators in the environment, and transmission of the commands to the corresponding actuators. This component also verifies the status of all the actuators in the environment to ensure that they are functionning. As shown in Figure 23, the *Adaptation* component contains an *ActuatorFactory* class, an *Actuator* class, and an *Adaptation* controller.

Figure 24 shows the sequence diagram for the *Adaptation* component. When the controller receives a trigger from the *Inference* component, it creates a new Actuator object based on the actuator type using the Factory pattern. Then, it transforms the inferred action into an actuator command. Finally, it connects to the actuator and sends it the command.

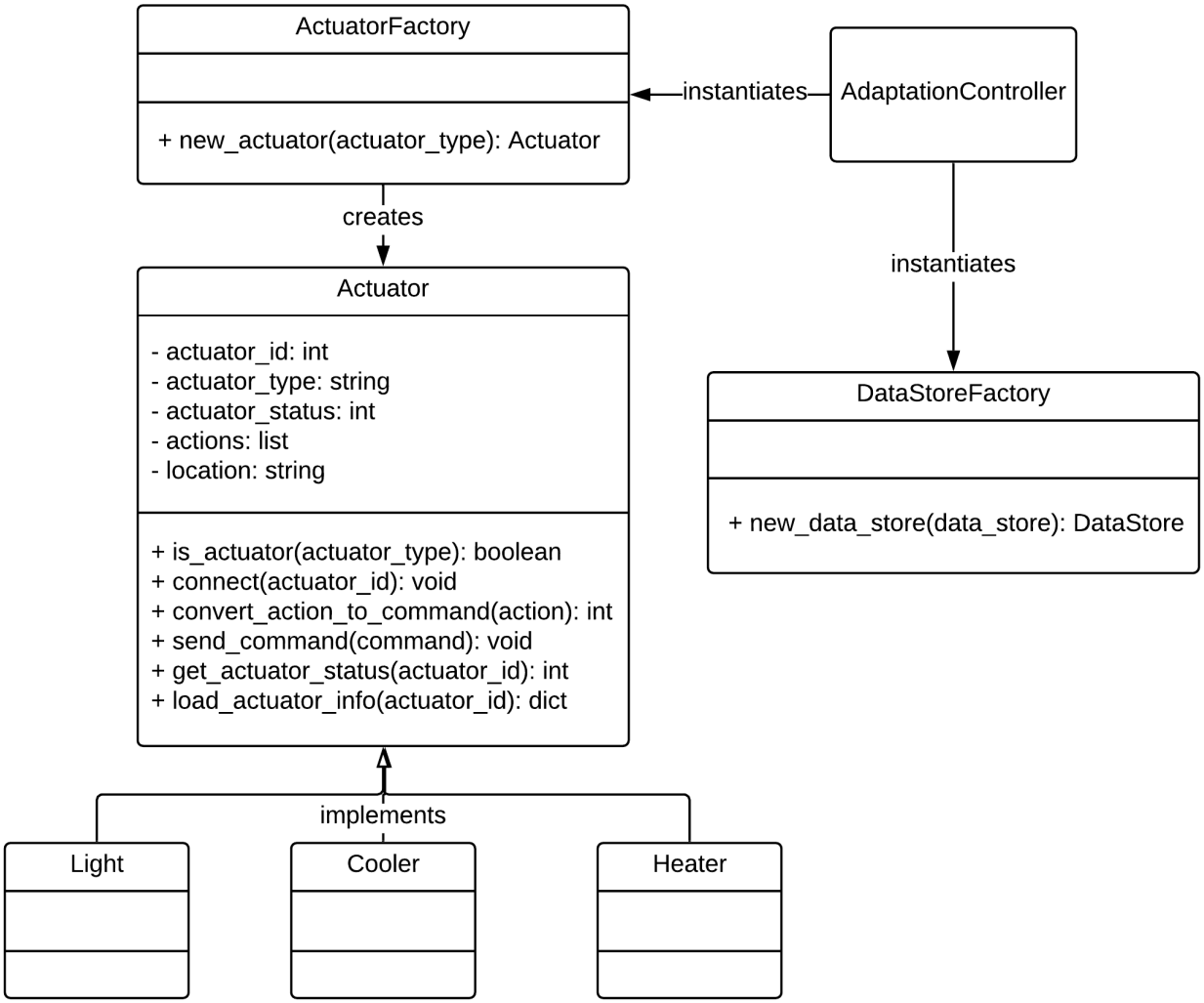

Figure 23: Adaptation Component Class Diagram

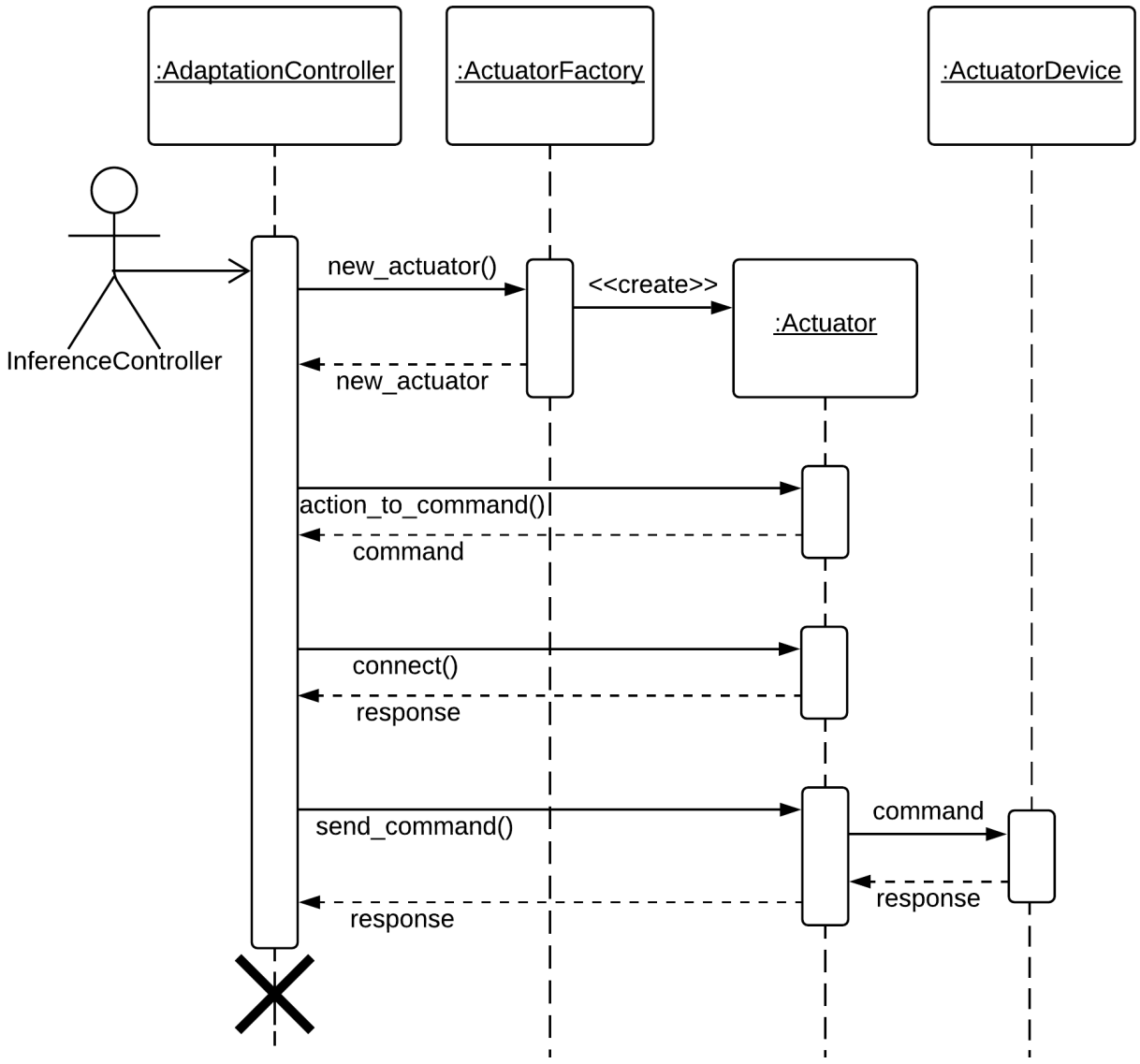

Figure 24: Adaptation Component Sequence Diagram

ActuatorFactory Class

The *ActuatorFactory* class is used to instantiate the correct actuator object using the type of the actuator. It contains one static method called *new_actuator* that takes the actuator type and location as parameters, as shown in Table 12.

Method	Description
new_actuator	This method takes the an actuator type and location as parameters and returns the appropriate *Actuator* class.

Table 12: *ActuatorFactory* Methods Description

Actuator Class

The *Actuator* class is a general class that contains variables and methods that are common to most actuator devices, as described in Tables 13 and 14. There are also specialized actuator classes that inherit from the *Actuator* base class. Examples of such classes include *Cooler*, *Heater*, and *Light*. The *ActuatorFactory* class is used to determine which actuator object to instantiate.

Variable	Description
actuator_id	ID of the actuator device.
actuator_type	Type of the actuator device.
actuator_status	Latest status of the actuator device.
actions	List of actions that the actuator can perform.
location	Location of the actuator device.

Table 13: *Actuator* Variables Description

Method	Description
is_actuator	Static method that takes the actuator type and returns true if the actuator type parameter is equal to actuator type in the class, otherwise it returns false. This method is called by the *ActuatorFactory* class.
connect	Method that takes the actuator ID and physical address and establishes a connection with it.
convert_action_to_command	Method that takes an action and an actuator type and transforms the action to a command that can be understood by the actuator.
send_command	Method that takes a command and an actuator ID and sends said command to the actuator.
get_actuator_status	Method that takes an actuator ID and pings said actuator. If the actuator responds, the status is stored as “ok”, otherwise it is stored as “down”.
load_actuator_info	Method that takes an actuator ID and returns its information from the general database.

Table 14: *Actuator* Methods Description

Adaptation Controller

The *Adaptation* controller is the unit that manages any input and output data to the adaptation component. As soon as it receives new inferred data from the inference component, it creates a new *Actuator* object using the *ActuatorFactory* class. Using the action from the inferred data, it

calls the appropriate methods from the *Actuator* class to convert the action to an actuator command and sends the command to the corresponding actuator.

5.2.5 DataStore Component

The *DataStore* component is responsible for all interactions with any data store or database. This component can be called and used by other components in the architecture to perform actions such as get, put, update, and delete on a specific data store. It contains a *DataStoreFactory* and a *DataStore* class as shown in Figure 25.

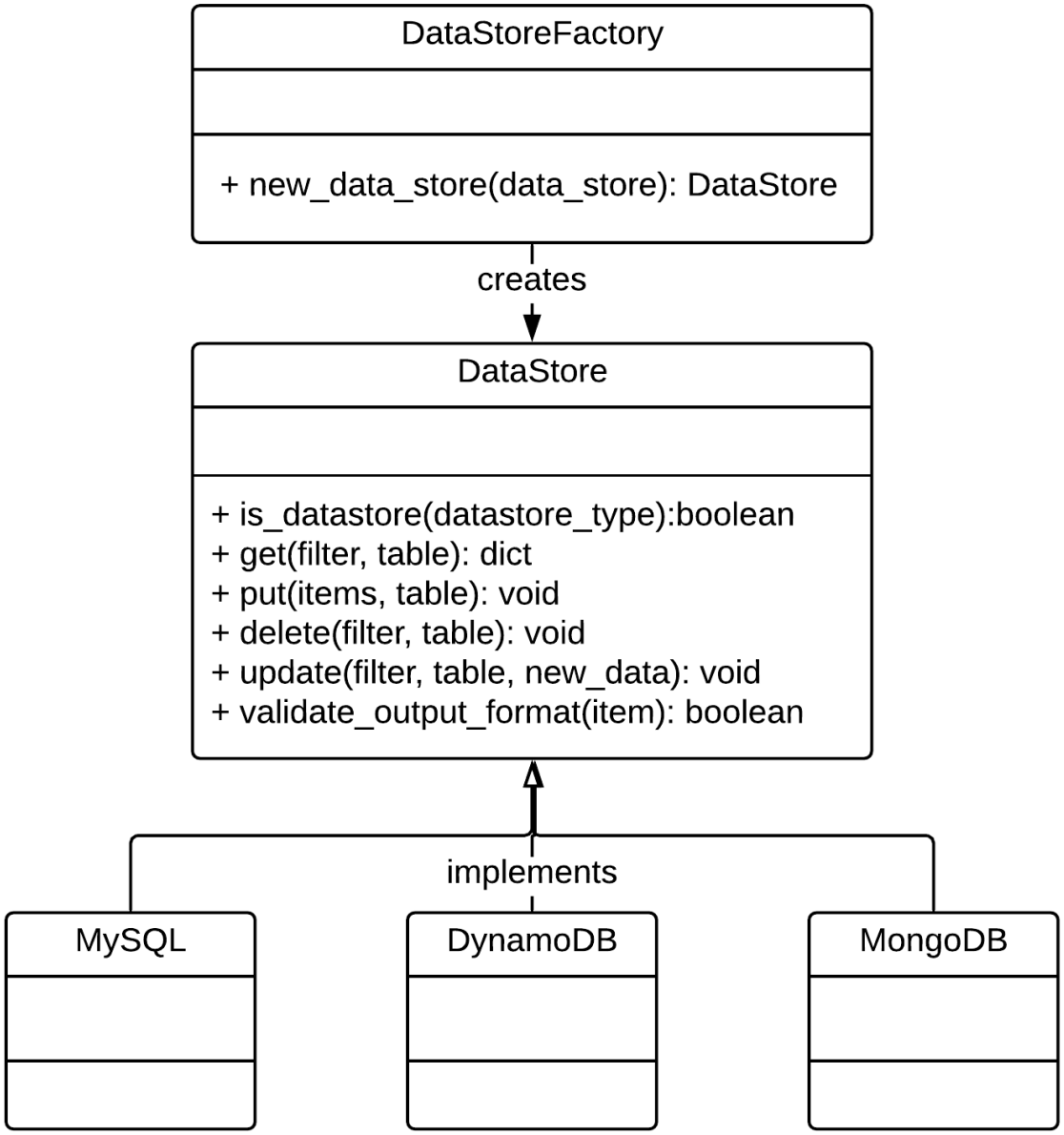

Figure 25: Datastore Component Class Diagram

DataStoreFactory Class

The *DataStoreFactory* class is used to instantiate the correct data store object using the type of the data store. It contains a static method, called *new_data_store*, which takes the data store type as a parameter, as shown in Table 15. This class uses the Factory design pattern to select the appropriate *DataStore* object to instantiate.

Method	Description
new_datastore	This method takes the data store type as a parameter and returns the appropriate *DataStore* class.

Table 15: *DataStoreFactory* Methods Description

DataStore Class

The *DataStore* class is a general class that contains variables and methods that are common to most data stores, as described in Table 16. There are also specialized data store classes that inherit from the *DataStore* base class. Examples of such classes include *MySQL*, *DynamoDB*, and *MongoDB*. The methods included in the base class allow the other components in the architecture to perform basic operations with data stores such as get, put, delete, and update. The *DataStoreFactory* class is used to determine which data store object to instantiate.

Method	Description
is_datastore	Static method that takes the data store type and returns true if the data store type parameter is equal to data store type in the class, otherwise it returns false. This method is called by the *DataStoreFactory* class.
get	Method that takes a filter and a table name and fetches the data that corresponds to the filter from the data store.
put	Method that takes the data to be stored and a table name and writes this data in the given table of the data store.
delete	Method that takes a filter and a table name and deletes the data that corresponds to the filter from the data store..
update	Method that takes a filter, the new data, and a table name and updates the data that corresponds to the filter with the new data provided.
validate_output_format	Method that takes data as a parameter and ensures that this data respects the format of the data store.

Table 16: *DataStore* Methods Description

5.3 Summary

In this chapter, we discussed the design details of our context-aware architecture in more depth. Each component was analyzed to show how its internal components communicate with each other. Also, the variables and methods for each component were presented and discussed.

Chapter 6

Implementation

This chapter provides more details on the implementation of the context-aware system for smart city applications developed in this thesis. We begin with an overview of the system requirements. We then discuss details about the implementation developed for this thesis using the requirements to justify our choices.

6.1 Context-Aware System Implementation Requirements

Before starting the implementation, a few choices had to be made regarding tools and technology that meet the implementation requirements for our system. We first identify those requirements in order to justify the choices made for the implementation.

6.1.1 Programming Language

When the time came to choose the programming language for the implementation, many languages were considered. After some initial screening, Java and Python were chosen. In order to choose one of these languages, a set of requirements was introduced.

- *Maintainability*: When choosing a programming language, it is always important to consider maintainability in order to fix critical issues, develop new features and ensure that the application evolves with time.
- *External libraries and packages*: In this implementation, the application has to interact with external components. With external libraries, these interactions can be made simpler and more efficient since the libraries are developed independently and optimized for each interaction.

- *Object-oriented capabilities*: In order to write clean and maintainable applications, object-oriented concepts and techniques are often required. This feature allows developers to decompose the code into smaller logical components using concepts such as encapsulation, abstraction, inheritance, and polymorphism, supported in object-oriented programming methods.

- *Speed*: One of the key features of the context-aware system developed for this thesis is the so-called real-time data processing. For this reason, timeliness and processing speed are both important aspects to consider when choosing a programming language.

Using the requirements described above, the languages Java and Python are compared to determine which one is more suitable for the implementation of the proposed context-aware system. When considering maintainability, both languages are proven to be suitable, however, we found Python to be better because it focuses on code "readability" by forcing indentation and using natural language (English) instead of symbols for some logical operators. Generally, a well written Python application requires less lines of code than other popular languages like Java.

Both languages allow users to download and use external libraries. In Java, a popular method of doing this is using a dependency manager called Maven. In this method, the user has to define and maintain the dependencies in an XML file. Although this is a good method, it is complex and error-prone. In Python, the use and maintenance of libraries is made simple using the "Pip Installs Packages" (PIP) package manager. PIP allows users to download and install a single library or manage a large number of libraries defined in a text file. Overall, both languages enable users to use external libraries. Java's method is more robust, while the Python method is simpler.

Object-oriented features and capabilities are important for implementation of context-aware applications, which explains why the chosen language should allow and support them. Both languages have object-oriented capabilities, although Java enforces the use of classes and some good object-oriented practices. Although Python is primarily a scripting language, it is equipped with the tools to support object-oriented programming. Most of the time, the same application can be built in both languages using object-oriented concepts.

The final requirement to consider for choosing a desired programming language is speed. When comparing both languages out of the box, Java is generally considered to be faster than Python. The reason for that is that Java is a compiled language, while Python is interpreted. That being said, Python allows the use and integration of modules written in C, allowing developers to write performance critical sections of the application in C in order to improve performance.

All in all, when analyzing the features of the proposed languages with the requirements, we found that both languages would be suitable for this implementation. However, choosing Python allowed us to come up with a working prototype in a short amount of time. The object-oriented

capabilities of Python allowed us to decompose the code into smaller logical components. Furthermore, if we encounter a feature that requires to be faster, we could simply write it in C for increased efficiency and performance.

6.1.2 Platform

After choosing the programming language, the next step was to choose a platform to run our context-aware system. The main choices were (1) to run as a monolithic application, and (2) to decompose the application into independent services and run them on the cloud. First, we define the requirements for the platform and development technique in order to justify our choice.

- *Scalability*: The system should be able to scale up or scale down, depending on amount of traffic and the load. It should only scale up if it really needs to in order to reduce operational costs. This requirement is especially important for our application because we need near real-time performance as some situations may be time critical.

- *Security*: The nature of the information used in context-aware systems makes security an important requirement. Security should be integrated in each component of the architecture, and the communication between components should only be allowed if and when needed.

- *Fault tolerance*: The application should keep running even if one of the components fail. This is important because some critical situations require that the application keeps running since otherwise it could result in undesired, and even catastrophic consequences.

- *Independent components*: Having independent components means that any developer can work on one component without having to know details about the others. This can also improve code maintainability significantly. Furthermore, components do not need to know about the implementation of other components other than the interface, that is, they only need to know which input they are receiving and what output they need to produce. Finally, developing independent components can help with fault tolerance, since the components do not depend on each other. It also makes it easier to identify the source of a failure when it occurs.

Given these requirements, we decided to implement our architecture as independent components on Amazon Web Services (AWS). AWS is a cloud platform that offers various services ranging from computing to database storage. The reasons we chose AWS are as follows:

- It offers flexibility, security, performance, and scalability.
- It has many cloud computing services that meet the technological needs of businesses and individuals. Each one of their services is independent from the others, allowing for more flexibility when designing and building an infrastructure.
- It puts a lot of emphasis on security, and they offer many services to easily manage the security of an application. In our implementation, we have used the Identity and Access Management (IAM), Cognito, and Key Management Services (KMS) services to manage security. These services will be explained in more details in the next section.
- It offers many services that are considered "serverless", which are components that are fully managed by a third-party provider. In other words, there is no need to worry about provisioning and maintaining servers. Finally, AWS offers a range of performance, where a client can pay more to receive high performance.

It is important to note that AWS is not the only platform that can be used for this implementation. In fact, any cloud provider could be used to host and run the implementation. An interesting alternative that can be used in case someone wanted to use a non-profit cloud platform is OpenStack, which is a free and open-source software platform for cloud computing. In this thesis, we use AWS for convenience and ease of use, and because it is well documented.

6.2 Implementation Details

In this section, we discuss how we leverage some AWS services in order to build a highly available, scalable, and secure serverless context-aware architecture for smart city applications. Figure 26 shows a high level architecture of the AWS implementation. To better understand this architecture, we explain the AWS services in some detail.

6.2.1 AWS Services and Architecture Description

As shown in Figure 26, there are different AWS resources and services used in the architecture. The following is a list of these services along with a description of their functionality.

- *Lambda*: AWS Lambda is a high-scale, provision-free serverless computing service. It is described as serverless because Amazon takes care of setting up and maintaining the servers on which the application runs. That is, the end users do not need to worry about the servers.

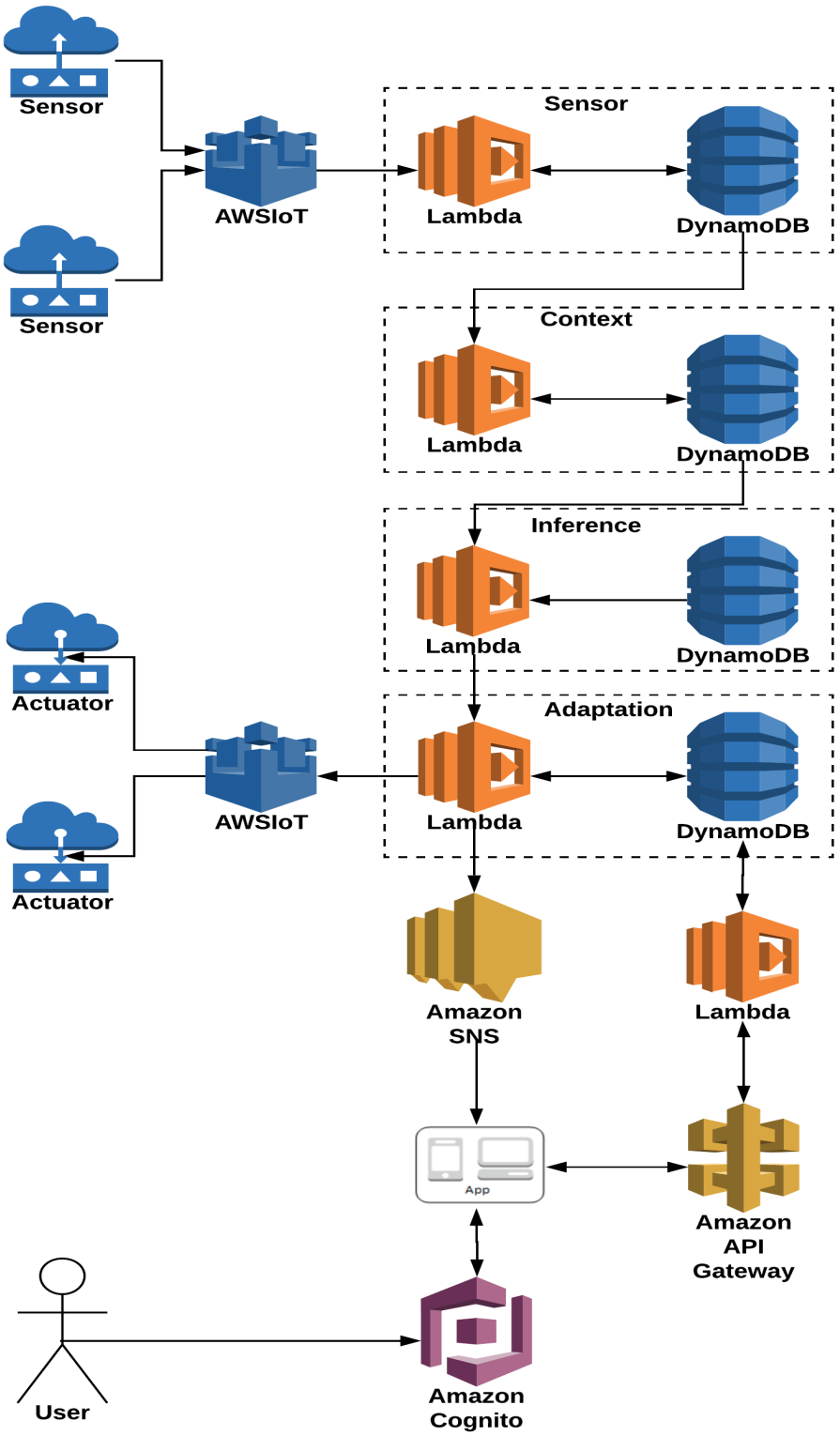

Figure 26: AWS Implementation Architecture

Lambda functions can be triggered by other AWS services such as API Gateway, Cloudwatch, and AWS IoT. Lambda functions can also be run periodically. These functions enable development of reactive, event-driven systems. Lambda functions scale well "horizontally", as more copies of the function are run in parallel when there are multiple simultaneous occurring events. AWS Lambda is used in our work to run each of the components.

- *DynamoDB*: DynamoDB is a fast, flexible, and reliable non-relational database service hosted by AWS. This database is fully managed by AWS, in the sense that users do not have to worry about hardware provisioning, setup and configuration, throughput capacity planning, replication, software patching, or cluster scaling. This can be useful for smart city applications in which scalability, speed, and reliability of the database are very important features to have. As shown in Figure 26, DynamoDB is used in the context-aware architecture developed in our work.

- *AWS IoT Core*: AWS IoT Core is a managed cloud platform that lets the connected devices interact securely and reliably with cloud applications hosted on AWS. AWS IoT Core can support billions of devices and trillions of messages, and can process and route those messages to AWS endpoints and to other devices reliably and securely [IOT]. In our work, IoT Core is used to receive data from the sensors and to send data to the actuators in the environment.

- *API Gateway*: Amazon API Gateway is a fully managed service that makes it easy for developers to publish, maintain, monitor, and secure APIs at any scale. APIs can be created to act as a point of entry to the back-end of various applications. Amazon API Gateway handles all of the tasks involved in accepting and processing up to hundreds of thousands of concurrent API calls, including traffic management, authorization and access control, monitoring, and API version management. API Gateway is used to create endpoints that will allow users to obtain required details of the context information and adaptations stored in the application data store.

- *SNS*: Amazon Simple Notification Service (SNS) is a web service that enables users to easily set up, operate, and send notifications from the cloud. It provides developers with a highly scalable, flexible, and cost-effective capability to publish messages from an application and immediately deliver them to subscribers or other applications. Amazon SNS follows the "publish-subscribe" messaging paradigm, with notifications being delivered to clients using a "push" mechanism that eliminates the need to periodically check or "poll" for new information and updates. This service is used to send notifications to the users of the context-aware system.

- *Cognito*: Amazon Cognito is an AWS services that allows developers to easily add user sign-up and authentication to mobile and web applications. Amazon Cognito also enables the authentication of users through an external identity provider and provides temporary security credentials to access back-end resources in AWS or any service behind Amazon API Gateway. Amazon Cognito works with external identity providers that support SAML or OpenID Connect, social identity providers (such as Facebook, Twitter, Amazon), or custom identity provider. This service will be used to authenticate users to the web or mobile application used in our context-aware system.

- *IAM*: The Identity and Access Management (IAM) service allows developers to securely control individual and group access to AWS resources. User identities can be created and managed, and permissions can be granted to those IAM users to access specific resources. Permissions can also be granted to users outside of AWS. Another interesting feature is the ability to create roles that contain access policies. Roles can be attached to AWS resources to control access between the various AWS resources in the architecture. This service is used to control access and communication between the various components in our architecture.

- *KMS*: AWS Key Management Service (KMS) is a managed service that lets developers easily encrypt their data. AWS KMS provides a highly available key storage, management, and auditing solution for developers to encrypt data within their applications and control the encryption of data stored across AWS services. KMS is used to manage the keys used to encrypt the databases in our context-aware architecture.

As shown in Figure 26, the architecture includes four main components, namely sensor, context, inference, and adaptation components. These components were discussed in details in Chapters 4 and 5. Each one of these components contains a Lambda function, which contains all the logic for the component, including all the sub-components and classes described in Chapters 4 and 5. Each component can be considered as an independent microservice, and can be developed and maintained on its own, i.e. independent of other components. The components also contain a DynamoDB database. In reality, this can be either separate database instances or the same instance, depending on the requirements. For the purpose of this thesis, the same database instance is used for all data storage needs, and the components use different tables in the database. The DynamoDB instance illustrated with each component also represents the general data store as well as the application data store, described in Chapters 4 and 5.

The point of entry and exit of our AWS architecture from the devices in the environment comes through the AWS IoT Core service. This component is responsible for managing all the devices in the environment securely and triggering the Lambda function from the sensor component. When

creating the Lambda function, one simply needs to configure a trigger from AWS IoT Core. Then, the Lambda function will have access to all the data coming from the sensors through the "event" variable. Once the data arrives, the Lambda function transforms, validates, and aggregates the data and stores it in the "SensorReadings" table in DynamoDB. Once the data is stored, the Lambda function from the context component, which is configured with a DynamoDB trigger, will receive an event with all the sensor data stored in the application database since the last trigger. The Lambda function in the context component will then manage, build, and store the context instances in the "Context" table in DynamoDB. This in turn will trigger the Lambda function from the inference component. This function is configured to receive triggers from the "Context" table in DynamoDB. Upon receiving the new context instances from DynamoDB, the Lambda function in the inference component proceeds to load the rules and facts from the application database and runs the inference engine. When the inference engine finishes its run and terminates, the inference Lambda function sends an event to the Lambda function in the adaptation component. The Lambda function in the adaptation component will in turn determine if an action on the environment is required, and if positive, it chooses the actuators, loads their metadata, and sends the appropriate signals to those actuators through AWS IoT Core. In addition, the adaptation Lambda function stores the results in the application database and sends a notification to SNS in case any application users needs to be notified.

From a user's perspective, a web or mobile application would be the point of entry to the system. To access the application, the user authenticates himself with Amazon Cognito. Once this is done, the user will have access to previous notifications and will be able to receive further notifications as they arrive. Also, the user will have access to the adaptation results, which will be stored in the DynamoDB instance in the adaptation component. In order to have access to this information, a GET API will be created in API Gateway. This API will point to a Lambda function that will be responsible to fetch the required data from the application database, format it, and return it to API Gateway, which will make it available to users through their web or mobile application. By doing so, a level of transparency is provided to the users, as they will know exactly what triggers notifications, and they will be able to investigate using the results stored in the application database.

6.2.2 Privacy and Security

As mentioned in the previous chapters, privacy and security are of the utmost importance when it comes to smart cities because of the sensitivity of the data being used and the impact of the consequences in case of a breach. For this reason, privacy and security must be incorporated at each step of the architecture. As mentioned in the previous subsection, Amazon Cognito will be

used for user authentication and Amazon IAM will be used to create roles and policies in order to control the communication between the various components in our proposed architecture.

User Authentication

Amazon Cognito enables developers to easily setup, sign-up, and authenticate themselves using a custom identity for web and mobile applications. Amazon Cognito also enables the synchronization of data across many user devices. So if a user performs an action on the mobile application, the resulting changes would be applied to the web application if there was one.

In addition to providing a secure and scalable authentication solution, Amazon Cognito offers the ability to set up password policies and Multi-Factor Authentication (MFA). Password policies may include characteristics such as password length, types of characters, and password duration. Setting up password policies provides an additional layer of security and protects users by forcing them to follow best practices. Configuring MFA in Amazon Cognito means that users need to sign in using their password and some automatically generated code. This can be done using SMS; users provide their phone numbers when signing up, and each time they sign in, they receive a temporary code via SMS, which they will then use to access the application. Using MFA for authentication provides users with more protection in the event of a password compromise since accessing the application requires an additional step.

Access Control

Access control will be enforced using the Amazon Identity and Access Management (IAM) service. Several roles are created for the components in the architecture, each role having the necessary policies to allow the component to perform its tasks. These policies act as permissions, controlling access and specific actions between components in the architecture. In what follows, we give an example in Figure 27 to illustrate how IAM roles function in AWS.

The role shown in this example is used by a Lambda function to read data from DynamoDB. The role is divided into two sections, namely a version and a statement section. The version section is hardcoded and does not need to be changed. The statement section is where the role is defined. It is further divided as follows:

- *Action*: The “Action” attribute represents a list of actions that are associated with this role. It is written as “resource:action”. In the case of the example illustrated in Figure 27, the only resource present is DynamoDB, and the actions included are BatchGetItem, DescribeTable, GetItem, ListTables, Query, and Scan.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "dynamodb:BatchGetItem",
        "dynamodb:DescribeTable",
        "dynamodb:GetItem",
        "dynamodb:ListTables",
        "dynamodb:Query",
        "dynamodb:Scan"
      ],
      "Effect": "Allow",
      "Resource": "arn:aws:dynamodb:us-east-1:465334783815:table/Sensor"
    }
  ]
}
```


Figure 27: IAM Role Example

- *Effect*: The "Effect" attribute determines whether to allow or deny the specified actions on the resource specified.
- *Resource*: The “Resource” attribute specifies which resource or resources the role is applied to. The resource is specified using its Amazon Resource Name (ARN). In the example illustrated above, the resource specified is the Sensor table in DynamoDB.

Roles can be customized and tailored to fit specific application and business needs. They also allow for an efficient and secure way to control access between components in a system.

Confidentiality

In order to protect user data and ensure its confidentiality, the databases in the architecture must be encrypted. One of the important features of DynamoDB is that it allows developers to enable fully managed encryption at rest. This is done using the keys stored in AWS Key Management Service (KMS). This encryption protects all the data in the database, including the primary keys, indexes, streams, and backups. This allows the application to be compliant to various policies and regulations enforced by governments and users.

Encryption of DynamoDB instances can be done with the help of AWS KMS. KMS manages the keys that are used to encrypt the databases at rest. In order to encrypt data in a DynamoDB table, the developer must enable encryption when creating the table. The developer must then choose a customer master key (CMK) to encrypt the table. The first type of key is the AWS CMK,

which is owned and managed by AWS. The advantage of this type of key is that developers do not have to worry about the key at all, as AWS takes care of the encryption. However, this also means that developers have no control over the CMK or the encryption. The second type of key is customer managed CMK. This type of key is created, managed, and owned by the developers. This gives developers full control over the key, making them responsible for ensuring that any policies are up-to-date and proper cryptographic rotation is put in place. The last type of key is the AWS managed CMK. These are keys that are created and managed by AWS, but developers can see these keys and audit them if needed.

It is important to note that when a component or a user with appropriate permission accesses an encrypted table, DynamoDB takes care of the decryption, without any changes needed by the developers. Furthermore, encrypting tables does not affect the database performance, and all the queries and transactions work the same way on encrypted tables as they do on non encrypted tables.

6.3 Summary

In this chapter, we discussed implementation details of the context-aware architecture proposed in this thesis. The requirements for the features of desired programming language and platform were analyzed, based on which we selected Python to develop the implementation, and selected AWS as the platform. The various AWS resources and services used in this implementation were presented and discussed in order to get a better understanding of the overall cloud architecture. In this implementation, a special emphasis was put on privacy and security features, since the type of information that can be used in smart city applications is sensitive. The source code for the implementation can be found at https://github.com/ZakiChammaa/aws-context-aware-implementation. In the next chapter, a case study will be presented in order to better understand the proposed architecture and to show the implementation at work.

Chapter 7

Case Study

In this chapter, we introduce a case study to showcase the implementation discussed in the previous chapter. The case study involves efficient configuration control for smart rooms. Many users with different preferences can use the rooms, and the context-aware system intelligently determines which preferences are to be applied in different contexts in the rooms.

7.1 Smart Room Configuration Control

A general description of the problem is as follows:

> A Smart Building (SB) is required to have a finite number of Smart Rooms (SR), where each SR has to automatically configure its *comfort level* according to the *preferences* of its occupants subject to certain *constraints* (policy) imposed by the *Supervisory System* (SS) that manages the SB.

We first provide an abstract view of SR settings in the SB, and emphasize on the role and responsibilities of SS in creating such views. Next, we introduce the specific instance of the problem solved and provide the solution steps that bring out the essential aspects of "smartness". Finally, we give an overview of possible generalizations to the algorithms in this approach and motivate how solutions to different conceptualizations of the general problem can be realized.

The SS determines, after requirements gathering and analysis, the set of sensors and actuators to be installed in different rooms of SB. They in principle determine the *comfort category* of every room. As an example, if a temperature sensor and a humidity sensor are installed in a room that communicate respectively with Thermostat and Humidifier actuators, then the comfort category of this room is defined by the range of temperature and humidity values that these actuators can handle. Assuming that each actuator is uniquely associated with a sensor we can restrict our discussion abstractly to the *comfort attributes* (features) associated with the services of sensor

types. Thus, the room with Thermostat and Humidifier sensors has the set of comfort attributes $\{Temperature,\ Humidity\}$. By putting together such comfort attributes for all SRs in a SB, the SS creates the set $CA(B) = \{a_1, a_2, \cdots, a_n\}$ of comfort attributes for a building SB. This set is predetermined by SS.

In a SB, some SRs may have only one comfort attribute, some SRs may have two comfort attributes, and in general the set of comfort attributes for a SR in SB is a non-empty subset of $CA(B)$. Thus, the total collection of possible comfort attributes for all rooms in SB is $\mathcal{CA}(\mathcal{B})$, the set of all subsets of $CA(B)$. It is well known that $\mathcal{L}(\mathcal{B}) = \{\mathcal{CA}(\mathcal{B}), \subset, \cup, \cap\}$ is a complete lattice, if we let the minimum element of the lattice be $\emptyset$. Thus, the set of comfort attributes for any room SR in SB is a node in this lattice. Knowing the range of values that a sensor can generate (and the actuator capabilities), the SS can associate for each attribute $a_i \in CA(B)$ a unique ordered set $V(a_i)$ of atomic values. That is, in the presentation of set $V(a_i)$ the atomic values are listed in increasing order, as in $V(Temperature) = \{18, 19, \cdots, 26\}$. By uniqueness we mean $V(a_i) \cap V(a_j) = \emptyset$, for $i \neq j$. The set $V(a_i)$ defines all possible *comfort levels* that an attribute $a_i \in CA(B)$ can achieve. The node $s = \{a_{i1}, \cdots, a_{ij}\}$ in the lattice $\mathcal{CA}(\mathcal{B})$ that is associated with a room SR can achieve the collection of comfort levels for the room SR defined by the set

$$SRL(SR) = \{s(x_1, x_2, \cdots .x_j) | x_1 \in V(a_{i1}), \cdots, x_j \in V(a_{ij})\}$$

Thus the lattice generates all possible comfort levels for the rooms in B. Because of this closure property, every smart adaptation in every room can be executed by an actuator in that room.

7.1.1 Role and Responsibilities of SS

The role of SS is "technical and administrative manager of SB". From gathering requirements, acquiring devices, setting up each SR in the building with sensor/actuator networking, and defining the roles and policies of SR users in SB, the SS takes a crucial role in the safe functioning of the SB. Below are the primary responsibilities of SS. The adaptation mechanism that we discuss later in this section assumes that these roles are fulfilled.

- Room Comfort Level: The SS assigns a unique ID for each SR, defines its location, and assigns the set of sensors/actuators for each SR. In addition, the SS defines the sets $CA(B)$, $CA(SR)$ for every SR in SB, and SRL(SR) for every room in SB.

- Sensor/Actuator Level: The SS provides IDs for each device, and precisely specifies the services provided by each device. In particular, the SS specifies for each sensor its type, the range of values, and the units of measurements, all of which will be used for validating the correctness of the data. Whenever there is a modification in sensor/actuator configurations

or capabilities in a SR, the sets $CA(SR)$ and $SRL(SR)$ are redefined and uploaded by the SS for dynamic adaptation.

- People Level: The SS provides an ID and assigns a *Security Level Clearance* (SCL) for each individual registered in the SB (SCL may be dependent on the role played by the person). The SS also defines the set of SRs that can be accessed by each individual.

- Policy Level: The SS defines "Configuration Control Policies" that are used by every SR adaptation system. A policy is a rule that is either *global* to all rooms in the building B or is *local* to each SR. For each room SR, the SS grants to each person p a "Preference Comfort Choice" that can be met during adaptation, using $SRL(SR)$. An adaptation of comfort level in SR is enforced by applying these policies in dynamically changing local *context*s.

7.2 Specific Instance: Problem Statement

We assume that the comfort attributes for all the rooms in SB are "temperature, lighting, and window shades". For the sake of clarity of presentation we consider all rooms in SB to have all the three attributes and there are no global policies. We explain later how the algorithms in the current solution can be modified easily to satisfy solutions to several generalizations. We require every SR in SB to adjust the room temperature, lighting, and window shades automatically according to the preferences of the occupants of the room subject to the following constraints:

- C_1::*Initial Configuration:* Room is empty, lights are not on, window shades are down (closed) and the room temperature is 22 degrees.

- C_2::*Dynamic Configuration:* Room is not empty. The room temperature, lighting, and window shades are to be configured according to the preferences of the *highest authority* inside the room. The person of highest authority is one who has the highest SCL granted by the SS. More than one person may hold the same SCL. In the case that more than one person holds the highest SCL in the room, the preferences of the person who was in the room first are to be applied.

This is a simple case study because of the small number of sensors, actuators, and rules for adaptations, regardless of the number of persons inside the room at any one instant and the room size. We discuss in the following sections (1) modeling the problem, (2) user authentication protocols for "entry" to SR and "exit" from SR, and (3) adaptation algorithms for achieving room comfort level according to constraints C_1 and C_2.

7.3 Abstract Modeling of Specific Example

Every SR has a unique ID, and has only one door that is used for entry as well as exit. Every sensor is abstracted by its ID, its type, the unit of measurement, and the minimum and maximum values transmitted by it. Every actuator has a unique ID and a type. Outside the door to the room SR is a sensor IN, and inside the frame of the door is a sensor OUT. They monitor the entry into the room and exit from the room of a person who has a unique Smart Card (SC) in which the ID and SCL of the person are embedded. In this case study, the set $CA(SR)$ consists of the attributes *"Temperature"*, *"Light"*, and *"Drape"*. The predefined sets of comfort levels (the Vs defined in Section 7.1) for these attributes, regarded as "enumerated types", are given specific names below(for readability):

- *Temperature Comfort Levels:* $TCL = \{18, 19, \cdots, 26\}$, the range of integers in the range 18 to 26. It defines the minimum and maximum temperatures allowed in the room.
- *Lighting Comfort Levels:* The three levels set by SS: $LCL = \{Low, Medium, High\}$
- *Drape Comfort Levels:* The three levels set by SS: $DCL = \{Down, Middle, Up\}$

The default comfort level of SR is $\langle 22, Low, Down\rangle$. The set of all possible comfort levels is $SRL(SR) = \{\langle x, y, z\rangle | x \in TCL, y \in LCL, z \in DCL\}$. Inside the room are three actuators TA, LA, and DA. The actuator TA is to execute temperature commands, the actuator LA is to execute lighting commands, and the actuator DA is to execute drape commands. The execution of a command takes some time, which is largely implementation dependent. In our specification of system behaviour we can abstract away the time and assume synchronization of commands.

7.3.1 Data Stores

In the architecture discussed in Chapter 4, we have used two data stores, called *Application Data Store* (ADS) and *General Data Store* (GDS), to manage sensor readings, contexts, facts and rules pertaining to a specific application analysis. To support user authentication and adaptation, we organize the information supplied by the SS in the following tables:

- Sensor table is maintained in GDS and it contains metadata on the sensors used by the application. The attributes of this table are $\{SensorID, SensorType, RoomID, MinValue, MaxValue, Unit\}$ This table is used to transform and validate sensor readings.
- Actuator table is maintained in GDS and it contains metadata on the actuators used by the application. The attributes of this table are $\{ActuatorID, ActuatorType, RoomID\}$. By

associating $SensorID$ and $ActuatorID$ with $RoomID$, we model precisely those comfort attributes that a room can have.

- Action table is maintained in GDS and it contains all the actions that the actuators defined in the database can perform. The attributes of this table are $\{ActuatorID, Action, Command\}$. The "Action" attribute refers to the action name, while the "Command" attribute refers to the actual command that needs to be sent to the actuator. Thus, the set of comfort levels that a room can achieve are defined by the actions (capabilities) of actuators in that room.

- User table is maintained in ADS and it contains information related to the users of the application. For each user, the SS associates a $USERID$ and SCL. Hence, the attributes of this table are $\{UserID, SCL\}$.

- Room table is maintained in ADS and is used to determine if a user has access to a specific room. The attributes of this table are $\{RoomID, UserID\}$.

- Preferences table is maintained in ADS and it contains the comfort level preferences for each user for all rooms where he/she is allowed access. The attributes of this table are $\{UserID, RoomId, Temperature, Light, Drape\}$.

Table 17 summarizes the information about the tables presented above and describes metadata on those tables in order to better understand the purpose of each table and to see which component can request data from each table.

Table Name	Data Store	Purpose	Component Access
Sensor	GDS	Store metadata on sensor devices. Used to transform and validate sensor readings.	Sensor
Actuator	GDS	Store metadata on actuator devices.	Adaptation
Action	GDS	Store actions and commands for each actuator. Used to determine actuator command based on action.	Adaptation
User	ADS	Store user ID and SCL for all users. Used when determining room comfort level.	Inference
Room	ADS	Store RoomID and UserID for each room. Used to determine which users are authorized to access specific rooms.	Inference
Preferences	ADS	Store UserID, RoomID, and comfort level attributes for all users for each authorized room. Used to determine adaptations based on user preferences.	Inference

Table 17: Metadata on Tables in ADS and GDS

7.4 The Dynamics of the Specific Example

The three stages in the dynamics of this example are *Initialization*, *Authentication*, and *Adaptation*. For each stage, we informally describe the interaction of the components in the architecture illustrated in Figure 12 from Chapter 4 and give a precise specification of the algorithmic steps. In the specification, we use the key words *ActiveUsers* to refer to the set of people inside a SR at any instant, *Adaptations* to refer to the comfort level that is defined by the system policy (Constraint C_2), C to denote the context information, $SUID$ to denote the ID of the person of highest authority in the room SR, and *Configure* to denote the command of the actuator that will execute the action achieving the adaptation.

7.4.1 Initialization

Initially the SR is empty. The context component sets $ActiveUsers$ to empty set, and constructs a context instance following step 2 from the *INITIAL* algorithm and stores it in the Context table in ADS, which is used to store active context information. This in turn sends a trigger to the inference component, which determines that the preference to be chosen is the default one. It triggers the adaptation component so that it can configure the room comfort with default values $\{22, Low, Down\}$.

INITIAL

1. $ActiveUsers \leftarrow \emptyset$
2. $C_{init} \leftarrow \{Null, RoomID, Time\}$
3. $Adaptation \leftarrow Preferences(\emptyset, RoomID)$
4. $Configure \leftarrow \{22, Low, Down\}$.

$Configure$ will communicate the preferences to the actuators whose IDs are in the Action Table.

7.4.2 Authenticating a User

In order to grant entry only to authorized persons into a SR, the SS creates a smart card $SC(p) = \langle uid, scl \rangle$ for each person. The SC consists of the identity uid and the security level clearance scl assigned to person p registered in the system. The sequence diagram, shown in Figure 28, informally explains the authentication steps. A more formal authentication algorithm follows it.

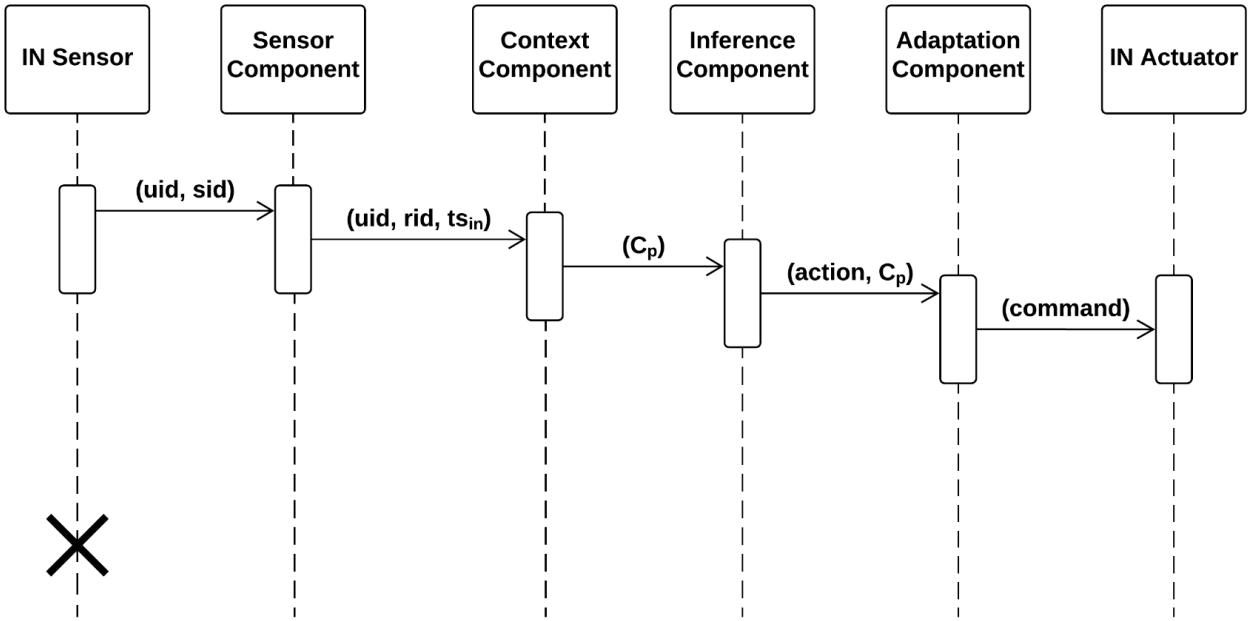

Figure 28: Authentication Procedure

AUTHENTICATE-ENTRY(p)

1. *Sensor IN:* Sensor IN, installed outside the room SR, scans $SC(p)$ of a person p and copies $Id.SC(p)$ in uid. It constructs the tuple $\alpha = \langle uid, sid \rangle$, where sid is its unique identity, and forwards the tuple α to the *Sensor Component*.

2. *Sensor Component:* It uses sid in α to determine the identifier rid of the SR from the $Sensor(SensorID, RoomID)$ table. It constructs the tuple $\beta = \langle uid, rid, ts_{in} \rangle$, where ts_{in} is the timestamp of receiving the tuple α from the sensor *IN*, and sends it to the *Context Component* through the ADS.

3. *Context Component:* It constructs the context for the authorization request in the format explained in Chapter 5. Using β it constructs the context $C_p = [UserID : uid, RoomID : rid, Entry\_Time : ts_{in}]$ and sends it to *Inference Component* through the ADS.

4. *Inference Component:* It searches the $Room$ table in the ADS for the tuple $\langle uid, rid \rangle$. It sends the tuple $\langle Grant\_in(uid, rid), C_p \rangle$ to the *Adaptation Component* and the *Application Database Controller* (ADL) if $\langle uid, rid \rangle$ is found in the $Room$ table, otherwise it sends the tuple $\langle Deny\_(uid, rid), C_p \rangle$ to the *Adaptation Component*.

5. *Adaptation Component:* It searches the $Sensor$ table in the GDS and selects the sids corresponding to the rid. It then selects the actuator identifier aid corresponding to the sid in room rid from the $Actuator$ table, selects the command of this actuator that is equivalent to

the received message from the $Action$ table, and sends it to the actuator for execution. Thus, if $Grant\_in(uid, rid)$ was received, the door opens to let person p enter the room, otherwise entry is denied to person p.

7.4.3 Adaptation

Adaptation is required whenever a user p enters or exits the room SR. We use $Adaptation$ protocols *ADAPT-ENTER(p)* and *ADAPT-EXIT(p)* respectively for entering and exiting the room. The adaptation policies are the constraints C_1 and C_2. Tables *Users*, *Sensors*, *Actuators*, *Actions*, and *Preferences* (constructed by SS for all the registered users in the system) are used by the adaptation algorithms. The interactions of components and the their actions are described in the protocols. We use the function $find(rid, ActiveUsers)$ in the specification of both *ADAPT-ENTER(p)* and *ADAPT-EXIT(p)* below. This function returns the user identity wid of user in the set $ActiveUsers$ in room rid who has the highest SCL and entered the room earliest.

SPECIFICATION OF FUNCTION $find(rid, ActiveUsers)$
The pre-condition for invoking this function is $ActiveUsers \neq \emptyset$. The *Inference Component* computes the subset of $ActiveUsers$ who all have the highest SCL in the user $User$ table, and then computes from this set the identity of the person who entered the room earliest, as described below.

- Compute $X = \{\langle uid, ts_{in}\rangle | \langle uid, rid, ts_{in}\rangle \in ActiveUsers\}$. This set extracts partial information from the set $ActiveUsers$ who are in room rid.
- Compute $Y = \{\langle vid, ts_{in}\rangle | \langle vid, ts_{in}\rangle \in X, \langle vid, scl\rangle \in UserTable\,, \langle xid, scl'\rangle \in UserTable, xid \neq vid, scl > scl'\}$. This set consists of all active users in room rid who have the same maximum security clearance level.
- $find \leftarrow wid$, where $\langle wid, ts_{min}\rangle \in Y$, $\langle vid, ts_{in}\rangle \in Y, ts_{min} < ts_{in}$

ADAPT-ENTER(p)

1. *ADL:* Upon receiving $\langle Grant\_in(uid, rid), C_p\rangle$ from the *Inference Component* it triggers the following database operation:
   - *Update* $ActiveUsers$: From the message $\langle Grant\_in(uid, rid), C_p\rangle$ received from the *Inference Component*, the ADL extracts uid, rid, ts_{in}, and sets $ActiveUsers \leftarrow ActiveUsers \cup \{uid, rid, ts_{in}\}$. That is, the $ActiveUsers$ table in the database is updated.

- *Apply Constraint* C_2 : The ADL triggers the function $find(rid, ActiveUsers)$, which returns the user identity wid of users in $ActiveUsers$ who has the highest SCL and entered the room earliest.
- *Select Preferences:* It selects the preference tuple $pref_{wrid} = \langle tel, lil, drl \rangle$ corresponding to the tuple $\langle wid, rid \rangle$ from the $Preference$ table, and sends it to *Adaptation Component*. In this case, tel corresponds to the temperature level attribute, lil to the light level attribute, and drl to the drape level attribute.

2. *Adaptation Component*: It performs the adaptation by invoking the appropriate set of actuators and their commands:

   - *Create Adaptation:* $Adaptations \leftarrow pref_{wrid}$.
   - *Apply configurations based on adaptations:* $Configure \leftarrow Adaptations$. It searches the $Sensor$ table in the GDS and selects the sids corresponding to the rid. It then selects the actuator identifier aid corresponding to each sid in room rid from the $Actuator$ table, selects the command of this actuator that is equivalent to the received message from the $Action$ table, and sends it to the actuator for execution.

We need a slightly different protocol for authenticating exit from SR, just to ensure that every one who exits from a SR was indeed authenticated to enter the room. This is to prevent a group of people from entering SR at the same instant by the fact that one member of the group was authenticated. The exit authentication protocol is described below.

AUTHENTICATE-EXIT(p)

1. *Sensor OUT:* Sensor OUT, installed inside the room SR, scans $SC(p)$ of a person p and copies $Id.SC(p)$ in uid. It constructs the tuple $\alpha = \langle uid, sid \rangle$, where sid is its unique identity, and forwards the tuple α to the *Sensor Component*.

2. *Sensor Component:* It uses uid in α and determines the identifier rid of SR from the *Sensor(SensorID,RoomID)* table. It constructs the tuple $\beta' = \langle uid, rid, ts_{out} \rangle$, where ts_{out} is the timestamp of receiving the tuple α from the sensor *IN* and sends it to the *Context Component* through the ADS.

3. *Context Component:* It constructs the context of authorization request in the format explained in Chapter 5. Using β' it constructs the context $C'_p = [UserID : uid, RoomID : rid, Exit\_Time : ts_{out}]$ and sends it to *Inference Component* through the ADS.

4. *Inference Component:* It searches the $ActiveUsers$ table stored in ADS for the tuple $\langle uid, rid, \star \rangle$, where $\star$ is a "don't care" condition. It sends the tuple $\langle Grant\_Out(uid, rid), C'_p \rangle$ to the *Adaptation Component* and the ADL if $\langle uid, rid, \star \rangle$ is found in the $ActiveUsers$ table, otherwise it sends the tuple $\langle Deny\_out(uid, rid), C'_p \rangle$ to the *Adaptation Component*.

5. *Adaptation Component:* In case $\langle Grant\_out(uid, rid), C'_p \rangle$ is received, it searches the $Sensor$ table in the GDS and selects the sid of OUT corresponding to rid. It selects the actuator identifier aid corresponding to the sid in room rid from the $Actuator$ table, selects the command of this actuator that is equivalent to the received message from the $Action$ table, and sends it to the actuator for execution. Thus, if $\langle Grant\_Out(uid, rid), C'_p \rangle$ was received, the door opens to let the person p exit the room, otherwise exit is denied to the person p.

ADAPT-EXIT(p)

1. *ADL:* Upon receiving $\langle Grant\_out(uid, rid), C'_p \rangle$ from the *Inference Component*, it triggers the following database operation:

   - *Update* $ActiveUsers$: From the message $\langle Grant\_out(uid, rid), C'_p \rangle$ received from the *Inference Component*, it extracts $\langle uid, rid, ts_{in} \rangle$, and sets $ActiveUsers \leftarrow ActiveUsers \setminus \{uid, rid, ts_{in}\}$. That is, $ActiveUsers$ table in the database is updated.
   - *Apply Constraint* C_1: If $AciveUsers = \emptyset$, then it sends the default adaptation preference $pref = \langle 22, Low, Down \rangle$ to the *Adaptation Component*.
   - *Apply Constraint* C_2: If $AciveUsers \neq \emptyset$, it triggers the function $find(rid, ActiveUsers)$, which returns the user identity wid of the user in $ActiveUsers$ who has the highest SCL and entered the room earliest.
   - *Select Preferences:* It selects the preference tuple $pref_{wrid} = \langle tel, lil, drl \rangle$ corresponding to the tuple $\langle wid, rid \rangle$ from the $Preference$ table, and sends it to *Adaptation Component*. In this case, tel corresponds to the temperature level attribute, lil to the light level attribute, and drl to the drape level attribute.

2. *Adaptation Component*: It performs the adaptation by invoking the appropriate set of actuators and their commands:

   - *Apply Adaptation:* $Adaptations \leftarrow pref_{wrid}$.
   - *Apply configurations based on adaptations:*
     $Configure \leftarrow Adaptations$. It searches the $Sensor$ table in the GDS and selects the sids corresponding to the rid. It then selects the actuator identifier aid corresponding to each sid in room rid from the $Actuator$ table, selects the command of this actuator

that is equivalent to the received message from the $Action$ table, and sends it to the actuator for execution.

7.5 Extensions

In this section, we discuss two extensions to the case study. The first extension allows different comfort attributes for different rooms, and permits users to choose a partial subset of comfort attributes of a room for their preferences. The second extension allows the application of global policies by the SS.

7.5.1 Different Comfort Attributes and Partial Set of Preferences

In the case study, we assumed that all the rooms had the same set of comfort attributes, and all users choose preferences based on all these attributes. It can be easily extended to handle smart room configuration when not all rooms have identical comfort attributes, and users can choose a partial subset of a smart room comfort attributes as their preferences. As an example, two rooms $R1$ and $R2$, respectively may have comfort attributes $\{Temperature,\ Light,\ Drape\}$, and $\{Temperature,\ Light,\ Humidity\}$. The selected sets of preference for a user p with respect to $R1$ can be $\langle 20, \star, Down\rangle$, and with respect to $R2$ the preferences can be $\langle \star, Medium, \star\rangle$.

The following sets are precomputed by the SS.

1. For each room SR with ID rid, the set $CA(rid)$
2. For the set $CA(B) = \bigcup_{rid} CA(rid)$, which is the set all comfort categories for the building
3. For each room rid, the set $SRL(rid) = \{s(x_1, \cdots, x_k) | a_i \in CA(rid), x_j = V(a_j), j = 1, \cdots, k\}$

Internally, the attributes in $CA(B)$ define the comfort attributes in a table. To specify $CA(rid)$ for a room, we use "NULL" values for the attributes that are not available in that room. Because the SS creates $CA(SR)$ for each user, the SS will ensure that users don't assign values to comfort attributes that are not available in a specific room. To better understand how this works, consider the above example with rooms $R1$ and $R2$. $R1$ has the comfort attributes "Temperature", "Light", and "Drape", and $R2$ has the comfort attributes "Temperature", "Light", and "Humidity". Suppose that the building has one registered user with user ID uid. Suppose that for $R1$, the user selects the value "23" for "Temperature" attribute, the value "Medium" for "Light" attribute, and omits to provide a value for the "Drape" attribute. Assume that for room $R2$, the user's selected preferences

are "23" for "Temperature" attribute, "Medium" for "Light" attribute, and "45" for "Humidity" attribute. In the data store tables, all the comfort attributes in $CA(B)$ are used as attributes of the comfort table, and the comfort attributes that are unavailable for a given room will be marked as "NULL", as shown in Table 18. The internal comfort level is constructed with the attributes $\langle Temperature, Light, Drape, Humidity \rangle$, the union of the attribute sets of rooms $R1$ and $R2$. With this change in the data store, all the algorithms presented in the previous sections will work with no modification. Although the adaptation for a room SR is now dependent on the room comfort $CA(SR)$ and preferences of user p for room SR, the *Select Preferences* step in $ADAPT-ENTRY(p)$ (and $ADAPT-EXIT(p)$) will select the preferences of user p (wid) with respect to the room rid of SR. If any attribute preference is not specified by the user, the default preference value for that attribute will be substituted at the database level by the SS, before the preferences are sent for adaptation.

UserID	RoomID	Temperature	Light	Drape	Humidity
uid	R1	23	Medium	Down	Null
uid	R2	23	Medium	Null	45

Table 18: Example Preference Table

7.5.2 Supervisory System Policies

For clarity of illustration, we assume that a policy is simple in the following sense.

- All policies must be consistent, meaning that no two policies can contradict each other.
- A policy mentions preferences on one or more comfort attributes for one or more rooms.
- A policy cannot include negation.
- A policy may include time constraints (durations).

Some sample policies that are considered for extending our case study are the following.

- *Evening Policy:* Every day between 7 PM and 7 AM, the window shades in all SRs are brought down. This policy applies to all the SRs that have drapes.
- *Cost Effective Policy:* The lights in a SR are always off when very bright natural light shines through the windows. User preferences on "Light" are taken into account only during night time or on cloudy days. This policy applies to all the SRs that have a light actuator as well as a window.

In general, a policy (with or without explicit time constraints) from SS is applied by "event-driven" mechanism. If no explicit time is specified in the policy, the SS can trigger an event to "start" applying the policy and trigger another event to "terminate" the application of that policy. For applying a policy in which time constraints are explicit, the SS can "start" and "terminate" through "clock-triggered" events at the specific times. The SS may wish to apply either one policy or more than one policy at disjoint time intervals. Consequently, we consider two cases for adaptations. The SS assigns itself the identification sid with the highest SCL at all times for all rooms.

Case 1: Apply one Policy for one or more SRs

The SS broadcasts the event "start" to all rooms in which a policy has to be applied, and then adds its ID $suid$ to the set $ActiveUsers$ in each of those room. This will trigger the context-aware system to perform the steps described in the previous sections for each room. The context-aware system determines that $suid$ has the highest authority in the SR and applies those preferences for each room. It is important to note that a policy may only apply to a subset of a room's comfort level attributes. Consequently, when a user either enters or leaves a room when the SS policy is in effect, only the comfort level attributes that are not affected by the SS policy must be configured according to constraints C_1 and C_2 described in Section 7.2. When the SS policy needs to be terminated, the SS broadcasts the "terminate" event to remove the $suid$ from the set of $ActiveUsers$ in all the affected rooms. The adaptation on exit of $suid$ will be according to the algorithm described earlier. This mechanism works whether or not the policy has explicit time constraints.

Case 2: Apply more than One Policy on one or more Rooms

When more than one policy needs to be applied on a subset of the SRs in SB, there are three scenarios that might occur. These may be described either by using explicit times or temporal moments *before* and *after*. Since they lead to the same conclusions, we use explicit times for "start" and "terminate" times for the policies. Let policy P_1 start at time t_1 and end at time t_2, and policy P_2 start at time t'_1 and end at time t'_2.

Case 2.1 $t_1 < t_2 < t'_1 < t'_2$ or $t'_1 < t'_2 < t_1 < t_2$ (Figure 29): The policies are applied sequentially. In the case where P_1 is applied before P_2, we must consider "atomicity" in the implementation. That is, the SS starts by applying P_1 by adding a super user with ID $suid$ to the set $ActiveUsers$ in the rooms affected by the policies (at its local clock time t_1) using the method described in "Case 1". The SS removes the user with ID $suid$ from the set of $ActiveUsers$ in the affected rooms to terminate the policy (at its local clock time t_2). Then the SS resets its local clock to apply policy P_2.

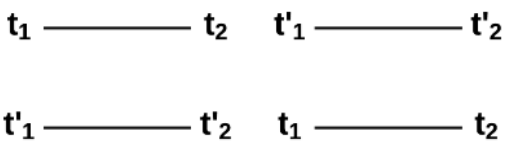

Figure 29: Policies Occurring Sequentially

Case 2.2 $t_1 < t'_1 < t_2 < t'_2$ or $t'_1 < t_1 < t'_2 < t_2$ (Figure 30): The policy application can be enforced as shown in Figure 31 if $t_1 < t'_1 < t_2 < t'_2$. There is a time period that starts at time t'_1 and ends at time t_2 where both P_1 and P_2 are applied on a given room. In order to deal with this scenario, the SS applies P_1 first using the same method described for "Case 1". At time t'_1, when P_2 needs to be applied, the SS removes P_1 from the room and removes the user with ID $suid$ from the set of $ActiveUsers$. It then applies a policy $P_3 = P_1 \cup P_2$ to the SR and adds the user with ID $suid$ back to the set of $ActiveUsers$. At time t_2, the SS needs to remove policy P_1 and keep policy P_2 in the SR. It starts by removing P_3 from the SR and removes user with ID $suid$ from the set of $ActiveUsers$. It then applies P_2 on the SR and adds the user with ID $suid$ back to $ActiveUsers$. At time t'_2, the SS removes policy P_2 from the SR and removes the user with ID $suid$ from $ActiveUsers$. The other situation $t'_1 < t_1 < t'_2 < t_2$ is dealt with in a similar manner.

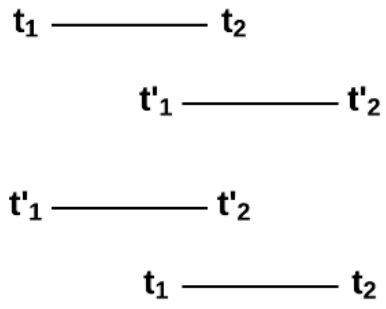

Figure 30: Policies Intersecting

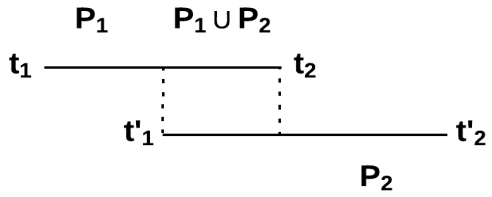

Figure 31: Intersecting Policies Example

Case 2.3 $t'_1 < t_1 < t_2 < t'_2$ or $t_1 < t'_1 < t'_2 < t_2$ (Figure 32): Policy application and adaptations are dealt with as in Case 2.2

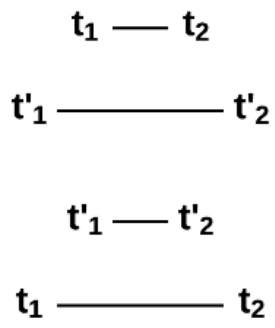

Figure 32: Policy Starting and Ending Within Runtime of Another Policy

In fact, through a generalized abstraction, event triggers from SS, and allowing "NULL" values in the data store implementation of our case study, it is possible to handle extensions with no modification to the inference and adaptation algorithms discussed in the previous sections.

7.6 Summary

In this chapter, we introduced a case study in which our context-aware system was responsible for the configuration control of a smart room. The case study involved many users with different preferences, and the context-aware system was responsible for determining the correct preferences to apply to a room. Furthermore, we described two extensions to the original case study in order to enrich it. In the first extension, we described an additional requirement in which smart rooms in the building could have different comfort attributes from one another. In the second extension, we discussed how the SS can apply global policies to a subset of the SRs in the SB. This extension explored a more complex example and allowed us to demonstrate how different types of policies are applied by the SS.

Chapter 8

Conclusion and Future Work

The smart city vision originated with the conviction that sensor technology and information and networking technologies can be efficiently integrated to provide efficient and uninterrupted services to urban areas which continue to attract many people from rural areas. In order to realize and support this vision, it is necessary to develop a rigorous platform where software engineering principles are applied to manage big data and integrate services from different application domains such as transportation, healthcare, and energy supply. This thesis addresses one fundamental issue in this context, namely the development of a generic context-aware architecture that can be used to build smart city applications. A summary of the thesis contributions together with possible future research are as follows.

8.1 Summary of Contributions

In this thesis, we presented a context-aware architecture for smart city applications. The focus of this work was to develop a generic architecture that can support and span various smart city applications, with a focus on privacy and security mechanisms at different levels. In our work, we considered context as a set of dimensions and attributes, and used it as a main component in the proposed architecture. This simple yet powerful method of modeling context has a well-defined representation that provides smart city applications with a rich way to model and represent data. Furthermore, an inference engine was used in our work to reason with context and infer new data. The proposed architecture can interface with any inference engine provided it knows what input to send and what output to receive. Finally, application and user policies were used to ensure protection of user information and roles with different levels of permissions were used to control the communication between the architecture components. These features helped us develop a generic and secure context-aware architecture for smart city applications.

In Chapter 1, we presented a list of contributions that we set to achieve in this thesis. The first contribution discussed in Chapter 1 describes the set of requirements to govern the design and implementation of a Wireless Sensor Network (WSN) in a smart city application. The rationale is that WSN is the foundation for building context-aware system architecture. In Chapter 2, a brief literature review of WSNs was presented. This embraced issues such as fault tolerance, privacy, and security requirements that must govern WSNs in smart city applications.

The second and third contributions presented in this thesis are to present a robust architecture for a context-aware system that incorporates privacy and security at every layer and to interface said architecture with an inference engine. In order to come up with a suitable context-aware architecture for smart city applications, we first started by reviewing other architectures from the literature in Chapter 3. By doing so, we were able to highlight the advantages and disadvantages of each solution and present an architecture that would address the shortcomings. While reviewing those architectures, we found that many of them did not address privacy and security concerns that often arise in context-aware systems. We also found that many did not discuss the data models used to represent information such as sensor data, context, and user preferences. In Chapter 4, we presented the context-aware architecture developed in this thesis. The various components were explained in detail, and the set of interactions between them was explained. A special focus was given for privacy and security, and solutions such as privacy policies, role-based access control, and encryption were discussed at each level of the architecture. In Chapter 5, a detailed software design was provided for each component of the context-aware architecture discussed in Chapter 4. Class diagrams and sequence diagrams were presented for each component in order to explain the inner workings of the components.

The final contribution was to implement a proof of concept and present a case study using the proposed architecture. Chapter 6 presented an implementation of the context-aware architecture discussed throughout the thesis. The proof of concept was developed on Amazon Web Services (AWS) using the Python programming language after it was determined that AWS and Python would meet the implementation requirements that we defined. The AWS services that were used in the implementation were discussed in details. Finally, we presented a "Smart Room Configuration Control" case study in Chapter 7. This case study involved having multiple users with different comfort level preferences use a smart room in a building. We were able to demonstrate how our proposed implementation intelligently determined which preferences to apply in the room based on some defined constraints. We further extended this case study to support different comfort attributes in different smart rooms in a building and to show how the "Supervisory System" can apply global policies on a subset of the smart rooms. These extensions allowed us to show how the proposed architecture and algorithms can be used to support a wide range of requirements without requiring any changes to the architecture, hence demonstrating the extensibility and the

generic aspect of the proposed architecture. This case study was inspired by the work presented in [MEeAT16], where the authors discuss an example in which they used their context-aware architecture for a smart living room. However, the authors failed to provide details on how they process and store user preferences and failed to demonstrate how their work can support more than one user in the room. Our case study addressed both of those issues and provided a complete solution that is easy to extend.

It is important to note that our proposed architecture is not restricted to smart buildings as it can be used for various smart city applications. An interesting example could be a smart traffic light system, where traffic is allowed based on various external sensors such as cameras and motion sensors. In this example, the external sensors would provide data to our context-aware system, where it would then be processed in order to generate adaptations that would indicate which traffic light should turn on. This simple example could be extended to include all the traffic lights on a road. This extension would require more complex rules combined with a sophisticated inference engine in order to generate adaptations that account for all the sensors in the environment, and since our architecture can interface with any inference engine, we could easily integrate the new inference engine with our system.

8.2 Future Work

In our view, context-aware architectures are important pillars of smart city development. Therefore, improving these architectures can vastly improve their performance, which would help in achieving the goals of smart cities. Some of the necessary improvements are the following:

- Evaluate the architecture for different smart city applications such as traffic management, air pollution, and energy consumption to further validate how generic the architecture is.
- Evaluate the architecture with a higher volume of data coming from a variety of sensors to study the scalability and fault tolerance aspects of the architecture. This would provide further research opportunities.
- Incorporate an intrusion detection system (IDS) in the architecture to provide an additional layer of security against some common attacks.
- Characterize the big data that arises from the Internet of Things (IoT) that forms the basis of smart city design and come up with efficient storage and retrieval methods in order to support different types of rigorous analysis and reasoning with such data to assess smartness.

Bibliography

[AANC+12] "Suha Alawadhi, Armando Aldama-Nalda, Hafedh Chourabi, J. Ramon Gil-Garcia, Sofia Leung, Sehl Mellouli, Taewoo Nam, Theresa A. Pardo, Hans J. Scholl, and Walker" Shawn. "building understanding of smart city initiatives". In "Hans J. Scholl, Marijn Janssen, Maria A. Wimmer, Carl Erik Moe, and Leif Skiftenes Flak", editors, *"Electronic Government"*, pages "40–53", "Berlin, Heidelberg", "2012". "Springer Berlin Heidelberg".

[AAS18] Ammar Alsaig, Vangalur Alagar, and Nematollaah Shiri. Formal context representation and calculus for context-aware computing. In Phan Cong Vinh and Vangalur Alagar, editors, *Context-Aware Systems and Applications, and Nature of Computation and Communication*, pages 3–13. Springer International Publishing, 2018.

[ABD15] Vito Albino, Umberto Berardi, and Rosa Maria Dangelico. Smart cities: Definitions, dimensions, performance, and initiatives. *Journal of Urban Technology*, 22(1):3–21, February 2015.

[ADB+99] Gregory D. Abowd, Anind K. Dey, Peter J. Brown, Nigel Davies, Mark Smith, and Pete Steggles. Towards a better understanding of context and context-awareness. In *Proceedings of the 1st International Symposium on Handheld and Ubiquitous Computing*, HUC '99, pages 304–307. Springer-Verlag, 1999.

[AHSC12] Muhammad R Ahmed, Xu Huang, Dharmandra Sharma, and Hongyan Cui. Wireless sensor network: Characteristics and architectures. *International Journal of Electrical, Computer, Energetic, Electronic and Communication Engineering*, 6(12):1398 – 1401, 2012.

[APW17] V. Alagar, K. Periyasamy, and K. Wan. Privacy and security for patient-centric elderly health care. In *2017 IEEE 19th International Conference on e-Health Networking, Applications and Services (Healthcom)*, pages 1–6, Dalian, China, October 2017.

[AvH04] Grigoris Antoniou and Frank van Harmelen. *Web Ontology Language: OWL*, pages 67–92. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[BDR07] Matthias Baldauf, Schahram Dustdar, and Florian Rosenberg. A survey on context-aware systems. *Int. J. Ad Hoc Ubiquitous Comput.*, 2(4):263–277, June 2007.

[BTR15] B. R. Tapas Bapu, K. Thanigaivelu, and A. Rajkumar. Fault tolerance in wireless sensor networks – a survey. *International Journal of Computer, Electrical, Automation, Control and Information Engineering*, 9(2), February 2015.

[CBN11a] Andrea Caragliu, Chiara Del Bo, and Peter Nijkamp. Smart cities in europe. *Journal of Urban Technology*, 18(2):65–82, 2011.

[CBN11b] Andrea Caragliu, Chiara Del Bo, and Peter Nijkamp. Smart cities in europe. *Journal of Urban Technology*, 18(2):65–82, 2011.

[Che04] Harry Lik Chen. *An Intelligent Broker Architecture for Pervasive Context-Aware Systems*. PhD dissertation, University of Maryland, 2004.

[Che10] Thomas M. Chen. Smart grids, smart cities need better networks [editor's note]. *IEEE Network*, 24(2):2–3, March 2010.

[CPFJ04] H. Chen, F. Perich, T. Finin, and A. Joshi. Soupa: standard ontology for ubiquitous and pervasive applications. In *The First Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services, 2004. MOBIQUITOUS 2004.*, pages 258–267, August 2004.

[Dey01] A. Dey. Understanding and using context. *Personal and Ubiquitous Computing*, 5:4–7, February 2001.

[FC04] Patrick Fahy and Siobhán Clarke. Cass-middleware for mobile context-aware applications. January 2004.

[GDB+13] "AmirHosein GhaffarianHoseini, Nur Dalilah Dahlan, Umberto Berardi, Ali GhaffarianHoseini, and Nastaran Makaremi". "the essence of future smart houses: From embedding ict to adapting to sustainability principles". *"Renewable and Sustainable Energy Reviews"*, "24":"593–607", "2013".

[GMW10a] O. Garcia-Morchon and K. Wehrle. Efficient and context-aware access control for pervasive medical sensor networks. In *2010 8th IEEE International Conference*

on Pervasive Computing and Communications Workshops (PERCOM Workshops), pages 322–327, March 2010.

[GMW10b] Oscar Garcia-Morchon and Klaus Wehrle. Modular context-aware access control for medical sensor networks. In *Proceedings of the 15th ACM Symposium on Access Control Models and Technologies*, pages 129–138, New York, NY, USA, 2010. ACM.

[HAAK+17] Muhammad Hamza Akhlaq, Mohammad Arif, Inam Khan, Nazia Azim, Shaheen Ahmad, Pakistan , Abdul Wali Khan, and University Mardan. Advantages, applications and research challenges in wireless sensor networks. 5:41–46, april 2017.

[HEH+10] C. Harrison, B. Eckman, R. Hamilton, P. Hartswick, J. Kalagnanam, J. Paraszczak, and P. Williams. Foundations for smarter cities. *IBM Journal of Research and Development*, 54(4):1–16, July 2010.

[Hna11] Sofian Alsalman Hnaide. A framework for developing context-aware systems. Master's thesis, Concordia University, April 2011.

[HNBR97] R. Hull, P. Neaves, and J. Bedford-Roberts. Towards situated computing. In *Digest of Papers. First International Symposium on Wearable Computers*, pages 146–153, October 1997.

[IOT] Aws iot core faqs. Online; accessed 14-April-2019.

[KFJ03] Lalana "Kagal, Tim Finin, and Anupam" Joshi. "a policy based approach to security for the semantic web". In *"The Semantic Web - ISWC 2003"*, pages "402–418", "Berlin, Heidelberg", "2003". "Springer Berlin Heidelberg".

[KK08] Cornel "Klein and Gerald" Kaefer. "from smart homes to smart cities: Opportunities and challenges from an industrial perspective". In Sergey "Balandin, Dmitri Moltchanov, and Yevgeni" Koucheryavy, editors, *"Next Generation Teletraffic and Wired/Wireless Advanced Networking"*, pages "260–260", "Berlin, Heidelberg", "2008". "Springer Berlin Heidelberg".

[KP88] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, August 1988.

[MCM14] Htoo Aung Maw, Hannan Xiaoand Bruce Christianson, and James A. Malcolm. A survey of access control models in wireless sensor networks. *Journal of Sensor and Actuator Networks*, 3:150 – 180, 2014.

[MEeAT16] Moeiz Miraoui, Sherif El-etriby, Abdulasit Zaid Abid, and Chakib Tadj. Agent-based context-aware architecture for a smart living room. *International Journal of Smart Home*, 10:39–54, 2016.

[MvH] Deborah L. McGuinness and Frank van Harmelen. Owl web ontology language overview. Online; accessed 20-July-2018.

[NM01] Natalya Fridman Noy and Deborah L. McGuinness. Ontology development 101: A guide to creating your first ontology. Technical Report KSL-01-05, Stanford Knowledge Systems Laboratory, 2001.

[NP11] Taewoo Nam and Theresa A. Pardo. Conceptualizing smart city with dimensions of technology, people, and institutions. In *Proceedings of the 12th Annual International Digital Government Research Conference: Digital Government Innovation in Challenging Times*, dg.o '11, pages 282–291, New York, NY, USA, 2011. ACM.

[OH13] Sharief Oteafy and Hossam Hassanein. Component-based wireless sensor networks: A dynamic paradigm for synergetic and resilient architectures. pages 735–738, October 2013.

[oPP] International Association of Privacy Professionals. What is privacy? Online; accessed 1-July-2018.

[PAW17] K. Periyasamy, V. Alagar, and K. Wan. Dependable design for elderly health care. In *2017 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 803–806, Sept 2017.

[PDY+17] P. Pappachan, M. Degeling, R. Yus, A. Das, S. Bhagavatula, W. Melicher, P. E. Naeini, S. Zhang, L. Bauer, A. Kobsa, S. Mehrotra, N. Sadeh, and N. Venkatasubramanian. Towards privacy-aware smart buildings: Capturing, communicating, and enforcing privacy policies and preferences. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 193–198, June 2017.

[PSC09] V. Potdar, A. Sharif, and E. Chang. Wireless sensor networks: A survey. In *2009 International Conference on Advanced Information Networking and Applications Workshops*, pages 636–641, May 2009.

[PSW04] Adrian Perrig, John Stankovic, and David Wagner. Security in wireless sensor networks. *Commun. ACM*, 47(6):53–57, 6 2004.

[RSC07] Kasim Rehman, Frank Stajano, and George Coulouris. An architecture for interactive context-aware applications. *IEEE Pervasive Computing*, 6(1):73–80, January 2007.

[SDA99] Daniel Salber, Anind K. Dey, and Gregory D. Abowd. The context toolkit: Aiding the development of context-enabled applications. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '99, pages 434–441. ACM, 1999.

[Tra16] Phillip Tracy. What is a smart building and how can it benefit you?, 2016. Online; accessed 2-July-2018.

[WA85] William W. Wadge and Edward A. Ashcroft. *LUCID, the Dataflow Programming Language*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[Wan06] Kaiyu Wan. *Lucx: Lucid Enriched with Context*. PhD thesis, Concordia University, Montreal, P.Q., Canada, Canada, 2006. AAINR16286.

[WSB+10] Doug Washburn, Usman Sindhu, Stephanie Balaouras, Rachel A Dines, Nick M Hayes, and Lauren E Nelson. *Helping CIOs Understand "Smart City" Initiatives: Defining the Smart City, Its Drivers, and the Role of the CIO*. 2010.