FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Context Aware Sensor Networks

Shashank Gaur

Doctoral Program in Electrical and Computer Engineering

March 28, 2022

Context Aware Sensor Networks

Shashank Gaur

Doctoral Program in Electrical and Computer Engineering

March 28, 2022

Resumo

Redes de sensores sem fio (Wireless Sensor Networks - WSNs) são uma tecnologia que se tornou popular nas duas últimas décadas em setores como agricultura, indústria de processo e de manufatura, hospitais, etc. As WSN tornaram-se uma ferramenta importante para recolher informações relacionadas com grandes infraestruturas, edifícios, utilizadores, vida selvagem, objetos em geral, etc. Com os avanços de hardware e software, as WSNs ganharam uma melhor capacidade de detetar, processar e comunicar, mesmo operando de forma autónoma com baterias. Este desenvolvimento veio permitir a execução de várias aplicações concorrentes nas WSN. Nestes casos, em vez de apenas recolherem uma única propriedade física num intervalo fixo de tempo e /ou local, as WSNs podem processar múltiplas informações recolhidas por vários dispositivos, conforme os requisitos do utilizador.

A pesquisa feita no contexto desta tese analisou sistemas implantados em cenários complexos, especialmente onde os requisitos do utilizador podem mudar com o tempo. Essas alterações podem ocorrer com as informações fornecidas pelos nodos que compõe as WSNs, após o processamento dos dados coletados. As alterações também podem ocorrer devido a intervenção do utilizador ou devido a mudanças no ambiente. Esses sistemas têm domínios de aplicação comuns, como edifícios inteligentes, fabricação e hospitais, onde os utilizadores interagem com vários objetos ou com o ambiente. A pesquisa e o estudo preliminar do estado da arte revelaram a necessidade de fornecer ferramentas confiáveis, que minimizem o esforço e automatizam a programação das WSN adaptativas. Este trabalho baseia-se em ferramentas existentes na comunidade e visa fornecer um melhor aproveitamento de WSNs.

A ênfase deste trabalho é nas WSNs heterogéneas uma vez que observámos um aumento no uso de sensores com recursos avançados junto com nodos de baixo custo e com baixo consumo de energia. A comunidade científica tem trabalhado para melhorar todos os aspetos desses sistemas com vista a melhorar o hardware, as comunicações e a cooperação entre os nodos. Com tudo isto, as WSNs tornaram-se parte integrante de sistemas avançados, como os Sistemas Ciber-Físicos (CPS) e a Internet das Coisas (IoT). Estas melhorias podem permitir que as WSNs detetem alterações e se adaptem adequadamente para fornecer um serviço mais eficiente.

A nossa pesquisa concentrou-se em fornecer melhor suporte para automação e adaptação de WSNs, e.g., para dar suporte a nodos sensores móveis que se podem localizar de forma eficiente, e depois alterar o aplicativo com base na localização atual. A construção destas WSNs exigiu trabalho no design do software do sistema, em particular para suportar aplicações que oferecem uma ampla gama de configurações para diferentes utilizadores e diferentes ambientes.

A nossa tese afirma que recorrendo aos conceitos de mobilidade, modularidade e abstração, juntamente com gestores de recursos e de contextos, é possivel construir um middleware com capacidade para adapação automática das aplicações a contextos de operação variáveis, recorrendo a uma programação intuitiva, adequada a utilizadores sem conhecimentos de progrmaação. Este tese foi validada com o middleware CAP - Context-Aware Programming, que constitui um passo importante para a construção de CASE - Context-Aware SEnsor networks. No todo, os trabalhos

para demonstrar a referida tese levaram à geração das seguintes contribuições para o estado da arte em WSN:

- *CAP - Context-Aware Programming*, um middleware que permite construir CASE - ContextAware SEnsor networks com programação com ferramentas de alto nivel de abstração, adequadas a utilizadores sem experiência de programação.
- *AdaptC*, um conjunto de politicas de adaptação elementares que os utilizadores podem utilizar para desenvolver applicações sensíveis a contextos.
- *mT-Res*, uma prova da exequibilidade da adaptação a contextos em WSN que consiste em adicionar os elementos arquiteturais necessários a uma solução existente de macroprogramação, nomeadamente T-Res.
- *NHS - Network-Harmonized Scheduling* aplicado a CAP. Trata-se de uma técnica de escalonamento desenvolvida em colaboração com outro trabalho, que gere as comunicações de múltiplas aplicações em execução numa WSN de fora eficiênca e pontual.

Abstract

Wireless Sensor Networks (WSNs) became a popular technology in the last couple of decades. Wireless sensor nodes are used in multiple industries from agriculture to process control and manufacturing industry, health systems, and many more. WSN are a major tool to collect information related to infrastructures, buildings, humans, wildlife, diverse objects, etc. With advances in hardware and software, WSN have gained a better ability to sense, process, and communicate, without negative impact in their battery-powered character. Thus, WSN started to move from application-specific to multi-application capable. Instead of just collecting a single physical property at a fixed interval of time and/or location, WSNs can collect and process multiple information and deliver it to multiple devices according to user requirements.

This research work has looked at systems deployed in complex scenarios, especially where user requirements can change over time. These changes can occur triggered by the information provided by WSN nodes after processing the collected data. Changes can also occur due to external factors from the user or environment. Such systems have popular application domains, such as process control, manufacturing or hospitals, where users interact with multiple objects. A preliminary research and study of the state-of-the-art revealed that there was a need to provide reliable and automated tools to program effortlessly such WSN systems. Our work builds on the existing tools in the community and provides a better generation of WSNs, namely CASE - Context-Aware SEnsor networks.

The emphasis of this work has been on heterogeneous WSNs. We have observed a rise in the use of sensors with advanced capabilities alongside low cost and energy-efficient sensor nodes. The research community has worked towards improving every aspect of such a system. Better hardware, efficient communication and cooperation among the nodes have been key efforts in recent years. With that, WSN have become an integral part of advanced systems such as Cyber-Physical Systems (CPS) and the Internet of Things (IoT). With such improvements, there is an opportunity to provide users with more application-centered systems. These improvements can allow WSNs to detect changes in the operational context and adapt accordingly to provide more efficient service.

This research focuses on providing better support for autonomous adaptation in WSNs. For example, support sensor nodes that can perform efficient localization and then change the application based on current user location. Building such WSNs required work towards the software design of the system, namely a middleware that supports applications that can target a wide range of solutions for different users and execution environments.

It is our thesis that supporting the design features of mobility, modularity and abstraction, together with managing resources and contexts, it is possible to build a middleware that allows automatic adaptation of WSN applications to varying execution contexts, which can be programmed intuitively by non-experts. This thesis was validated with the CAP middleware (Context-Aware Programming) representing an important step towards the construction of CASE - Context-Aware

SEnsor networks. Overall, the work towards validating the thesis led to the following list of contributions to the state-of-the-art in WSN:

- *CAP - Context-Aware Programming*, a middleware that allows building CASE - Context-Aware SEnsor networks with high-level programming tools suitable for non-experts.
- *AdaptC*, a set of basic adaptation policies that programmers can use to write context-aware applications.
- *mT-Res*, a proof of feasibility of context-awareness in WSN that consisted in adding features to an existing macroprogramming approach, namely T-Res.
- *NHS - Network-Harmonized Scheduling* applied to CAP, a scheduling technique, developed in collaboration within another work, that manages the communications of multiple applications in a WSN efficiently and timely.

Acknowledgement

Firstly, I want to start with thanking my supervisors, Prof. Eduardo Tovar and Prof. Luis Almeida for their unconditional support during my PhD. Eduardo's support in the beginning of thesis work was essential for my professional growth. Eduardo provided excellent opportunities and avenues to expose me to the WSN research community and never hesitated to help in anyway possible. Luis's supervision was essential for me to learn many useful skills while writing publications and even this thesis document. Without Luis's help, I would not be able to find motivation and skills needed to put together this document and his confidence in me as a student has been important for me. I am thankful to ISEP for the financial support provided over the years to conduct my PhD at CISTER, one of the most prestigious research institute in Portugal.

I also want to thank Nuno Pereira and Luca Mottola for their contributions during my PhD. They were very helpful on evaluating my ideas and criticizing the thought process. It helped me greatly to develop a critical sense of ideation and taught me the essential process for thinking like a researcher. Both of them always asked the hard questions, which kept me grounded, helped realize my potential and evaluate my skills time to time. I want to thank Luca for hosting me for few days at NESLAB at Politecnico di Milano, and allowing me to see how excellent work is done.

I could not have finished this work without the help and support provided by João Loureiro, Vikram Gupta, Geoffrey Nelissen and Vincent Nélis. They were always available for any discussion needed either scientific or philosophical. João helped me a lot in discovering many engineering skills that I lacked and without his hints about various solutions I would not be able to do any experiments. Vikram helped me a lot to understand basics of implementations of proof of concepts and also provided me a useful opportunity to contribute to his doctoral thesis. It was an amazing experience to be able to work with these exceptional researchers and generous human beings. Their impact will last as long as I will live. Several other researchers at CISTER helped whenever they could to keep me motivated and push me towards finishing the work. I am obliged for the positive environment created by my peers.

I want to convey special thanks to Inês and Sandra who contributed to all the administrative and everyday hurdles one faces when living in a foreign country. Their welcoming nature and tolerance to my innocent queries made life much easier.

In the end, a lot of other people have helped me reach this stage. Thank you to everyone who has helped me stay on the path. Hope I have been worth it.

Gratitude turns what we have into enough

Shashank Gaur

"I believe in intuitions and inspirations.
I sometimes feel that I am right. I do not know that I am."

Albert Einstein

Contents

List of Figures

List of Tables

Glossary

AoA	Angle-of-Arrival
CASE	Context-Aware Sensor Network
CPPS	Cyber-Physical Production System
CPS	Cyber-Physical System
DSP	Digital Signal Processor
FPGA	Field-Programmable Gate Array
GNSS	Global Navigation Satellite System
GPS	Global Positioning System
HVAC	Heating, Ventilation and Air-Conditioning
ICPS	Industrial Cyber-Physical System
IMU	Inertial Measurement Unit
IoT	Internet-of-Things
ISM	Industrial, Scientific and Medical
LoWPAN	Low-Power Wireless Personal Area Networks
6LoWPAN	LoWPAN based on IPV6
MANET	Mobile and Ad-hoc Network
MDS	Multi-Dimensional Scaling
MES	Manufacturing Execution System
NHS	Network Harmonized Scheduling
OSGi	Open Services Gateway initiative
QoS	Quality of Service
RF	Radio Frequency
RSSI	Received Signal Strength Indicator
TDMA	Time-Division Multiple Access
TDoA	Time-Difference-of-Arrival
ToA	Time-of-Arrival
VM	Virtual Machine
VSN	Virtual Sensor Network
WiFi	IEEE 802.11 Wireless Local Area Network
WSAN	Wireless Sensor and Actuator Network
WSN	Wireless Sensor Network

Chapter 1

Introduction

Sensors and Actuators have been around for a long time. It is assumed that the first thermostat was invented in the early 1600s. Measuring physical quantities such as Temperature, Pressure, Light Intensity, etc, has always proved useful for various purposes. With continuous research and technological advancements there has been an increase in the number of sensors around, as well as in their capabilities. In particular, the last decades of the 20^{th} century saw the advent of the so-called smart sensors that added a range of functionality to the core sensing function, from digital conversion to filtering, encoding and communication. A room in a smart home, nowadays, easily counts with 10 or more sensors to track light, temperature, pressure, sound, humidity, etc. Reading from all these sensors motivated their integration in a network, making use of their digital communication capabilities. Networked sensors simplify the execution and coordination of applications. Moreover, sensing devices can be aware of each other and exploit each other's data to perform more efficiently.

1.1 Wireless Sensor Networks

Building sensor networks requires considering which communication medium to use. Wired networks have been used since the 1980s to connect both sensors and actuators, e.g., the fieldbuses developed for industrial applications. However, despite the enhancements achieved by fieldbuses when comparing to the previous solution based on point-to-point connections from each sensor/actuator to the associated controller, wired networks still present intrinsic undesirable limitations arising from the use of wires, namely their rigid physical layout and their cost.

These limitations can be circumvented resorting to wireless communications, however, these became a viable option recently, only. Further research and technological advancements were needed to develop wireless low cost interfaces with high bandwidth, sufficient reliability and adequate security, particularly using Radio-Frequency (RF).

Nowadays, connecting sensors in a wireless network is the most efficient way to broaden the range of *environment-aware* applications. With an increase in mobility-based innovations it can be realized that information flow is growing quite rapidly. Hence, connecting sensor nodes with

wireless technologies has been promoted in all possible aspects. Technologies such as WiFi, Zigbee, Bluetooth, etc. are used to connect nodes in various network topologies suitable to different applications. In a generic scenario, all nodes are connected via a coordinator or gateway node, which relays the required information to a user/controller as shown in Figure 1.1.

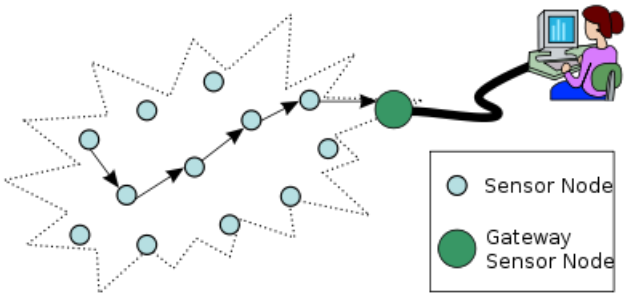

Figure 1.1: Wireless Sensor Network

Wireless Sensor Network (WSN) nodes, as shown in Figure 1.2, can be typically analyzed according to hardware and software aspects, as we explain the following sections.

Figure 1.2: A typical sensor node hardware

1.1.1 Hardware

The hardware of a sensor node is typically comprised of four main blocks, namely Radio, Microcontroller, Sensor and Power (Figure 1.3).

Radio The radio of the node combines both transmitter and receiver, thus being a transceiver. Most frequently, communication is RF-based. The free band allocated to Industrial, Scientific and Medical (ISM) equipment is used more often, hence the common frequencies 868 MHz, 903 MHz and 2.4 GHz. RF communication is kept as simple as possible in WSNs. Hence, the digital signal processing needed for modulation and filtering, as well as medium access techniques are kept the simplest that the needed performance requires. There are four states of a radio, which are transmit, receive, idle, and sleep. Both transmit and receive modes, the so-called active modes, impose comparatively large energy consumption. The

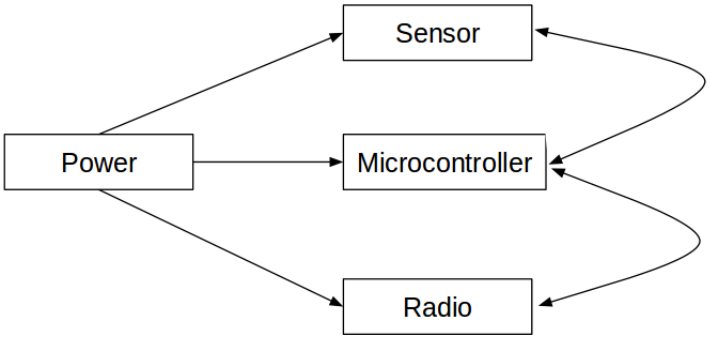

Figure 1.3: Main blocks of a sensor node hardware

sleep mode requires the least energy but switching from sleep to the active modes consumes a significant amount of energy. The idle mode consumes more energy than sleep mode but the switching to the active modes is faster and less costly, energy-wise. Hence, if an application may require transmission or reception very often, then its radio is kept in idle mode. If the application presents relatively long periods without communication then the radio can be kept in sleep mode.

Microcontroller Every sensor node has a brain of its own. Usually this is a microcontroller that performs all the processing tasks, from collecting data to analyzing it and then taking subsequent action defined by the user via the software. The microcontroller is connected to the sensor circuit in some way so that it can collect data accordingly. While a microcontroller is the most common choice, there can be other options such as Digital Signal Processors (DSP) and Field-Programmable Gate Arrays (FPGA), but the microcontroller low cost, low energy consumption and easiness of programming make it a hard to beat option.

Sensor This is the most specific part of the node, including the sensing transducer and the electronic circuits for signal conditioning and interfacing to the microcontroller that allow it to read the corresponding physical entity such as temperature, pressure or light intensity. Most frequently, the signal coming from the sensor transducer is analog and it is converted to digital by the microcontroller. Alternatively, some sensors already produce data in a form that can be read using digital inputs of the microncontroller, e.g., pulses of variable width or even using digital inter-component communications such as I2C or 1-Wire.

Power Source Most WSNs require that the sensor nodes are small in size, cheap in cost and consume the least energy possible. However, these requirements may differ from application to application. In some cases, it may be possible that WSN nodes are physically deployed in a place with access to an unbounded energy source. In other more common cases, nodes are either mobile or deployed in an isolated area needing a small and light energy source or some form of energy scavenging.

1.1.2 Software

The software behind a wireless sensor network determines the communication protocols above the radio level, the algorithms for processing the data collected by the sensor nodes, the configurations of the nodes and even the programming language used. The options available in these aspects are typically constrained by the operating system in place. Figure 1.4 shows two of the major operating systems used in WSN, namely TinyOS and Contiki. Both use C as the programming language, but with different versions.

Figure 1.4: The two most popular Operating Systems for WSN

1.2 Application Domains

WSN can be deployed and operated in many different configurations according to different application domains. We can roughly identify three main domains:

Industrial WSN are widely applied in industry, frequently called Wireless Sensor and Actuator Networks (WSAN) because of the presence of actuator nodes and the corresponding inverse flows of information when compared to the typical flows from sensors. Note that sensors are information sources whereas actuators are sinks. Examples range from tracking of goods in transportation or warehouses, tracking of materials in production lines or continuous process plants, to monitoring of machines and devices status. All benefit from the support of sensor nodes. In the particular case of the transportation industry, various sensors can be used to acquire information about not only the goods but also the status of parts of each vehicle or even of the surroundings of the vehicle. In construction, sensor nodes can be used to monitor the structural health of buildings, bridges, dams, etc. In agriculture, sensors distributed in the field may allow the construction of maps of humidity, pH and other environmental variables to control irrigation, fertilization and even pests (Ivanov et al., 2015).

Personal Some of the research boosts observed in WSN addressed building applications for personal use. The most popular and common application is Smart Home(s) or Office(s). WSN are becoming the very next technological evolution in these domains (Samuel, 2016), not only providing ease of access to the users but also contributing toward features such as energy conservation, safety & security and health monitoring.

Public The applications in this domain are the ones that benefit a larger number of people instead of just a few persons or one organization. These applications include traffic monitoring in cities, natural disaster detection, water quality monitoring, patient monitoring in hospitals or even health monitoring of specific population groups (Hu et al., 2017), etc. They provide early detection of issues enabling prompt mitigation, not only automatically but also by engaging public authorities.

These domains can be further divided into different sectors. For example, the Industrial domain includes the aviation sector, the industrial automation sector, etc. The Personal domain includes sectors such as smart-homes or smart-offices. The Public domain includes sectors such as the forest, smart cities, buildings, civil engineering structures, etc. In all we can find WSN being used in an increasing manner.

1.2.1 Cyber-Physical System

We referred before that, particularly in the industrial domain, WSN frequently include actuators, thus forming WSAN. Hence, a full control loop can be defined encompassing a WSAN and adequate controllers.

In the past, the design of such distributed control systems used to be done separately considering the physical laws of the control plant and the discrete laws of the communication and computation system. However, it was realized that each of these two parts influenced each other, thus an efficient design required a unified approach. By the end of the first decade of the 21st century, this unified approach became generally known as Cyber-Physical Systems (CPS).

Hence, CPS bridge the gap between the cyber-world of computation and communication with the physics that govern the evolution of the system state. These systems are a combination of physical sensors and actuators and the associated software, including real-time operating systems, protocol stacks, and control algorithms, that can monitor, control, and execute certain tasks on behalf of the user. Figure 1.5 shows the typical composition of a CPS highlighting the presence of the WSAN.

When applied to the industrial domain, it is common to find more focused expressions to refer to a CPS, such as Cyber-Physical Production Systems (CPPS) or Industrial Cyber-Physical Systems (ICPS). A few examples of CPS closer to the target of this thesis are:

- Robotic production cells
- Manufacturing Execution Systems (MES)
- Smart homes and offices
- Smart buildings

The evolution of CPS technologies enables new opportunities and poses new research challenges. CPS can be composed of multiple groups of devices connected through a combination of

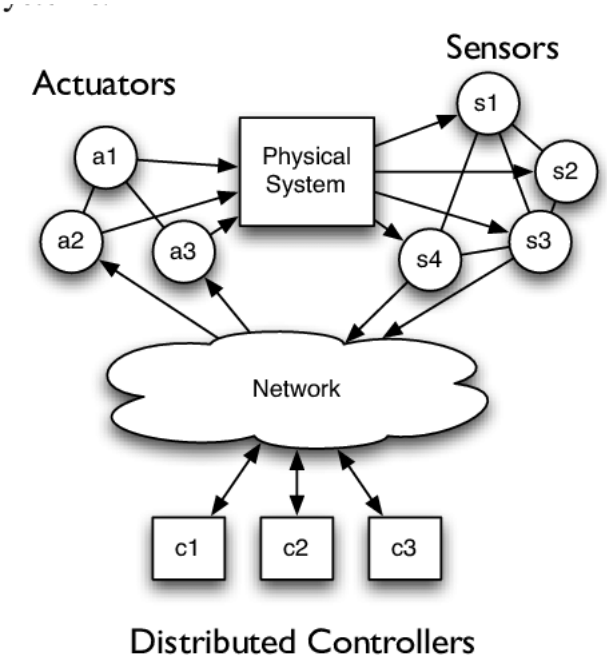

Figure 1.5: Generic components that make up a CPS

both wired and wireless networks, and driven by new applications and demands coming out of various application domains. One such case that is particularly relevant to this thesis is that of evolving systems, which reconfigure themselves during the system operation according to specific events. Reconfigurations include the addition of new nodes, new services or new applications, or their removal or adaptation.

In general, CPS bring the promise of a strong impact on modern societies, either in economic terms but also in environmental aspects and also comfort for users. CPS in smart houses can reduce energy spent for lighting, heating and cooling. CPS can help users reach *Net Zero Energy* status managing user requirements together with generation of energy from local renewable sources, such as solar and wind. CPS in smart houses can also monitor patients or elderly users and provide assistance when needed. These applications demonstrate the benefits of CPS, which can lead to wider adoption.

With such expansion, it is clear that CPS will interact with many non-technical users who wish to utilize the system capabilities efficiently or whose input is critical for some applications. This raises the demand for easy to use tools to write applications for such systems, but also to deploy them and control their execution. To do so, in this thesis we advocate using model-based development for CPS. As referred in (Rajkumar et al., 2010), we also believe that a framework providing high-level abstractions that can cover the entire CPS design space will also provide a better interface for users.

1.2.2 Internet of Things

Another concept highly related to WSN and CPS is that of the Internet of Things (IoT) (Atzori et al., 2010). Despite being more than two decades old, it became a widely disseminated research topic soon after the CPS concept became widely accepted.

The IoT is realized embedding each and every physical object with intelligence, i.e., a microcontroller, and communication capabilities. IoT devices are frequently described as WSN nodes, with limited RF-communication, computing and energy resources, with the capability of collecting data. However, the IoT assumes a denser network of more heterogeneous devices. Ultimately, the IoT vision of *anytime, anywhere, anything* implies ambitious new challenges in communications and applications execution and control.

IoT devices are currently being developed by all consumer industries, with a strong push towards adoption. Soon, the IoT will be fully integrated with the Internet at large, including all its devices. This will be particularly noticeable in applications such as smart homes and offices, location-based services, smart transportation, etc.

A key part IoT relies on is the density of devices. It is expected to reach 50 billion IoT devices around the globe in a few years. This will lead to much more data collection than ever, which provides an opportunity to new applications, services and processes. Hence, providing a better interface between the IoT user and all these densely networked IoT devices is mandatory for the success of the IoT but also a significant research challenge.

Figure 1.6 shows an example of an IoT system in the smart homes sector. This system connects devices with very different capabilities using different networking protocols to provide users with greater access to complete usual tasks in a house. This coverage of devices and protocols enables the IoT to become pervasive and able to incorporate an increasing number of functionalities, particularly based on environmental conditions, including location, activity, availability of resources or time.

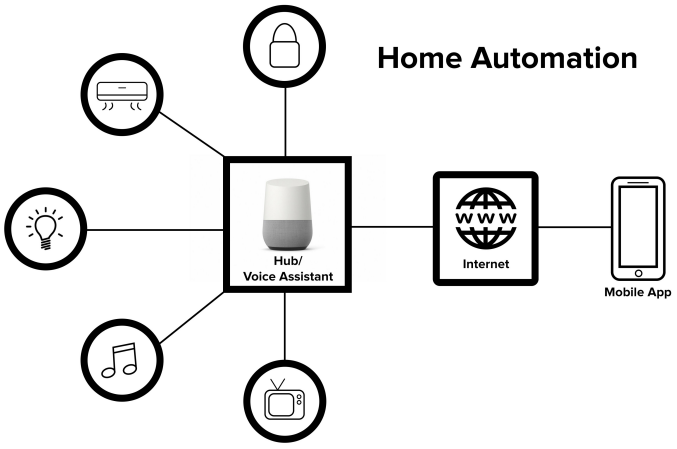

Figure 1.6: Generic example of the IoT in smart homes

Many of the IoT functionalities can be implemented with in-network processing of simple rules (e.g. detection of presence, changes in temperature and light) at local nodes. However, heavier processing may require more powerful nodes. This can be provided locally following an *edge computing* approach, e.g., the smart hub shown in figure 1.6, or remotely in the *cloud*. Ultimately, functionalities should be able to migrate without user intervention, and execute seamlessly in the sensor nodes, in the edge nodes or in the cloud according to availability of resources and user-defined performance requirements. The user can not be expected to anticipate these cases and hardcode them in the applications.

The user should write applications based on functionalities while being essentially agnostic to the details about the available devices in the network. Significant research efforts are being being carried out recently to provide programming solutions that enable that feature.

1.3 Context-Aware Sensor Networks

WSN are intrinsically embedded in the environment in which they operate. Thus, they are particularly subject to changes in that environment and in their own operating conditions. These changes may require the suspension of some applications and the launching of other ones, or the adaptation of running applications. This capacity to control applications based on environmental or operating conditions is commonly referred as context-awareness.

Context-aware software adapts according to the location of use, the specific user(s), the number of nearby people, the hosting device, other accessible devices in the network, energy level, noise level, network connectivity, communication costs, communication bandwidth, etc. All these aspects form the notion of *context*.

1.3.1 Context-Aware Computing

The roots of Context-Aware Computing go back to the early 1990s with the development of PARCTAB (Schilit et al., 1994). This was a small handheld device, a tablet, that used an infrared-based cellular network for communication. The tablet acts as a graphics terminal and most applications run on remote hosts. Based on the location of the tablet in an office, meeting room, or a lab, it can display different applications.

Examples of context-awareness can be found in diverse application sectors from healthcare, to manufacturing, smart homes, etc. In healthcare, a body sensor network can detect changes in the context of a patient and adapt different applications accordingly. For example, generally patient's health data is collected from different sensing devices to smart devices that are in range, which pre-process the data and send it to medical staff. On-body sensing devices would lack the processing power and energy to carry out such data pre-processing and transmission. If an emergency situation is detected the smart device can generate an alarm, instruct the sensing devices to adapt the sensing, e.g., increasing rate, and automatically set up a phone call with the appropriate medical staff, which can lead to life-saving scenarios.

1.3.2 Context-Awareness in WSN

The current capabilities of modern WSN nodes already allow using context to drive actions in the WSN. Not only nodes can sense and collect data from the operational environment, but they can also generate a context from it and adapt the running applications accordingly.

For WSN we can define three possible aspects of context: user, system, and environment. The context for users would be based on location, identity, schedule, etc. For system purposes, a context can be derived from the energy level of the node, neighboring nodes, communication cost, etc. From the environment, the context would refer to the physical location of the node, actual sensed data, etc.

A benefit for the user arising from the context-awareness of WSN is that the network can typically detect changes in context faster than a user can perceive them. In this way, WSN can anticipate user needs. This is achieved with two actions: first the WSN must be aware of the current context and the respective applications; second the WSN must detect the activation of a new context, typically by means of associated sensing variables, and enforce the corresponding changes in the set of running applications, possibly deploying new ones, triggering idle ones, or even migrating or removing running ones. If a WSN can carry out both of these actions, we name it Context-Aware Sensor Network (CASE).

An example of a CASE is shown in Figure 1.7 concerning a smart office. The CASE collects temperature and location data as well as inputs from fire detectors. A *normal habitat* context, when fire detectors are idle, a temperature control application is executed, maintaining a desired value in different parts of the building where users are present, actuating heating/cooling equipment.

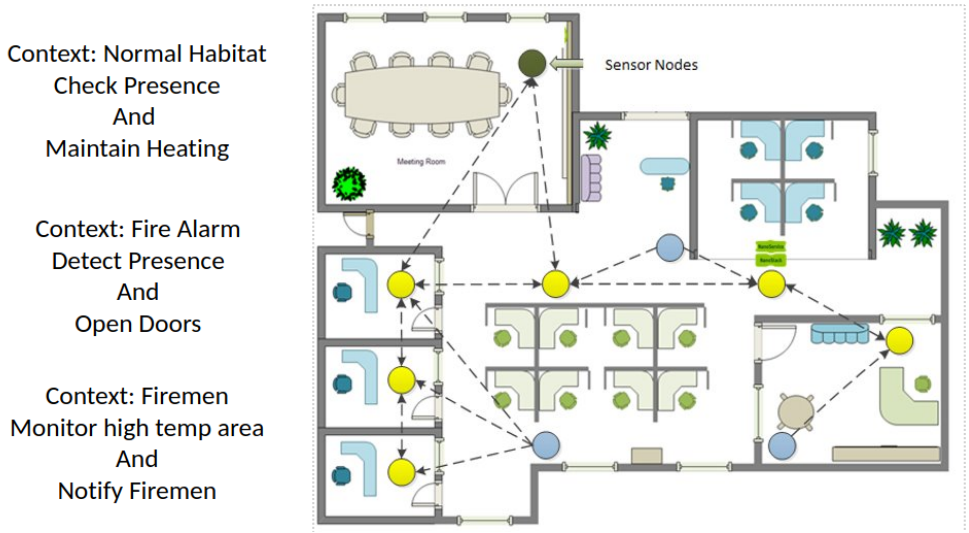

Figure 1.7: Example of Context-Aware Sensor Network

A *fire alarm* context is triggered whenever a fire detector is activated in the building, launching an evacuation application that leads users to exit the building through an adequate path. A *firemen* context is activated when firemen arrive onsite, launching a firemen support application that informs them of the presence of users in high-temperature zones.

A context can also be defined with simple types corresponding to the types of the data collected by the WSN, as shown in Figure 1.8. These are situations in which the data directly represents contextual information. Context changes can be triggered by applying simple rules on the collected data.

Existing Examples	Context Types	Human Concern
Auto Lights On / Off	Room Activity	Convenience
File Systems	Personal Identity & Time	Finding Info
Calendar Reminders	Time	Memory
Smoke Alarm	Room Activity	Safety
Barcode Scanners	Object Identity	Efficiency

Figure 1.8: Contexts defined by corresponding data types

More powerful devices can process the acquired data and generate more complex contextual information, for example, based on structures, leading to more elaborate contexts, as shown in Figure 1.9.

Potential Examples	Context Types	Human Concern
Auto Cell Phone Off In Meetings	Identity	Convenience
Tag Photos	Time	Finding Info
Proximal Reminders	Location Proximity	Memory
Health Alert	Activity History	Safety
Service Fleet Dispatching	...	Efficiency

Figure 1.9: Contexts based on complex data structures

Finally, note that in both cases each context relates to a particular human concern. This can help the CASE to consider the user needs. It is also possible to attach priorities to contexts and corresponding applications based on the human concern it satisfies. For example, in certain scenarios safety will take precedence over convenience.

1.4 Programming Wireless Sensor Networks

The implementation of context-awareness in WSN requires a robust and efficient programming platform not only to develop and deploy the CASE operating structures but also the respective applications.

Programming WSN has been a key research area for more than a decade. Despite the significant advances that led to the hardware available today, an efficient implementation of WSN can only be realized if proper tools are available for the user to write applications.

Currently, most WSN deployment programming is done very close to operating systems such as Contiki-OS or TinyOS, relying extensively on OS-specific calls. The code displayed in Listing 1.1 showcases an application to make an LED blink in Contiki[1]. Even a simple piece of code as in this example requires the use of several OS-related functions, such as *process* and *timer*. Despite the hardware abstraction granted by the OS, this kind of OS-based programming still requires knowledge of low-level details that is at the reach of expert programmers, only.

Listing 1.1: Code to blink an LED in Contiki

```
#include "contiki.h"
#include "led.h"
process_start(blink_led)
{
        leds_off();
        timer_set(10s);
        Wait(timer);
        leds_on();
}
```

Currently, low-level programming is still the dominant WSN programming paradigm. For example, one of the most widely used WSN platforms in Europe, the IoT-LAB[2], still relies on low-level programming. The main problems with low-level programming are that it requires significant technical background and the level of details involved may drive the focus of users that design applications away from the application logic. If the user is not well versed in the low-level details, the application can be error-prone and the development and deployment process may become more time-consuming than desired. This can lead to frustrating user experience.

Facilitating applications design is paramount to support the wide adoption of WSN, CPS, IoT and CASE because many applications will be written by non-technical users. With the aim of providing high-level programming platforms, several programming models and approaches were proposed in the past decade, each with different characteristics. For example, *Macroprogramming*, one of such approaches, provides the user with both network and node level abstractions. However, after listing and comparing the most popular proposals for high-level WSN programming support we concluded that no solution exists, yet, that provides context-awareness. Hence, in this thesis we will show how to build on top of a convenient existing programming platform to provide support for context-awareness and build CASE.

[1] This is a simplified version of the actual code for demonstration purposes, only.
[2] https://www.iot-lab.info

1.5 Thesis Statement

As we discussed along the previous sections, the original WSN concept evolved and became intertwined with on-going relevant trends, particularly WSAN, CPS and IoT. As a consequence of this evolution, WSN became a highly distributed computing platform that can run applications with multiple purposes, with the specificity of being deeply connected to its operational environment. This specific WSN feature allows detecting changes in such environment and adapt applications automatically to suit user needs. This adaptation is named context-awareness and we referred to the WSN that support it as context-aware sensor networks or CASE. However, developing CASE for wide adoption requires a suitable middleware that hides as much low-level platform details as possible to allow non-expert users to define and manage contexts and applications. Such middleware is still an open challenge to realize the full potential of CASE while minimizing any impacts on system performance in terms of storage and energy. Hence, we state our thesis as follows:

By PROVIDING
i) programming models with mobility, modularity, and abstraction features,
ii) a resource manager that tracks computing resources available and their capabilities, and
iii) a context manager that tracks collected data to activate/deactivate contexts,
THEN
a WSN middleware can automatically adapt the execution of applications to changes in contexts while enabling high-level applications programming with efficient operation. This middleware is a cornerstone for the wide adoption of context-aware sensor networks (CASE).

Figure 1.10 shows a draft of the architectural plan for the envisaged middleware that is implied in the thesis. The figure highlights the fundamental middleware components (represented in grey), namely the context manager, the resource manager and the high-level programming module to write applications.

1.6 Methodology

To realize the proposed middleware, we defined and followed the methodology that we describe next, consisting of a set of tasks that addressed different aspects of the development and validation process.

- **Requirements** A deep understanding of programming models is required to determine the essential features of the proposed middleware. We do this with a detailed literature survey of all available macroprogramming models for WSN. We also survey programming support available for context-aware systems in general. Special attention is given to service-oriented

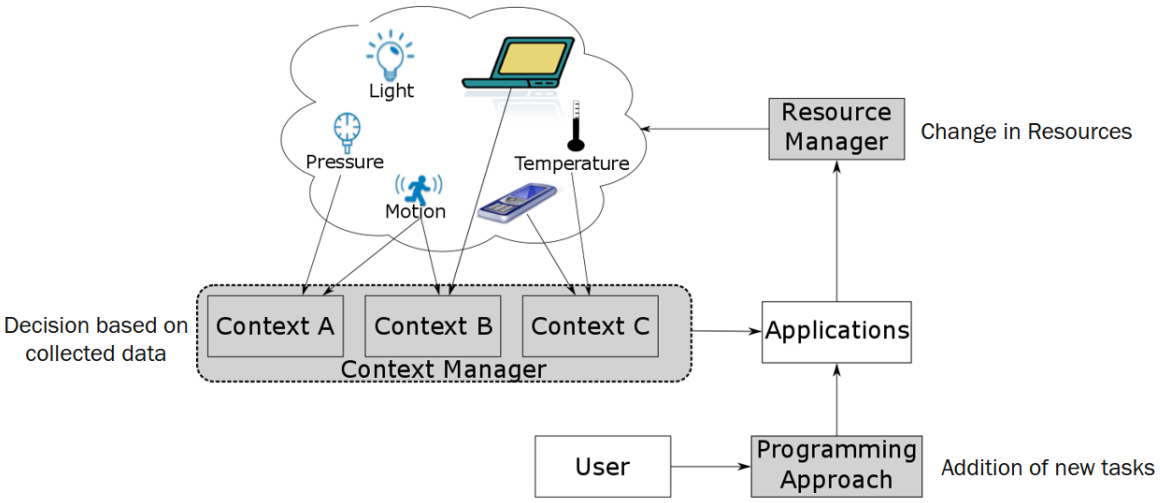

Figure 1.10: Proposed Middleware for Context-Aware Sensor Network

architectures to understand the abstraction of low-level details from the application logic. Finally, we also survey all the frameworks for multi-application support in WSN.

- **Extending the State-of-the-Art** Using the literature survey carried out in the previous task, we identify the related state-of-the-art and from it we select a work suitable for extension towards the proposed middleware. Building on the state-of-the-art facilitates scientific progress while providing support for existing applications. It also allows incorporating existing features into the proposed middleware.

- **Feature Design** Having identified an existing work to be extended, we design the required additional features. We evaluate these using metrics available in the literature and select the most important features. This helps us finalizing an architectural plan of the middleware to support the implementation.

- **Implementation** With the design features and the architecture, we implement the middleware with a user-friendly interface. We write the support documentation for the implementation and also provide an API to be used by external tools. Also, we implement a web-based interface for users to provide input and monitor the middleware.

- **Experiments** Towards the end, we conduct various experiments to validate the implementation. Particularly, we run the applications written using the middleware in both simulation and real deployments, and particularly using a large scale test-bed available for experimentation.

1.7 Contributions and Publications

The work developed in this thesis generated two main contributions to the state-of-the-art in programming and operating WSN (bullets 1 and 2 in the list below). In addition, we also present two

other contributions of our work (bullets 3 and 4), though of smaller conceptual strength. These contributions are listed next, together with the publications in which they were shared with the community.

1. The target **middleware** of the thesis, which allows building CASE - context-aware sensor networks, with high-level programmed applications that can be written by non-experts. The middleware relies on multiple tools and technologies, particularly CoAP instructions in Python scripts. It was presented and explained in the following publication:

    - Shashank Gaur, Luis Almeida, Eduardo Tovar and Radha Reddy. *CAP: Context-Aware Programming for Cyber-Physical Systems*. ETFA 2019 - 24th Conference on Emerging Technologies and Factory Automation, Zaragoza, Spain. 10-13 September of 2019.

2. A set of basic **adaptation policies** that programmers can use to write context-aware applications. These policies are part of the context manager and allow activating/deactivating contexts previously defined by the users and are then applied by the middleware on the set of running applications. This contribution was presented and explained in the following publication:

    - Shashank Gaur, Luís Almeida, Eduardo Tovar. *AdaptC: Programming Adaptation Policies for WSN Applications*. SAC 2019 - 34th ACM/SIGAPP Symposium On Applied Computing. Limassol, Cyprus. 8-12 April of 2019,

3. A **proof of usability** of context-awareness, that consisted in adding features to an existing macroprogramming approach, namely T-Res. This work validated the possibility of providing context-awareness by extending an existing WSN programming approach. This contribution was described in the following publication:

    - Shashank Gaur, Raghuraman Rangarajan, Eduardo Tovar. *Extending T-Res with mobility for context-aware IoT*. IoTDI 2016 - 1st IEEE International Conference on Internet-of-Things Design and Implementation. Berlin, Germany. 4-8 April of 2016.

4. A **scheduling technique**, developed in collaboration within another work, to manage the execution of multiple applications in a WSN by scheduling the associated traffic in a way that reduces communication latency. Our contribution is essentially the discussion on the application of this technique to support the middleware proposed in 1 above. The scheduling technique was presented in the following publication:

    - Vikram Gupta, Nuno Pereira, Shashank Gaur, Eduardo Tovar, Raj Rajkumar. *Network-Harmonized Scheduling for Multi-Application Sensor Networks*. RTCSA 2014 - 20th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications. Chongqing, China. 20-22 August of 2014.

The work in this thesis was also shared with the community in the following communications in poster sessions and PhD Fora:

- – Shashank Gaur, Luis Almeida, Eduardo Tovar. *PhD Forum: Automatic Allocation of Tasks in T-Res for WSN*. DCE 2019 - 3rd Doctoral Congress in Engineering. Porto, Portugal. 27-28 June of 2019.
  – Shashank Gaur, Raghuraman Rangarajan, Eduardo Tovar. *POSTER: Bringing Context-awareness to wireless sensor networks*. CPS Student Forum Portugal, within the CPS Week 2018. Porto, Portugal. 10-13 April of 2018.
  – Shashank Gaur, Raghuraman Rangarajan, Eduardo Tovar. *Demonstration Abstract: Automated Resource Allocation for T-Res*. IPSN 2016 - 15th ACM/IEEE International Conference on Information Processing in Sensor Networks. Vienna, Austria. 11-14 April of 2016.
  – Shashank Gaur. *PhD Forum: Bringing context awareness to IoT-based wireless sensor networks*. PerCom Workshops 2015 - IEEE International Conference on Pervasive Computing and Communication Workshops. St. Louis, USA. 23-27 March of 2015.
  – Shashank Gaur, Nuno Pereira, Vikram Gupta, Eduardo Tovar. *POSTER: A Modular Programming Approach for IoT-Based Wireless Sensor Networks*. EWSN 2015 - 12th European Wireless Sensor Networks Conference . Porto, Portugal. 9-11 February of 2015.

1.8 Thesis Structure

The remainder of this dissertation is structured as follows: Chapter 2 discusses different programming approaches available for WSN, followed by the popular programming platforms for context-aware systems. Chapter 3 discusses the work done on extending the existing state of the art towards the proposed middleware. Chapter 3 also provides a working demonstration of some of the new additions to existing work. We propose a new middleware in Chapter 4 with design features and its architecture. Chapter 4 also includes details on the implementation of this middleware. Chapter 5 describes some new adaptation policies for writing applications in the proposed middleware. Chapter 6 presents a scheduling technique to support multiple application running in a WSN in a timely fashion and it also discusses how this can be used to support the proposed middleware. Finally, in Chapter 7, we conclude this thesis by providing a summary of the present research contributions and the future work.

Chapter 2

From WSN to Context-Aware SEnsor networks

Wireless sensor networks are currently being deployed in many application domains to satisfy various user needs, frequently within the trendy frameworks of CPS and IoT. This has been enabled by significant research work done in the past two to three decades in several fundamental topics, such as **localization**, **networking**, **WSN programming**, considering energy management and resource allocation, and finally dynamic adaptation and **context-awareness**.

These developments fostered new deployments that, in turn, motivated new research and new developments, in a positive cycle. In this chapter we discuss these fundamental topics, giving an overview of WSN development towards context-awareness and current status.

2.1 Localization

Localization is one of the most relavant WSN challenges. Many applications such as smart home, targeted advertisement, geo-social networking, patient monitoring and search and rescue operations, need an accurate localization method to perform efficiently. While the cellular infrastructure of the telecommunication operators and particularly Global Navigation Satellite Systems (GNSS) like the Global Positioning System (GPS) can provide support for relatively accurate localization outdoors, it still remains an issue indoors. Being able to localize mobile nodes in offices, homes, buildings, warehouses, etc, can strongly impact the way we interact with the surroundings. It can fundamentally change how applications will adapt their behavior according to location, potentially enabling a myriad of new services for the user. This adaptation is our main concern in this thesis, not the specific localization techniques used. Nevertheless, in this section we provide a brief overview of several techniques so the reader is aware of their capabilities and limitations.

Indoor localization is typically carried out into three steps, namely coordination or synchronization, measurement or ranging and finally position estimation. Figure 2.1 illustrates these phases in a particular system. A central node, call it client, wishes to localize itself with respect to an infrastructure of four fixed nodes in the system commonly referred to as anchors (with

known positions). First the client sends an RF broadcast signal to synchronize the anchors and trigger a timer in each of them (Figure 2.1 left - synchronization). Then the client sends a slow broadcast signal, e.g., ultrasound, with known speed of propagation. When the anchors receive this second signal they stop the timer and compute the distance traveled by this signal since it departed from the client (Figure 2.1 right - ranging). Once each anchor knows its distance to the client, it can either send it back to the client or to a particular node, e.g., one specific anchor, that will use a suitable method with all the measured distances and anchors positions to compute the client localization within the anchors framework (position estimation, not explicitly shown in the figure).

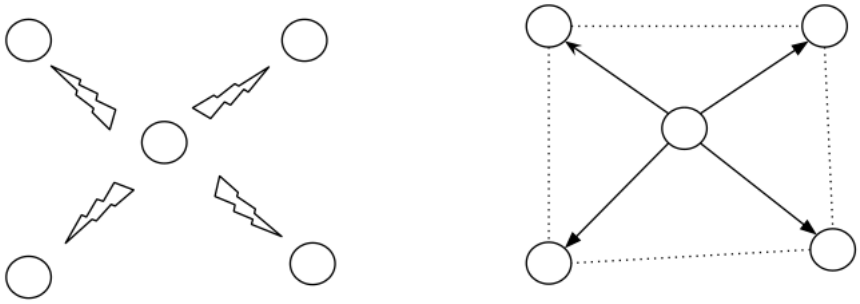

Figure 2.1: Localization of a node (center) with four anchors

For each of the steps referred above, there is a wealth of techniques, some more adapted to specific application scenarios and desired precision and accuracy than others. For example, the coordination or synchronization can be done with a broadcast signal as explained above or with high-precision clock synchronization.

Concerning the second step, there are techniques based on measuring angles, e.g., Angle-of-Arrival (AoA), and other ones based on measuring distances (ranging), either by time-stamping the signal arrival, e.g., Time-of-Arrival (ToA), or by measuring time intervals, e.g., Time-Difference-of-Arrival (TDoA), both requiring the knowledge of the measurement signal speed, or by measuring the power attenuation of the signal, e.g., by means of its Received Signal Strength Indicator (RSSI), which requires a model of the attenuation in space (Mao et al., 2007).

In what concerns the third step, the techniques depend on the measurements, naturally. *Angulation* is used when the measurements are angles. In a flat space, the angles to three non-co-linear anchors are the minimum needed for an unambiguous localization. This is called *triangulation*. If angles to more anchors are available, the technique is called *multiangulation* and it allows improving the localization precision. When the measurements are distances, the general technique is called *lateration*. Again, in a flat space, the distances to at least three non-co-linear anchors are needed, *trilateration*, and if distances to more anchors are available better precision can be achieved, *multilateration*. Figure 2.2 shows an example of trilateration, in which the anchors B1, B2 and B3 measured their distances to the node in the center and constrained the node to be on a circle centered on the respective anchor and with a radius equal to the respective measurement.

The node position is given by the intersection of all the circles. Finally, other more complex techniques to estimate the nodes positions include optimization methods, e.g., Multi-Dimensional Scaling (MDS) (Franco et al., 2017), and probabilistic methods (Xu et al., 2012).

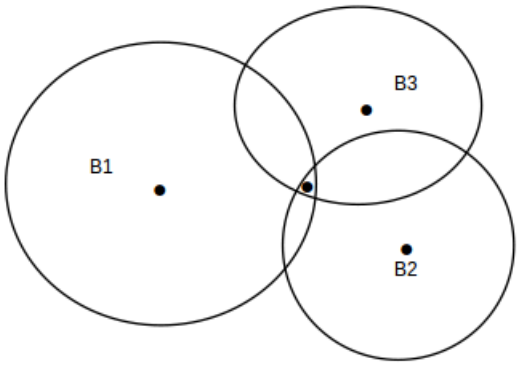

Figure 2.2: Localizing one node by trilateration with three anchors

In practice, there are many variants and combinations of the referred techniques to ensure the desired localization accuracy (Amundson and Koutsoukos, 2009). Nevertheless, some techniques are more common than others, frequently due to ease of use. This is the case of RSSI-based ranging, which is referred as the most common method. This is curious since, despite the easiness of use, RSSI-based localization is affected by noise, limiting the achievable accuracy to values that are frequently worse than required by many real use cases. Common problems affecting the accuracy of distance estimates based on RSSI measurements include path loss, fading and shadowing (Heurtefeux and Valois, 2012).

In the last decades there has been a significant amount of work in the localization problem. In the particular scope of wireless sensor networks, one work that stands out for its impact in the scientific community is the Cricket location-support system (Priyantha et al., 2000) for in-building, mobile, location-dependent applications. Cricket uses the method explained above in Figure 2.1, using both RF and ultrasound communication to measure distances and estimate nodes location. According to the authors, Cricket provides accuracy of 2cm and works in the range of 10m. However, the reliability of the system can degrade in case the communication is affected by multi path effects. Another work that also uses RF and ultrasound communications is the Dolphin system (Fukuju et al., 2003), also claiming an accuracy of 2cm, but inside a normal room with minimal manual configuration.

Differently from Cricket and Dolphin, Ecolocation (Yedavalli et al., 2005) uses RF communications, only. A node with unknown position sends an RF beacon that is received by a set of anchors. The system gathers and examines the ordered sequence of message receptions and the respective RSSI values, and then applies a constraint-based approach to estimate the position of the node. The authors claim a position estimation with less than 10% of error.

Another RSSI-based system, but aiming at simplicity of usage is EasyLoc (Jamâa et al., 2012). Normally, RSSI-based methods require a significant effort prior to deployment for profiling the RSSI-to-distance model in each location, aka fingerprinting. EasyLoc, instead, builds such model using known distances between anchors thus eliminating the need for such complex pre-deployment configurations. The authors claim EasyLoc provides location errors less than 1m with 90% probability, with an average of 0.48m for small spaces and 1.8m for larger spaces.

One application domain where precision requirements are less demanding while scalability and power efficiency are of foremost importance is precision agriculture, e.g., for pH or humidity sensing. Abouzar et al. (2016) propose an RSSI-based distributed Bayesian localization algorithm for precision agriculture in which nodes share broadcast messages to infer their most probable position, with fast convergence.

An approach that builds on RSSI-based localization systems and improves their accuracy taking advantage of mobility is Social-Loc (Jun et al., 2013). This is a middleware, initially deployed on Android systems, that exploits the encounters among mobile devices to calibrate the existing indoor localization systems and improve their accuracy. For example, improvements of over 22% were claimed in a system based on RSSI fingerprinting of WiFi communications. The MaWi localization system (Zhang et al., 2014), instead, improves the accuracy of WiFi systems with RSSI fingerprinting by combining it with geomagnetic information.

Other works studied the use of low frequency magnetic fields (in the kHz range) for localization underground, given the good propagation properties of these waves through the soil. For example, the Magneto-Inductive localization system (Abrudan et al., 2014) uses a locally generated magnetic field and the nodes to be localized are equipped with inertial measurement units (IMU) and tri-axial magnetometers. The RSSI at the magnetometer coils, with appropriate compensation for field distortion effects, is more stable than with common high frequency RF communications, hence providing more accurate localization.

Finally, a brief mention to yet another different technique, used by the Easy-Point (Bestmann and Reimann, 2014) localization system, which relies on the phase difference between sequential communications sent by the node to be located to a set of anchors using different frequencies in the 2.4 GHz band. It shows good resilience to multi-path effects and achieves an accuracy better than 0.72m.

Both this and the previous systems were demonstrated and benchmarked in the Microsoft Indoor Localization Competition in 2014, a competition that took place for more than ten years within the International Conference on Information Processing in Sensor Networks (IPSN) and until 2018[1], constituting a practical benchmark of the effectiveness of a large set of indoor localization systems.

[1] https://www.microsoft.com/en-us/research/event/microsoft-indoor-localization-competition-ipsn-2018/

2.2 Networking

The network is a central, thus critical, resource in WSN. The networking technologies that made WSN possible fall in a category generally called Low-power Wireless Personal Area Networks (LoWPAN). Within these, the link protocols that became more popular are part of the specification IEEE 802.15.4 and its amendments. On top of this specification, several full stack network protocols were developed, noticeably ZigBee (Alliance, 2006) and the IPv6-based 6LoWPAN (Shelby and Bormann, 2011). More recently, low-power embedded nodes with WiFi technology became available, enabling the use of this technology for WSN-like applications, particularly in the scope of the IoT and in environments with WiFi infrastructure (Pereira et al., 2019).

In general, these technologies offer a limited capacity. Even when using WiFi, the low power usage constrains significantly the available throughput. Thus, such capacity has to be adequately managed so it can be efficiently shared among the network nodes. This is particularly relevant when multiple applications run simultaneously on the WSN and when there are bidirectional communication patterns between the network gateway and the nodes or among the nodes themselves. Both aspects are considered in this thesis.

Hence, it is generally of interest to reduce the usage of the network in WSN and there are many works available in the scientific literature addressing such problem. The solutions cover various aspects ranging from link-layer protocols to network flooding and distributed TDMA solutions. Conversely, less solutions exist for the case of multiple applications.

One of the approaches that had significant impact in the community for efficient flooding in multi-hop topologies is Glossy (Ferrari et al., 2011). This protocol relies on precise synchronization among the network nodes (better than 0.5μs for IEEE 802.15.4) so that when two nodes in the same hop retransmit a packet for the next hop they do so at the same time, creating constructive interference. Not only the total number of transmissions needed for flooding is significantly reduced but the constructive interference increases the power with which packets are received, reducing errors significantly, thus improving reliability.

The efficiency of Glossy was then used to create the so-called Low-Power Wireless Bus (LWB) (Ferrari et al., 2012) that multiplexes in time flooding processes from one or more initiators. Since the data sent by each initiator reaches every other device in the network, the protocol emulates a bus-like behavior, independently of the network topology, even being multi-hop. However, the fact that all packets are flooded through the network, even if they are directed to a small subset of nodes within a few hops, leads to a high degree of unnecessary redundant transmissions. In the case of multiple applications, the situations of transmissions directed to different subsets of nodes located at different depths of the network topology become more common and the resulting overhead can become prohibitively large.

Other approaches aim at reducing the amount of time that a node keeps the RF interface active by defining a periodic wake-up scheme. These approaches can be divided in two groups, namely synchronous and asynchronous. In synchronous wake-up schemes, nodes agree on a common sleep/wake-up schedule (Ye et al., 2002; Van Dam and Langendoen, 2003) to save energy, which

is also frequently called *duty-cycling*. Asynchronous approaches are based on channel polling by receivers and preamble transmission by the transmitters (Polastre et al., 2004; Buettner et al., 2006). In this case, nodes that wish to transmit send a preamble followed by the actual packet. In turn, the receivers wake up periodically and sense the channel. If the channel is active, then the nodes stay awake to receive the packet transmissions, else they go back to sleep. This process requires that preambles are at least as long as the wake up period of the receivers.

An approach that was proposed specifically to improve network efficiency in scenarios with multiple applications is Unified Broadcast (UB) (Hansen et al., 2011). In this case, packets from broadcast services running in each sensor node, e.g., synchronization, heartbeats and data dissemination, are accumulated in the protocol stack. There they are transparently combined in UB broadcast packets and transmitted when the number of packets reaches a certain threshold, only. In their work, the authors also showed experimentally that UB preserves the correctness of a set of representative WSN protocols, such as FTSP (Maróti et al., 2004), Trickle (Levis et al., 2004) or CTP (Gnawali et al., 2009), even when their packets are delayed due to UB bundling. The periods of the applications are implicitly detected when a second packet of the same application is sent to the protocol stack for transmission and automatically recovered at the receiver side.

Finally, another large class of wake-up techniques relies on Time Division Multiple Access (TDMA) transmission control. In this case, nodes wake-up at the scheduled transmit/receive time instants, only, at a cost of tight synchronization requirements and limited flexibility to changes. TDMA for multi-hop networks typically uses 2-distance graph-coloring algorithms to build the transmissions schedules. Such protocols require much more information about the network topology, which can be acquired with a central coordinator, as in RT-Link (Rowe et al., 2006), or in a distributed way, as in Distributed TDMA (Herman and Tixeuil, 2004).

An interesting approach to create a TDMA round in a self-organizing manner, without clock synchronization, is proposed in Desync (Degesys et al., 2007). This approach follows the principle of pulse-coupled oscillators and creates a round that all nodes synchronize to, and then use different offsets to create non-overlapping slots. This approach was later on extended to multi-hop networks. A very similar approach is followed by the so-called Reconfigurable and Adaptive TDMA (RA-TDMA) family of protocols, which use maximum consensus and flooding to make the nodes converge to a common TDMA frame (Oliveira et al., 2018).

At the end of this thesis we will also build upon another TDMA-based approach that is particularly suited for compact schedules and energy-saving by maximizing nodes sleep time, namely Rate-Harmonized Scheduling (Rowe et al., 2008). In particular we will show how to extend this framework to a multi-application scenario and context-awareness.

2.3 Programming Wireless Sensor Networks

The technological progress of the last decades enabled many new use cases for WSN. However, the effective deployment and use of WSN depends significantly on the availability of easy ways to program the sensor devices and the respective network (Mottola and Picco, 2011). In this section,

we revisit commonly referred taxonomies for programming WSN such as (Sugihara and Gupta, 2008), (Mottola and Picco, 2011) and (Alajlan and Elleithy, 2015). We will also present a brief description of Contiki and Tiny OS, which have been the two most popular programming tools adopted by the WSN community. Finally, we take a view on the progress made in recent years in macroprogramming and in supporting multiple applications on WSN, topics that are of particular relevance to our work.

The taxonomy proposed by Sugihara and Gupta (2008) considers, at the highest level, two approaches to programming sensor networks, namely low-level and high-level programming models. The same classification is proposed by Alajlan and Elleithy (2015) in their analysis and taxonomy. However, despite having been published seven year later, this work goes little beyond the previous taxonomy, following essentially the same classification.

Low-level programming models are focused on abstracting hardware and allowing flexible control of nodes. These models take a platform-centric view and are also referred to as Node-level models. TinyOS with nesC is one of the earliest examples in this class and has been the de facto standard software platform for programming WSN nodes. An interesting approach in this class is to run a virtual machine on each node. A virtual machine provides an execution environment for scripts that are much smaller than binary codes for TinyOS. Thus it is appropriate for the situations where the code on each node needs to be dynamically reprogrammed after deployment via a wireless channel.

High-level programming models take an application-centric view instead of the platform-centric view and focus on programming the application logics, as opposed to each node individually. More specifically, they mainly focus on facilitating collaboration among sensors, which is a major drive of sensor network applications and also one of the most difficult challenges for sensor network programming. A typical approach in this class provides a set of operations for a group of sensors defined by several criteria. These operations include data sharing and aggregation so that programmers can describe collaborative data processing using them. Other approaches of high-level programming provide communication abstractions that use a simple addressing scheme like the access to a variable.

High-level programming models are further divided into two classes, namely group-level and network-level abstractions. Group-level abstractions provide a set of programming primitives to handle a group of nodes as a single entity. These define APIs for intragroup communications and thus make it easier for the programmers to implement collaborative algorithms. Network-level abstractions go beyond the notion of group by treating the whole network as a single abstract machine. The sensor database approach, which allows users to query sensor data by declarative SQL-like languages, falls within this class. Similarly, Macroprogramming languages that offer a macroscopic viewpoint are also in this class. In fact, macroprogramming is frequently referred as a synonym of network-level programming.

Finally, note that high-level programming models are often based on lower-level components as a runtime environment that allows them to run on actual hardware platforms. Figure 2.3 shows this entire taxonomy of programming models for sensor networks proposed by Sugihara and Gupta

(2008).

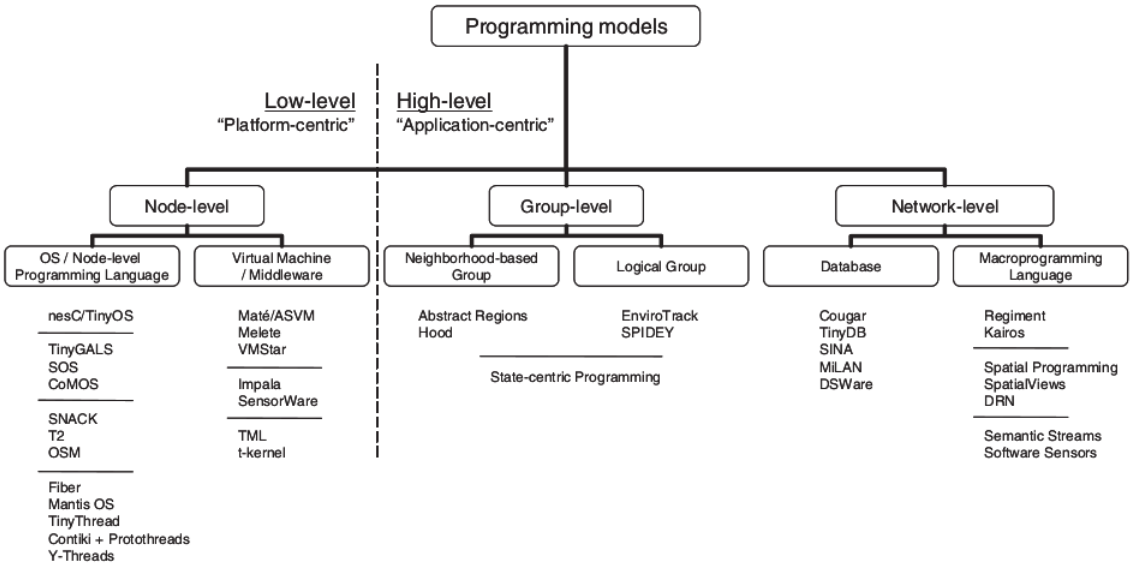

Figure 2.3: The taxonomy of Sugihara and Gupta (2008) for WSN programming models.

The survey presented by Mottola and Picco (2011) subsumes the previous taxonomies, addressing more dimensions, namely application, language and architecture, and proposing a specific taxonomy for each of them. This global view allows relating the application requirements with the programming solutions and separating the language and architectural aspects.

The WSN applications taxonomy focuses on the applications requirements. By understanding these requirements we can also understand the motivation that is behind the development of specific programming constructs. This taxonomy classifies WSN applications according to their *goal* (pure WSN or WSAN), *interaction pattern*, *mobility*, *space* (coverage) and *time* (temporal-control).

The taxonomy on language aspects considers six classes, namely, *communications scope, addressing, awareness, computations scope, data access model* and *programming paradigm*. The authors dedicate a significant space to the analyses and explanation of the primitives provided to the programmers to carry out communication and computation, discussing the specifics of each programming model. One of the most interesting aspects of this survey is the extensive use of concrete code examples, really facilitating the understanding of each approach.

The last taxonomy in the survey is on software architectural aspects of WSN. In this case, existing WSN programming solutions are analyzed with respect to the *programming support* (holistic or building blocks), *layer focus* (vertical or application), *low-level configuration* and *execution environment*.

For the sake of simplicity, the remainder of the section will follow the classification in (Sugihara and Gupta, 2008) concerning node-level, group-level and network-level programming abstractions. We are aware that it does not capture some detailed aspects referred in (Mottola and Picco, 2011), but it is enough for our purposes.

2.3.1 Node-level Programming

At the level of programming WSN nodes there are two classes of approaches, those that rely on node operating systems or programming languages and those that rely on virtual machines and specific middleware. The first class is the most popular and whithin it, the two most popular tools are Contiki and Tiny OS. In this section we present an overview of these tools as well as of a few virtual machine approaches.

Contiki

Contiki is a lightweight open-source operating system for the Internet of Things. It was developed at the Swedish Institute of Computer Sciences (SICS) by Dunkels et al. (2004). It is written in the C programming language and all programs for it are also in C. Contiki is a highly portable OS and it has already been ported to several platforms running on different types of processors. The most common are the Texas Instruments MSP-430 and the Atmel ATmega series of microcontrollers. Contiki relies on an event-driven programming model to handle concurrency. All processes share one single stack, allowing substantial memory savings, which is one of its main advantages. *Protothreads* are used to realize this model. Protothreads provide conditional and unconditional blocking wait and they use local continuations to save the state when they block. When the Protothread is resumed, it jumps back to the next instruction.

Contiki supports both IPv6 and IPv4 stack implementations, along with the recent low-power wireless standards: 6LoWPAN, RPL and CoAP. It also uses the Rime protocol stack. This is a lightweight communication stack for sensor networks and it has thinner layers than traditional stacks. The layers are simple and messages use small headers (only a few bytes). Rime also supports code reusing and the main purpose of this protocol stack is to simplify the implementation of sensor networks (Figure 2.4). Contiki is used in numerous systems, such as electrical power meters, industrial monitoring devices, alarm systems, remote house monitoring, city sound monitoring, street lights and radiation monitoring. The latest version of Contiki is Contiki 3.0.

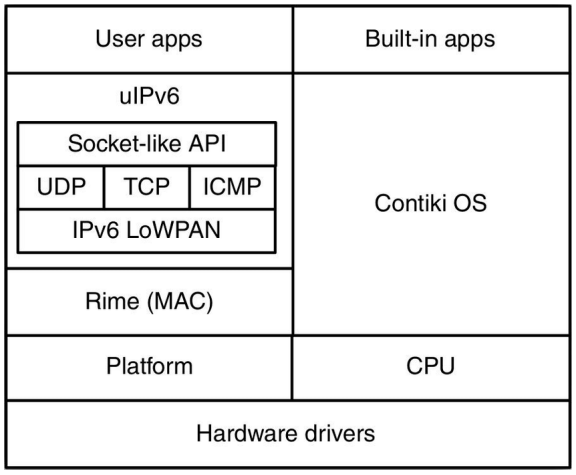

Figure 2.4: Contiki Architecture

A running Contiki system is made of the core, i.e., the kernel, libraries, protocol stack and the program loader, and a set of processes related to the specific application. Each process can be an application program or a service. The latter is a function that can be used by multiple application processes. All processes can be dynamically replaced at run-time. Interprocess communication is achieved through the kernel by posting events. For the sake of simplicity, there is no hardware abstraction layer and device drivers and applications communicate directly with the hardware. Moreover, the kernel provides just simple processor scheduling and event handling. Additional functionality is provided by system libraries and processes.

A process has two handler, an event handler and an optional poll handler. Its state is saved locally in the process memory. The kernel uses pointers to access process states. To give an idea of size, the process state uses 23 bytes in the ESB platform[2] that uses the MSP430 microcontroller (Dunkels et al., 2004). There is only a single global address space that is shared by all processes.

The composition of a Contiki system is defined at compile time specifically to each application. Typically, the core is compiled into a single binary that is loaded in the nodes before deployment, not being modified at runtime. Programs can also be linked statically to libraries in the core and loaded together with the core. However, programs can also be loaded by the program loader at runtime, either using the communication stack or from local storage, e.g., non-volatile memory. In this case, the programs can be linked with libraries that are part of the loadable program or the programs can use libraries implemented as services, which they invoke at runtime. These libraries can also be replaced at runtime. The kind of libraries to use depends on how frequently they are used and by how many programs. Widely used libraries are more efficiently implemented in the core, saving space in the loadable programs. Conversely, libraries that are local to just one program are better implemented in the loadable program and save space in the core. Libraries that may need to be frequently updated during the system lifetime are better implemented as services.

One very important component of the Contiki ecosystem is Cooja (Osterlind et al., 2006), the Contiki network simulator. It is a Java-based application with a graphical user interface. The GUI, shown in figure 2.5, is based on Java standard Swing toolkit. Cooja also supports simulation of the radio channel and integration with external tools, providing an emulation framework that offers additional features to the application. It can simulate large and small networks of Contiki motes (simulated sensor modules). Motes can be emulated on less detailed level, which is faster and allows simulation of larger networks, or at the hardware level, which is slower but allows precise inspection of the system behavior. This tool has two emulator software packages: Avrora for emulation of Atmel AVR-based devices, and MSPSim for emulation of TI MSP430-based devices. Given the higher popularity of the latter, MSPSim ends up being the most used software package for simulation of WSN.

Cooja can emulate multiple node platforms like the TelosB/SkyMote, the Zolertia Z1 mote, the Wismote, the ESB and the MicaZ mote. It is a very useful tool for Contiki applications development and debugging. It allows developers to test their code and systems before running them on

[2] http://www.scatterweb.com

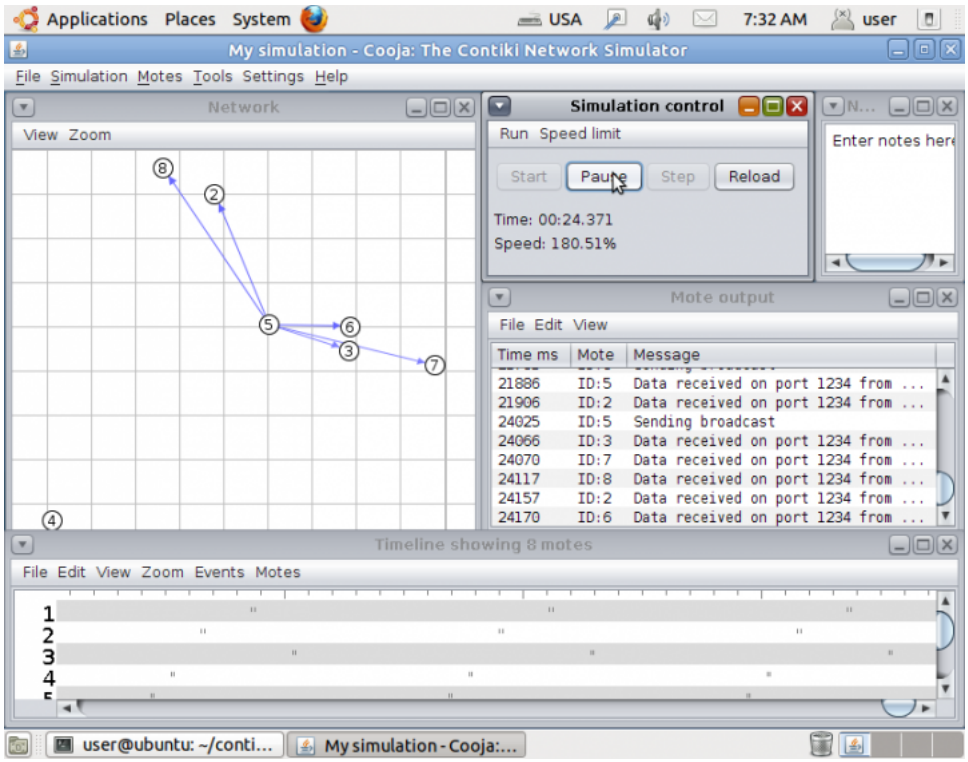

Figure 2.5: The Cooja simulation tool

the real target hardware, for example, to estimate power consumption of nodes or to analyse radio transmissions and receptions.

More recently, a new branch of Contiki has been created named Contiki-NG. This new version is more focused on dependable IPv6 communication. In addition, it is meant for modern Internet of Things platforms such as the ARM Cortex M family of microcontrollers. However, this is a recent development and yet to be used in real WSN deployments.

Tiny OS

TinyOS (Levis et al., 2005) is an operating system specifically designed for WSNs. It has a component-based programming model, provided by the nesC language, a dialect of C. Similar to Contiki, TinyOS is not an OS in the traditional sense. It is a programming framework for embedded systems that includes a set of components that enable featuring each application with its own application-specific OS.

A TinyOS program is formed by a graph of components and each component is an independent computational entity. Each TinyOS component has a frame, which is a structure of private variables. i.e., accessible to that component, only. Components can make use of three computational abstractions: commands, events, and tasks. The former two support inter-component communication and the latter allows expressing intra-component concurrency.

Commands and events are like the two directions of non-blocking service invocations. The actual invocations are done with commands that return immediately and the service completions

are signalled by corresponding events that trigger associated handlers. Events can also signal other asynchronous occurrences, such as hardware interrupts or message arrivals.

Commands can also be used to post tasks, i.e., functions to be scheduled by TinyOS for execution at a later time, concurrently. This can be very useful to decouple in time the execution of a component from specific complementary computations that can take longer. Tasks follow a *run-to-completion* execution model, i.e., they run once, and they can access their component's frame, only. The scheduling policy used by TinyOS to schedule tasks is FIFO. In TinyOS all hardware resources are abstracted as components. Thus, it is possible to issue commands to invoke hardware functions and use events handlers to receive the respective outcomes.

To help application developers, TinyOS offers a large number of already defined components that can be wired to application-specific components to compose applications. These TinyOS components provide typical services, some abstracting hardware components, such as sensors, single-hop networking, ad-hoc routing, power management, timers, and non-volatile storage. These components are available to application developers through interfaces (Table 2.1).

Interface	**Description**
ADC	Sensor hardware interface
Clock	Hardware Clock
EEPROMRead/Write	EEPROM read and write
Hardware ID	HardwareId access
I2C	Interface to I2C bus
Leds	Red/yellow/green LEDs
MAC	Radio MAC layer
Mic	Microphone interface
Pot	Hardware potentiometer for transmit power
Random	Random number generator
ReceiveMsg	Receive Active Message
SendMsg	Send Active Message
StdControl	Init, start, and stop components
Time	Get current time
TinySec	Lightweight encryption/decryption
WatchDog	Watchdog timer control

Table 2.1: Interfaces provided by TinyOS, adapted from (Levis et al., 2005)

Virtual Machine / Middleware

Another class of node-level programming models is based on using a virtual machine (VM) that enables the sharing of resources and/or infrastructure among multiple independent entities. One of the main purposes of these solutions has been reprogrammability. This is clear in the early VMs that were proposed for WSN.

For example, Mate (Levis and Culler, 2002) is one of the first VMs for WSN and it was developed on top of TinyOS. It is a stack-oriented application-specific VM that offers a small and application-tailored instruction set to support a safe and efficient programming environment. Mate

aims at supporting code propagation in WSN to re-program sensors in an energy efficient way. For this purpose it is important to produce lean programs to reduce the overhead. Programs are broken into small *capsules* of code that are transmitted throughout the WSN with a single command. A small piece of code in each node identifies whether the node must retrieve a passing capsule to run the respective program locally. Programs can run several tasks, which are executed sequentially.

Another historical VM for WSN nodes aiming at reprogrammability is VMStar (Koshy and Pandey, 2005), a Java-based framework to build application-specific virtual machines that allows updating both the WSN applications and the OS itself. VMSTAR also provides abstraction of the platform, maintaining code portability. It supports multi-threaded application programs that execute concurrently.

However, VMs were not the only way for reprogrammability. Other early works achieved it by means of middleware, a software layer that provides adequate abstractions to the applications. For example, Impala (Liu and Martonosi, 2003) is a middleware that was designed for the ZebraNet project to provide modularity, adaptability to dynamic environments and repairability to the applications. The modularity, particularly, supports easy and efficient on-the-fly reprogramming over the wireless channel.

Since these early works, many other developments occurred, some focusing on other specific aspects. For example, Yu et al. (2006) presented a significant extension of Mate called Melete specifically designed to support multiple concurrent applications, which is particularly relevant to this thesis. With Melete, users separately compile the code of their applications. Then the framework creates and maintains a dedicated execution space for each application. This allows constructing multiple application-specific environments in each node. Another interesting feature of Melete is the capacity to limit the deployment area to which an application code is to be sent, instead of sending to the whole network. Overall, Melete creates a flexible runtime environment that executes simultaneously in each node multiple applications. However, the framework did not address the node heterogeneity issue.

More recently, Raee et al. (2018) considered the case in which the WSN executes a set of applications, each with different sensing tasks. They proposed a method to assign these tasks to physical and virtual sensors that leads to the lowest energy consumption. This flexibility for task assignment builds directly on the virtualization layer. Similarly, Wu et al. (2021) also considered the general problem of mapping applications to nodes, physical or virtual, but motivated by the problem of *survivability*. In this case, the virtualization layer is used to support the re-mapping of applications tasks to nodes in case of node failures, for application resilience.

Along these recent year, the concept of virtualization expanded to the network-level, too, giving rise to the so-called Virtual Sensor Networks (VSN), which use subsets of the physical WSN as logically independent networks. This concept, however, goes beyond the node-level approaches. Up-to-date thorough surveys of the topic of virtualization in WSN can be found in (Khan et al., 2016) and (Farias et al., 2016).

2.3.2 Group-Level Programming

Some WSN applications, such as environmental monitoring and target tracking, benefit from the capacity to address multiple nodes located in a given area instead of a single sensor. Applications such as environmental monitoring also require the nodes in a certain area to collaborate among them and produce consolidated data such as averages. Programming these applications can be significantly simplified if specific group-level abstractions are available. For example, consider language constructs that allow assigning sets of operations to sets of nodes. These group-level programming abstractions are typically divided in two classes according to how the group is defined, whether by physical neighborhood or logically.

Neighborhood Based

WSN applications that monitor continuous processes in space naturally fit the concept of physical neighborhood. For example, the measurements done by sensors that are physically near each other can be processed together to consolidate their readings in a shorter but more robust data to be sent up through the WSN sink. This kind of local collaboration is rather common in WSN applications and also fits well with the broadcast nature of the wireless communication, which allows efficient data dissemination among neighbor nodes.

One of the emblematic approaches offering neighborhood-based group-level programming abstractions is Abstract Regions (Welsh and Mainland, 2004). It provides spatial operators that support neighborhood discovery, variable sharing and reduction operations. In Abstract Regions, groups are defined in terms of radio connectivity, geographic location, or other properties of nodes. For example a group can be defined including all nodes within N-radio hops from a given node, or including its k-nearest neighbors. In addition, Abstract Regions expose the trade-off between computations accuracy and communication bandwidth. This allows not only the programmer to tune the accuracy according to the network capacity available but also the applications to adapt to changing network conditions. Abstract Regions were implemented in the TinyOS programming environment and demonstrated in adaptive sensor network applications.

Using neighborhood-based group programming is intuitive and easy, and can be efficiently mapped on top of the broadcasting network. On the down side, the neighborhoods depend on the topology which is frequently unknown at the time of development and can even vary at runtime. This raises the importance of using adaptive approaches, as provided by Abstract Regions, to allow the application to adapt to changing neighborhoods.

Logical Group

Certain WSN applications, such as target tracking, may require the collaboration of a rather dynamic group of nodes. In this case, defining groups based on physical closeness is not adequate since such groups are essentially static. Conversely, a convenient approach may be to define a group according to some adequate logical properties beyond physical closeness, such as the type of nodes and the instantaneous input acquired from the environment. This would allow building

highly dynamic groups with rapidly changing group membership, i.e., nodes quickly joining and leaving the group as determined by dynamic properties such as target detection.

We refer to two ionic approaches of this kind. The first one is EnviroTrack (Abdelzaher et al., 2004), which provides a convenient and powerful interface to the application developer to facilitate programming WSN applications that track dynamic objects in the physical environment. For this purpose, EnviroTrack uses addresses assigned to physical events in the environment. For example, a set of sensors detecting the same event are automatically integrated in a group. As other group-based approaches EnviroTrack also provides data sharing and aggregation features. However, it includes a sophisticated distributed group management protocol to support highly dynamic group membership.

The second approach is integrated in the SPIDEY language (Mottola and Picco, 2006) for WSN programming, which defines the notion of logical neighborhood. Each node has a logical representation that includes exported attributes, some of which are static, e.g., node type, and others are dynamic, e.g., sensor data. A logical neighborhood can be defined using a predicate that conditions these attributes. This allows joining the intuitive concept of neighborhood with that of dynamic groups. SPIDEY provides APIs to communicate within the logical neighborhood that are supported by an efficient routing mechanism.

2.3.3 Macroprogramming

Network-level programming treats the whole network as a single abstract machine. Two typical approaches can be found in this class, the database approach, in which the whole WSN is managed like a real-time database accessed with queries, and macroprogramming that provides high-level programming constructs to control the behavior of the whole network. The latter is the most common and it is frequently taken as a synonym of network-level programming (Mottola and Picco, 2011).

Macroprogramming has been a popular area of the research in WSN for more than a decade, already. Some well-known works include Regiment (Newton et al., 2007), Abstract Task Graph (Bakshi et al., 2005), Kairos (Gummadi et al., 2005) and Flask (Mainland et al., 2008), from which we give an overview of the first two.

Regiment is a functional reactive programming model, which treats the outputs of sensor nodes as *streams*. A programmer can write functionalities based on these streams instead of worrying about the nodes. There are some basic primitives, such as rmap, rfilter and rfold, to operate on these streams, as shown in Figure 2.6. The streams can be combined into groups which are called regions. Due to using a functional approach, Regiment provides a high level of accuracy.

As opposed to Regiment and its functional approach, Abstract Task Graph (ATaG) follows a data driven model. In ATaG, every application is divided into three declarations: *Abstract Tasks, Abstract Data* and *Abstract Channels*. Abstract Tasks represent the type of processing in any application, Abstract Data represents the type of data handled by the applications and Abstract Channel associates the task declaration with data declaration. Using these declarations any application can

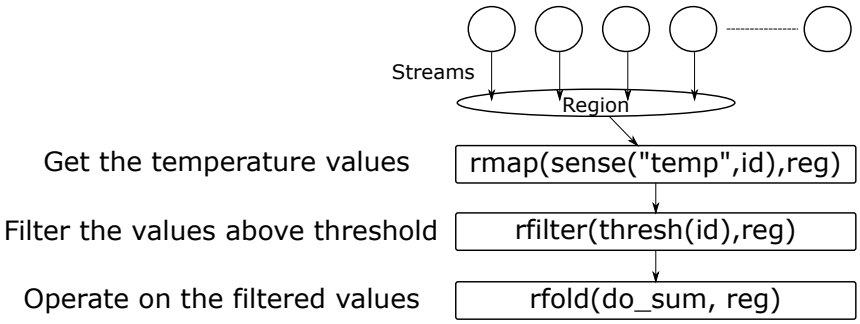

Figure 2.6: Regiment Macroprogramming with some basic primitives

be described by a model, as shown in Figure 2.7, and then this model can be instantiated any number of times through out the sensor network. ATaG provides abstraction because the number and placement of applications can be determined at compile or run time according to target devices.

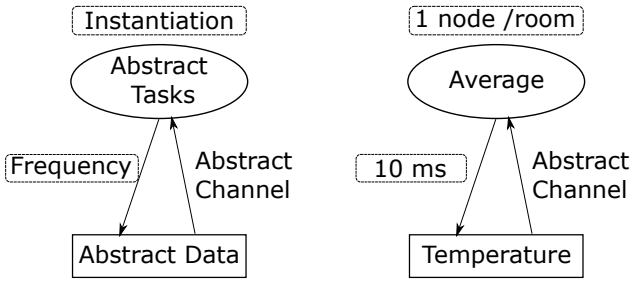

Figure 2.7: Abstract model of an application in Abstract Task Graph and its instantiation

One of the limitations of these early works is that they do not take into account the potential diversity of devices in the sensor network. They essentially considered homogeneous WSN. However, such diversity became more relevant with time given the growing availability of more capable IoT devices. These devices bring in new capabilities, e.g., stronger in-network processing, and imply new challenges, too, e.g., presence of different software and hardware platforms and need for more complex resource management. Some of these challenges were already addressed in earlier works. For example, Nano-CF (Gupta et al., 2011) is a macroprogramming framework designed to support in-network processing and resource management to enable concurrent application execution. This particular aspect of multi-application support will be further developed in the next section.

Another example is T-Res (Alessandrelli et al., 2013), which deserves a special reference for the influence it had on our work. T-Res also follows a macroprogramming approach to decouple the IoT applications from the network infrastructure while still supporting in-network processing. This allows the user to think about applications in an abstract manner, offloading any responsibilities regarding the resources needed for the actual computations. One important feature of T-Res is the use of IPv6 and CoAP, building on the concept of resources that can be control through REST operations, thus increasing interoperability and modularity. In Chapter 3 we will introduce T-Res in more detail and show how its features can be used to support mobility.

More recent work embraced the diversity of IoT devices. For example, Giang et al. (2016) propose a programming model that focuses on coordinating heterogeneous IoT distributed systems in the context of smart cities. Other complete framework-based IoT solutions that address interoperability can be found in (Hatzivasilis et al., 2018; Gaglio et al., 2019).

In (Giang et al., 2016), which we referred just above, one aspect that authors also considered, and which is particularly relevant in the scope of our thesis, is dynamic user requirements. These concerns with dynamic behavior became growingly important during the last decade leading to significant amount of related recent work. For example, Afanasov et al. (2018) propose a set of tools to define, implement and verify desired adaptive behavior in WSN. This topic will be further extended later in this chapter in the scope of context-awareness and adaptation and/or reconfiguration to multiple contexts of operation, which is central to our thesis.

2.3.4 Multi-application support

The evolution of the technology behind WSN brought significant increases in hardware integration, effectiveness and efficiency. This is visible in the typical 8-bit microcontrollers used in WSN nodes. For example, the relatively recent ATMEGA128RFA1 microcontroller uses an AVR enhanced RISC architecture capable of executing powerful instructions in a single clock cycle with a fully static design granting a throughput close to 1 MIPS per MHz. More importantly, the static design allows the designer to tune and adapt the trade-off power consumption versus processing speed. Moreover, integrated with the processor is a high data rate transceiver for the 2.4 GHz ISM band that has hardware assisted features, e.g., frame handling. This grants the hardware platform sufficient capacity to multiplex several concurrent applications with low energy consumption. This feature is further motivated by the capability to handle multiple sensors integrated together on the same node with low impact on cost, which can enable different concurrent sensing tasks.

Naturally, this evolution created the opportunity to turn WSN into networks of nodes on which one or more users can deploy multiple applications that may run at the same time. This requires specific support in complementary directions, such as to write applications transparently to each other, to abstract the specific IO, to develop protocols to deploy and control applications, and architectures to track resource usage, from processing and communication resources to memory and specific IO. At the end, the goal is to improve scalability, flexibility, adaptivity and energy-efficiency of WSN physical deployments.

The path towards supporting multi-application execution naturally crosses several of the dimensions of the classification on programming models that we addressed before. This is expected since the programming models provide the abstractions on which multi-application execution lies. This is just to say why we will refer again to some works that were already referred in the previous sections.

One early work towards supporting multiple applications that had reasonable impact in the WSN community is (Yu et al., 2006). It proposed the Melete system, which supported multiple applications based on the node-level VM Mate. Virtualization techniques have frequently been the enabler of multi-application execution. Ultimately, the whole physical WSN is virtualized,

offering the users logical subsets of the network that they can use transparently as a separate network on its own. These became know as VSN (Sarakis et al., 2012), as we already referred before, and gained significant adherence in the current WSN landscape (Khan et al., 2016).

SenShare (Leontiadis et al., 2012) is another example of using abstraction to support multiple applications. It is an early work towards virtual sensor networks, building upon the concept of overlay networks. SenShare separates the infrastructure from running applications in such a way that each application runs on its isolated environment, i.e., an overlay network that accesses node hardware through an abstraction layer.

Other frameworks were developed to support multiple applications focusing on simplifying in-network programming. The purpose was to allow users to deploy applications on the network without necessarily knowing programming languages. CITA (Ravindranath et al., 2012) is one of the finest examples of how users can deploy multiple applications that run at the same time over a mobile sensing platform. It provides abstraction between building the application on the user side and programming it on the server side. The user can simply give commands in laymen terms such as "turn off WiFi while outside the home". The programmer, in turn, programs the server side in a platform-independent way, writing code that can be deployed on one or multiple devices seemingly.

In a similar line, other frameworks used macroprogramming to provide abstraction and support multiple applications. PyoT (Azzara et al., 2014) is one such example, aiming at the simplification of development for applications that coordinate activities across groups of nodes. The framework abstracts nodes functionality and low-level communication details, allowing programmers to focus on high-level application goals. This abstraction also facilitates distribution, supporting in-network processing applications and providing a distributed sensor data storage system that can be used as data sensor caching to reduce communications. Similarly to our work, PyoT is built on top of T-Res, referred in the previous section, using IPv6 and CoAP resources, but it adds a resource monitor capable of resource discovery and the possibility to program WSN applications as Python scripts. However, while PyoT aimed at simplifying the development of complex applications for the Internet of Things, our work aims at providing autonomous reconfiguration mechanisms that allow applications to continuously adapt to varying execution contexts.

An almost parallel thread to PyoT is PyFUNS (Bocchino et al., 2015). Both allow programming WSN applications with Python scripts and use IPv6 and CoAP to control applications in the network, providing abstraction and modularity. However, PyFUNS builds directly on Contiki and uses a Python-based small footprint virtual machine, namely PyMite, to isolate and run applications. PyFUNS, similarly to PyoT and particularly T-Res, has been very influential in our work.

A substantially different focus is found in this early work, UMADE (Bhattacharya et al., 2010). This framework focuses on Cyber-Physical Systems and on assigning nodes to applications according to a quality-of-monitoring (QoM) metric. This metric, for any physical parameter, depends on the measurements from various associated sensors. One node's contribution to the QoM of the whole application depends on other nodes as well. Hence, all such nodes with QoM

dependencies are allocated to the same application, leading to a positive effect on QoM.

Finally, we present another early work that supports multiple applications on a WSN, focusing on yet another perspective, the management of system resources in this case. It is called Nano-CF (Gupta et al., 2011) and it supports the deployment and concurrent execution of multiple independently written applications. Nano-CF builds upon the Nano-RK (Eswaran et al., 2005) operating system that manages the system resources. However, Nano-RK lacks adequate support to network communications scheduling. This limitation was handled later on by Gupta et al. (2014) with the so-called Network Harmonized Scheduling (NHS). NHS is a technique that schedules packets from different applications on same node in such way that the radio transmission remains coordinated across the network. NHS uses a harmonizing period to transmit all the packets from the node, which is inspired on Rate Harmonized Scheduling (Rowe et al., 2008). It is a fully distributed approach that works well in multi-hop networks that lack a central coordinator. Later on, in Chapter 6, we will show how our work can benefit from NHS and extend it to support dynamic application settings.

2.4 Context-Awareness and Mobility

In the previous section we addressed WSN programming and we saw different programming abstractions, many of which providing specific support to aspects like application development and deployment, resource allocation, energy management and concurrent applications.

A specific topic we referred a few times but did not cover in detail, and which is central to our thesis, is dynamic adaptation. This is particularly relevant when the WSN applications need to make an efficient use of the system even when the operational scenarios change, be it due to changes in physical parameters such as location of nodes, i.e., mobility, or in the system structure such as availability of nodes, energy levels, etc. For this purpose, programmers need to account for resources globally and anticipate possible future scenarios and thus program actions to adapt to such scenarios.

These concerns frequently extend beyond the strict domain of WSN into broader areas such as IoT and networked mobile systems. Nowadays, it is common to have smartphones and wearable devices contributing to, or integrating, general WSN systems. Because of the intrinsic mobility of these devices, such systems are also referred to, in the research literature, as Mobile Sensing Systems (Macias et al., 2013). In these systems, the sensing can be performed by both mobile devices and traditional WSN devices together (Figure 2.8). For example, a smartphone can provide temperature data, a wearable device can complement a motion sensor for better accuracy while a smart card or RFID tag can provide user presence and identification information.

Therefore, WSN, pushed by these mobile sensing systems, have become much more pervasive and ubiquitous than ever before. Devices can collect more quantity of data about the user and also different types of data such as location, temperature, motion, etc. Such broad expansion of data collection can allow a system to have a better understanding of its users and environment. It is

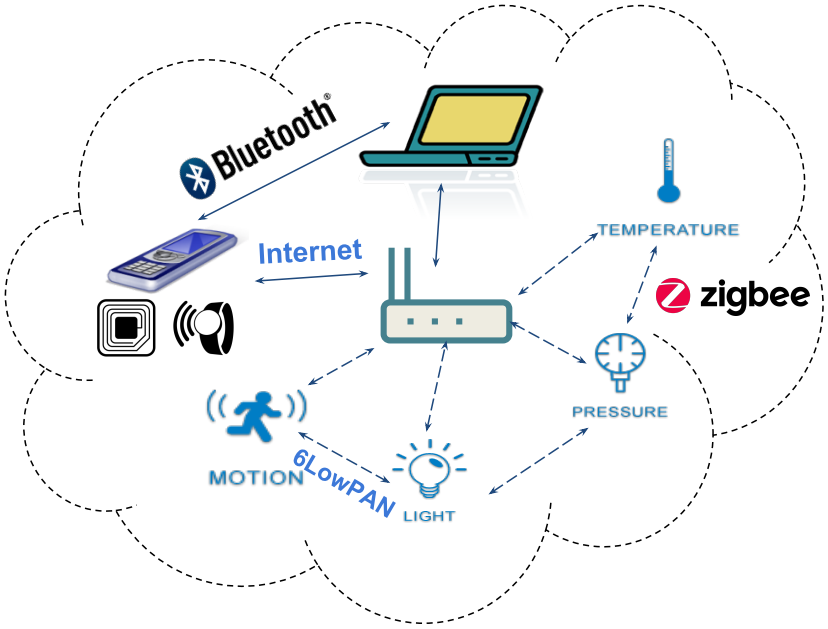

Figure 2.8: WSN and Mobile Sensing Systems.

now possible to infer different operational scenarios during system run-time and also adapt based on such scenarios.

Mobility, in particular, is crucial as it is one of the main triggers of operational changes. Moreover, discussing operational changes requires a structured look on *what* actually changes, which leads to the concept of *context*. In this section we will revisit the notions of context and context-awareness in mobile systems that are keystones in our thesis, followed by an overview of context-aware middleware architectures and context-aware programming, with an emphasis on the WSN and IoT domains.

2.4.1 Basics of Context-Aware Mobile Systems

Context-aware mobile systems are a type of mobile systems that can observe the physical environment around them and adapt to it. Such systems are generally categorized as pervasive or ubiquitous systems. These systems offer *anytime, anywhere, anyone* computing by creating clear separations, i.e., interfaces with adequate abstractions, between devices, applications and users. For a system to offer *anytime, anywhere, anyone* computing it must be aware of three key aspects related to itself: i) where it is, i.e., the environment, ii) who it is with, i.e., the user, and iii) which other resources are nearby, i.e., other systems. Some examples of these aspects are nearby people, sensor or actuator devices, lighting conditions, noise levels, network connectivity, location of devices, and even social status of the user. Those three aspects combined together can represent an entity called *context*. Context can capture the status of the system, needs of the users and the activities in the environment.

The concept of *context-awareness* goes back to early research on ubiquitous computing conducted at Xerox PARC in the early 1990s. The term *context-aware* was coined in an experimental work at PARC facilities with mobile devices called PARCTAB (Schilit et al., 1994). In this work, context-awareness refers to context as location, identities of nearby people and objects, and changes to those objects. In a contemporaneous work, Brown et al. (1997) define context as location, identities of the people around the user, the time of day, season, temperature, etc. These works define context to be the *constantly changing execution environment*, including i) computing, ii) user and iii) physical parts of the environment. Abowd et al. (1999) frame the concept in a better defined form that we reproduce here, for the importance it has within our thesis:

> **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.

This definition makes it easier for an application developer to enumerate the context for a given application scenario. If a piece of information can be used to characterize the situation of a participant in an interaction, then that information is context.

A categorization of context can help application developers uncover the pieces of context that, most likely, will be useful in the application itself. The previously referred descriptions of context lead to different context types. However, there are certain types of context that are more relevant than other. Those types we consider more relevant within our scope, i.e., WSN and IoT systems with mobility, are *location*, *identity*, *activity* and *time*. The location type helps identifying *where*, identity helps with *who*, activity helps with *what* and time helps with *when*. To characterize a context properly, all of these four types can be used in different combinations.

Context-aware mobile system were first defined by Schilit and Theimer (1994). In their work, the authors referred to such a system as software that *adapts according to its location of use, the collection of nearby people and objects, as well as changes to those objects over time*. Rephrasing to refer explicitly to the notion of context, we also reproduce here the definition of *context-aware system* as initially given by Abowd et al. (1999):

> A system is **context-aware** if it uses context to provide relevant information and/or services to the user, where relevancy depends on the user's task.

In 2009, Hong et al. (2009) provide one of the first extensive literature reviews of context-aware systems, covering the research carried out in this topic in the first decade of the years 2000. This review proposed an abstract layered architecture for context-aware systems. According to this study, such architecture is composed of the following four layers:

- *Network infrastructure layer* that involves the network that supports context-aware mobile systems and allows sensors to collect low-level context-related information;
- *Middleware layer* that manages processes and context-related information, as well as their instantiation and interactions to support desired abstractions;
- *Application layer* that provides users with services and abstractions that are relevant for different classes of applications;
- *User infrastructure layer* that offers suitable interfaces for interaction with the user.

Beyond these layers, Hong et al. (2009) also refer to a *concept and research layer* that includes the basic research on the concepts that support context-aware systems. However, in the following we focus on the middleware layer, for its importance in the operation of context-aware systems and in our thesis, and also because it was not specifically covered in the previous sections.

2.4.2 Context-aware Middleware Architectures

Middleware generally refers to a software layer encompassing operational information, its organization and the processes that manage it, to support desired abstractions that are not natively provided by the underlying platform, typically an operating system. In the scope of context-aware systems, it is the middleware that supports acquiring contextual information, reason about it using different logic and then adapt to changing contexts. In their review, Hong et al. (2009) propose a classification of middleware for context-aware systems in six categories that we refer next with existing examples.

Agent-based middleware relies on mobile agents. The Apricot Agent Platform (Alahuhta et al., 2006) is an example designed for developing context-aware, personalized and user-friendly mobile services. This agent architecture consists of an agent platform, agents and agent containers, that provides built-in functionality and communication mechanisms.

Reflective middleware is characterized by its distinguishing ability to model itself through self-representation, allowing the manipulation of its behavior. An example of this category is the Mobile Platform for Actively Deployable Service (MobiPADS) system (Chan and Siu-Nam Chuang, 2003). It provides an execution platform that supports active service deployment and reconfiguration of the service composition following varying contexts, adjusting the configuration of resources to optimize the operations of mobile applications.

Metadata-based middleware uses metadata such as information on users, devices, resources and runtime binding strategies to carry out dynamic binding of services. CARMEN (Bellavista et al., 2003) is an example of such a middleware that combines metadata with mobile agents to achieve component reusability and automatic service reconfiguration upon context changes. The context is determined using metadata, including declarative management policies and profiles for user preferences, terminal capabilities and resource characteristics.

Tuple spaces-based middleware uses a blackboard, i.e., a shared repository of information, to hold partially known information in the form of a tuple-space and share it among applications.

TinyLIME (Curino et al., 2005) is an extension of a previous MANETs middleware to the scope of WSN that provides access to sensors through a tuple space interface. Mobile monitoring stations access the sensors within a proximity range and perform location-dependent data collection.

Adaptive and objective-based middleware can be found in smart-homes as well as generally in WSN. It uses a quality-based objective function to optimize the matching between the application QoS requirements and the quality that a specific context can provide. An example is MidFusion (Alex et al., 2008) that aims at WSN and allows discovering and selecting dynamically the best subset of sensors that maximize QoS for the application and minimize the data acquisition cost.

OSGi-based middleware uses the OSGi specification[3], describing a modular system and a service platform for the Java programming language that implements a complete and dynamic component model. OSGi allows both applications and components to be remotely installed, started, stopped, updated and uninstalled without requiring a reboot. An example of an OSGi-based middleware is CMM (Zhiwen Yu et al., 2006), a context-aware multimedia middleware that supports multimedia content filtering, recommendation, and adaptation according to a dynamic context. It also performs context aggregation, reasoning, and learning. OSGi is key for device and service interoperability.

A more recent survey of context-aware middleware architectures is presented in (Li et al., 2015). They discuss the principles of context, context awareness, context modelling, and context reasoning to set a baseline to then provide an overview of eleven paradigmatic middleware architectures. These architectures are then compared on architectural style, context abstraction, context reasoning, scalability, fault tolerance, interoperability, service discovery, storage, security and privacy, context-awareness level, and cloud-based big-data analytics. As a conclusion, the authors stated that, at the time the paper was published there was no context-aware middleware architecture that complied with all requirements. We believe this is still true.

Li et al. (2015) refer to eleven paradigmatic middleware architectures, some of which could be included in the classes referred above. Nevertheless, we refer briefly to the following three for focusing on concepts that are particularly relevant to our thesis, namely heterogeneity of devices, simplicity of programming applications and contexts, efficiency in using resources and suitability for wide-spread usage.

FlexRFID (El Khaddar et al., 2015) is a policy-based middleware that aims at facilitating the development of context-aware applications and the integration of heterogeneous devices. Heterogeneity is supported on a Device Abstraction Layer (DAL) that abstracts the interactive operations among the physical network devices. The main feature of FlexRFID, though, is the capability of policy enforcement, which can be used to ensure privacy, constrain access control or offer customized services.

Octopus (Bernhard Firner et al., 2011) is an open-source and extensible middleware system for data management and IoT applications that is specifically designed to help non-technical people to deploy sensors, manage context and develop applications quickly. Octopus follows a distributed

[3] https://docs.osgi.org/specification/osgi.core/8.0.0/framework.introduction.html

node-based architecture and is organized in abstraction layers that separate the developer from data analysis and application manipulation.

Finally, we refer to FIWARE[4] as an example of a middleware aiming at wide-spread usage. It provides an infrastructure to support creation and delivery of services with high QoS and security, and which can be easily adapted to multiple domains, such as safety, logistics, environment, energy, traffic and mobility, and agriculture. Its architecture is composed of elements called enablers that manage aspects of the system, e.g., context, and support the three main middleware abstractions, namely *Actors*, *Resources*, and *Applications*. One specific type of enablers, the *Cognitive Enablers*, make use of metadata to take decisions regarding the efficient use of available resources while satisfying the application requirements.

In a recent work, Temdee and Prasad (2018) present another survey of context-aware middleware architectures, but it includes essentially the same examples as those already included in the previously referred surveys. Nevertheless, it also includes a survey of context-aware middleware applications, with a focus on smart environments such as smart homes and personalized spaces.

2.4.3 Programming for Context-Awareness

One particularly challenging issue in context-aware systems is their programming, which has to consider the applications operating in dynamic scenarios as well as the contexts that represent such scenarios. There has been some effort towards developing complete software solutions for context-aware systems, such as (Villegas, 2013), but they are still hard to program, putting significant responsibility on the programmer to manage the context. Conversely, there is not much effort towards providing abstractions for the sensor nodes to detect changes globally and adapt accordingly.

Among the existing approaches to tackle the programming of context-aware systems is Context-Oriented Programming (COP) (Hirschfeld et al., 2008a), a network-level approach according to the classification in Section 2.3. COP is a tool to modularize context-dependent behavioral variations in an application, where such variations can be dynamically switched on and off in response to changes of execution contexts. For example, a navigation program running on a smartphone displays a map with different resolution (behavioral variations) depending on whether the device is outdoor and indoor (execution contexts). Along the years, COP was further enhanced with specific programming techniques (Ghezzi et al., 2010b; Kamina et al., 2011; Salvaneschi et al., 2012; Sehic et al., 2011; Aotani and Leavens, 2016).

Another example of context-aware programming is EventCJ (Aotani et al., 2011), a Java-based language that combines existing modularization mechanisms for context-dependent behavioral variations with a novel event-based layer activation mechanism. It employs the layered partial methods that can also be found in other context-aware programming languages such as ContextJ (Hirschfeld et al., 2008b) and JCop (Appeltauer et al., 2010). There are many similar event-based programming languages which can be used for context-aware programming.

[4] https://www.fiware.org/

In the same way, there are several other examples of context-aware programming. For example, ContextL (Costanza et al., 2006) is an extension of Common Lisp Object System (DeMichiel and Gabriel, 1987) that describes features independently of the object model. The Erlang programming language can also be adapted to introduce context in a parallel or distributed system (Ghezzi et al., 2010a). This approach natively supports distribution and concurrency. Erlang comes with the OTP platform, which provides a rich set of libraries supporting reliable, large-scale, distributed and dynamically evolvable applications.

Afanasov et al. (2014) also focus on providing support for context-awareness, specifically for WSN. They note that when contexts change, so do applications, too. However, there is still a need for more support in such areas for programmers and end-users. Another complete framework is RTCOP (Tanigawa et al., 2019), which is based on C++ and aims at providing support to context-awareness in generic embedded systems, not just for WSN.

Context Toolkit (Salber et al., 1999) provides an extensive support to build context-aware applications using features like context widgets. This is a step forward in easiness of programming making this approach one of the closest solutions to the concerns underlying our thesis. Recently, following the general trend of using Machine Learning, the work in (Deniz et al., 2020) proposed using Deep Learning to detect context in Cyber-Physical Systems. This approach also alleviates the programmer since it uses a model that is defined through training and learning, but still these steps requires significant effort.

There are many more context-aware programming tools available (Gonzalez Montesinos et al., 2011). However, most are targeted to mobile systems with complex devices or continuous user interface. WSN applications must require less direct user interface once applications are deployed.

In spite of all the evolution and solutions reported above, there is still a lack of open and easy to use approaches to define any kind of adaptations, particularly the corresponding abstraction and its re-usability.

2.5 Summary

In this chapter we have discussed related research work in the major aspects of the WSNs, namely in localization, networking, programming and adaptivity. To highlight the importance of the problem underlying our thesis, we focused on programming wireless sensor networks, particularly for context-awareness, and looked at different programming approaches proposed by the research community roughly after the 1990s. In this review we did not find any context-aware middleware, particularly for WSN, that could support heterogeneous devices, high-level programming of applications and contexts suitable for non-expert users and efficient use of resources, while being suitable for wide-spread usage. This highlights the novelty and relevance of the middleware we developed and present in Chapter 4, which supports all these properties.

Chapter 3

Macroprogramming and Mobility

As we saw in the previous chapters, WSN applications were originally designed to perform a single task, repetitively, often in strictly controlled environments. This meant that programming applications for a specific system was a one-time task. Nowadays, WSN have become an integral part of the IoT, making it more pervasive, powerful, reliable and able to incorporate an increasing number of functionalities. These advancements also enabled the development of a self-adapting IoT that can respond to changes in context, such as location, activity, availability of resources or time, without need of actions from the users. This development is our target, towards Context-Aware Sensor Networks, but to achieve it we need adequate tools.

In this chapter we introduce and discuss two technical steps upon which we will develop our main contributions in the following chapters. These steps are T-Res (Alessandrelli et al., 2013) and its extension to support mobility mT-Res (Gaur, 2015). T-Res decouples the IoT applications from the network infrastructure while supporting in-network processing. Thus, it provides a fundamental property to our purposes, which is *abstraction*. However, despite its flexibility, T-Res does not consider another fundamental property that we need, which is *mobility* of nodes, particularly as a driver of context changes. Thus, we added a mobility feature to it and called it mT-Res (Gaur, 2015), which is a preliminary contribution of this thesis, as stated in Section 1.7. We will use the abstraction and the mobility of mT-Res to develop our main contribution, i.e., a context-aware middleware for sensor networks.

3.1 T-Res: macroprogramming for IoT

T-Res is a task-based approach to macroprogramming for IoT nodes. As mentioned in (Alessandrelli et al., 2013), T-Res is meant to support the following functionalities:

- Resource Monitoring: Using T-Res, it is possible to monitor one or more *resources* identified with IPv6 addresses;
- Data Processing: T-Res allows for processing of data collected and/or generated by resources;

- Connecting Tasks with Resources: T-Res connects *tasks* with different resources using both input and output connectors.

T-Res is able to support these functionalities by enabling representation of a task as a resource itself. For example, an IoT device executing a task would represent two resources, the device itself as one resource and the task as another resource. Similarly to the IoT device, this task is also assigned an IPv6 address that allows addressing and controlling it. This control is done using Constrained Application Protocol methods. These methods are used to create, remove, modify or retrieve resources. The CoAP methods used in T-Res are GET, POST, PUT and DELETE, as well as the OBSERVE feature. With these methods a user can build applications by connecting tasks to other tasks, or generally other resources, or assigning various resources to tasks including input and output devices. This also allows for resources to automatically retrieve tasks from other resources or other complex scenarios for automatically finding tasks or resources.

3.1.1 T-Res Basics

As referred above, in T-Res the concept of task is central. In order to connect tasks to resources in general, be it other tasks or input / output devices, T-Res assigns the following four *sub-resources* to each task:

- **Input Source** *(/is)* collects input data for the respective task from the devices declared here;
- **Processing Function** *(/pf)* is the actual code of the task that will process the referred input(s) and generate the expected output;
- **Output Destinations** *(/od)* is the set of destination devices where the output of the respective task is posted;
- **Last Output** *(/lo)* stores the most recent output generated by the respective task.

IPV6 URI addresses are used to assign input and outputs to */is* and */od*, respectively. An executable file is provided to */pf* as the task function. This function will consume the input obtained from */is* and generate an output that will be sent to */od*. Note that inputs are received with the OBSERVE feature of CoAP, which supports asynchronous communication. This allows triggering the task function */pf* automatically, as a callback function, whenever new input arrives at */is*. Finally, */lo* also has an assigned URI and makes available the output that was generated by */pf* in the last function invocation and this can also be used as input for another task. Table 3.1 summarises the sub-resource structure of a task together with the CoAP methods that are used to access the addressed resources.

3.1.2 T-Res Processing Functions

As previously mentioned, T-Res isolates the key data-processing part of a task from its input and output using processing function sub-resource. In order to have a fully realized macroprogramming approach, this processing function sub-resource has to be device independent. This

Table 3.1: The sub-resource structure of T-Res tasks and associated CoAP methods.

Tasks	Sub-resources	Possible actions	CoAP methods
Task_Name	input sources (/is)	fetch/update	GET, PUT, POST
	processing function (/pf)	fetch/update	GET, PUT
	output destinations (/od)	fetch/update	GET, PUT, POST
	last output (/lo)	fetch/observe	GET

allows the mobility of the code across multiple resources in a WSN. To enable this, the processing function is written in Python. Python is a scripting language that can be compiled directly into executable bytecode and allows for ease of access for various interpreters. In addition, Python scripts are commonly used for IoT devices and are getting more and more popular with machine-to-machine applications.

Moreover, T-Res also provides to the processing functions written in Python an API to access the task input/output data and state, namely:

- *getInput()* returns the value of the last input received;
- *getInput()* returns the tag of the last input received;
- *setOutput()* sets the output value;
- *getState()* retrieves the task state;
- *saveState()* saves the task state.

The tag is a string that identifies the source that triggered the data-processing function. This is relevant when there are multiple inputs that can trigger the processing function so that the function can select the right variable to receive the new input. The state is represented by an object and can be defined by the user and which allows preserving the history of the processing function execution. The example that we show in the following section clarifies these features.

3.1.3 Illustrative Example using T-Res

The need for the abstraction that T-Res provides becomes clear when similar functionalities are needed from an IoT system across multiple scenarios. For example, let us consider an IoT system where a temperature sensor is used to track the temperature of a room and actuate a heating system, accordingly. Most common heating devices already include the capability to sense temperature locally, but if additional resources, namely external temperature sensors, are available the system may give preference to those additional resources. This is demonstrated in Figure 3.1 where the same IoT temperature control system can be applied to two different scenarios transparently, namely a smart office and a smart home.

To better describe T-Res functionality, we follow a similar example to that provided in (Alessandrelli et al., 2013) applied to the smart home scenario above. Consider an abstract IoT application that keeps the temperature of a given space between two values, say LO and HI. The application

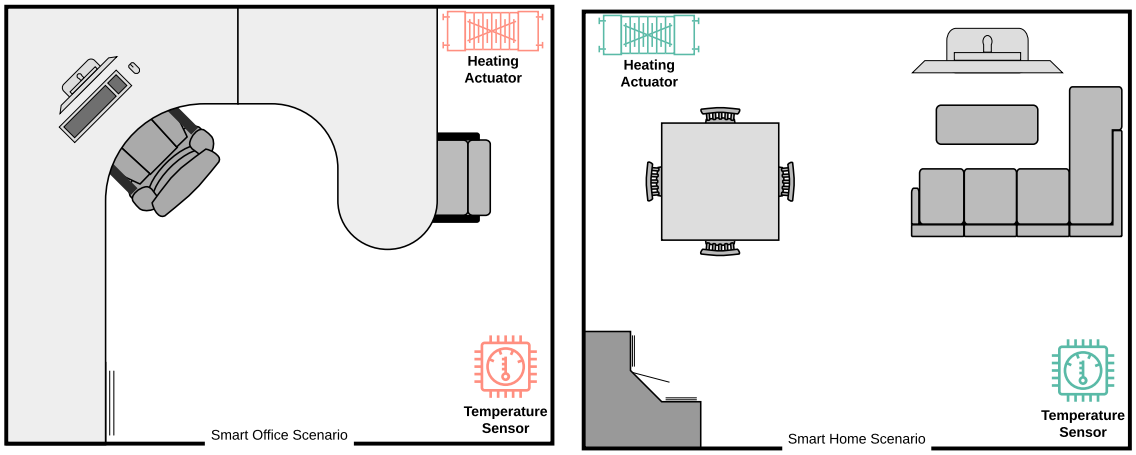

Figure 3.1: Example of an abstract IoT-based system applied to two different scenarios.

can operate with just a heating device provided with a local sensor. In our concrete case, let us consider this device is assigned the address *aaaa::1* and that the sensor and actuator are accessible through CoAP methods in the following URIs, respectively:

```
coap://[aaaa::1]/sens/TEMP–HEAT–LIVING
coap://[aaaa::1]/act/HEAT–LIVING
```

The sensor can be read sending a GET command, and the heater can be controlled ("on"/"off") sending a PUT command, to the respective URIs. This setting is shown in Figure 3.2.

In order to build the desired temperature control application we need to create a T-Res task in an existing resource. In this case, we create the task TEMP-CONTROL in the heating device[1]. This is achieved sending a PUT command to the following URI:

```
coap://[aaaa::1]/tasks/TEMP–CONTROL
```

Now we need to set the sub-resources of the task appropriately. At this point, the task will read from the local sensor and actuate in the local actuator. Thus:

```
/is:
    coap://[aaaa::1]/sens/TEMP–HEAT–LIVING
/od:
    coap://[aaaa::1]/act/HEAT–LIVING
```

The */pf* sub-resource is then set with the bytecode from the Python script that performs the desired action. In this example, the bytecode can be complied from the following Python script, in which HI and LO are adequate predefined constants:

[1]Note that a T-Res task can be created in any node in the system.

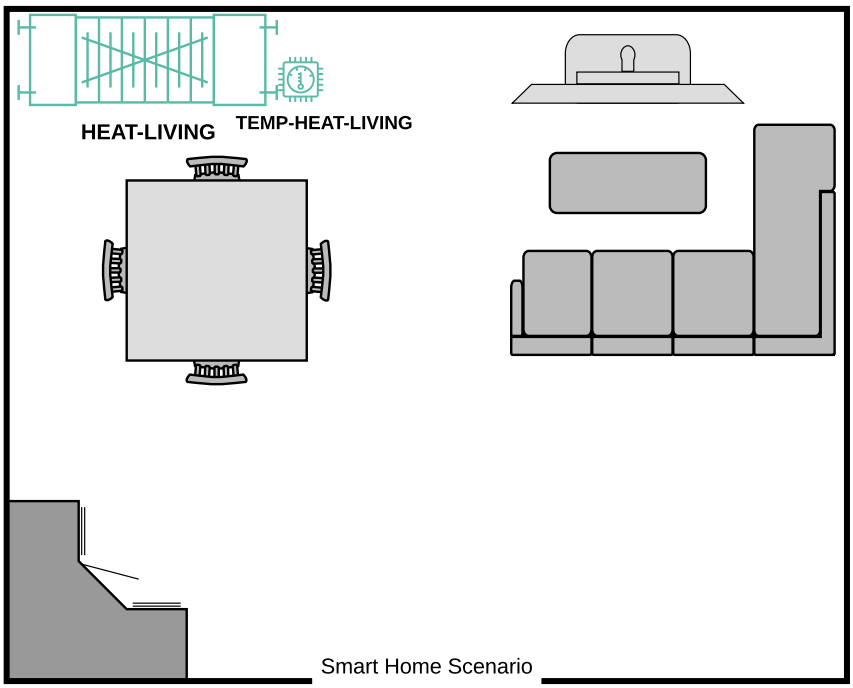

Figure 3.2: Initial application deployment with a T-Res resource in the heating device.

```
from tres import *

t = getInput()
if t < LO:
setOutput("on")
if t > HI:
setOutput("off")
```

As referred in (Alessandrelli et al., 2013), the script above has the inconvenience that it is sending commands to the actuator every time it receives an input. This is not needed most of the times, since only changes to the state of the actuator need to be applied to the output. This can be easily fixed using the task *state*. This is shown in the following script that includes the definition of the *state* class with a variable *prev* that holds the last command sent to the actuator.

```
from tres import *
class state:
    def __init__(self):
        self.prev = "off"

t = getInput()
s = getState(state)
if (t < LO) and s.prev=="off":
    setOutput("on")
    s.prev = "on"
if (t > HI) and (s.prev=="on"):
    setOutput("off")
    s.prev = "off"
saveState(s)
```

Now consider that a new sensor is added to the scenario, namely NEW-TEMP-LIVING as shown in Figure 3.3, with address *aaaa::3*. In this case, we want to adapt the application so that it now reads from this sensor and not the sensor local to the actuator as before.

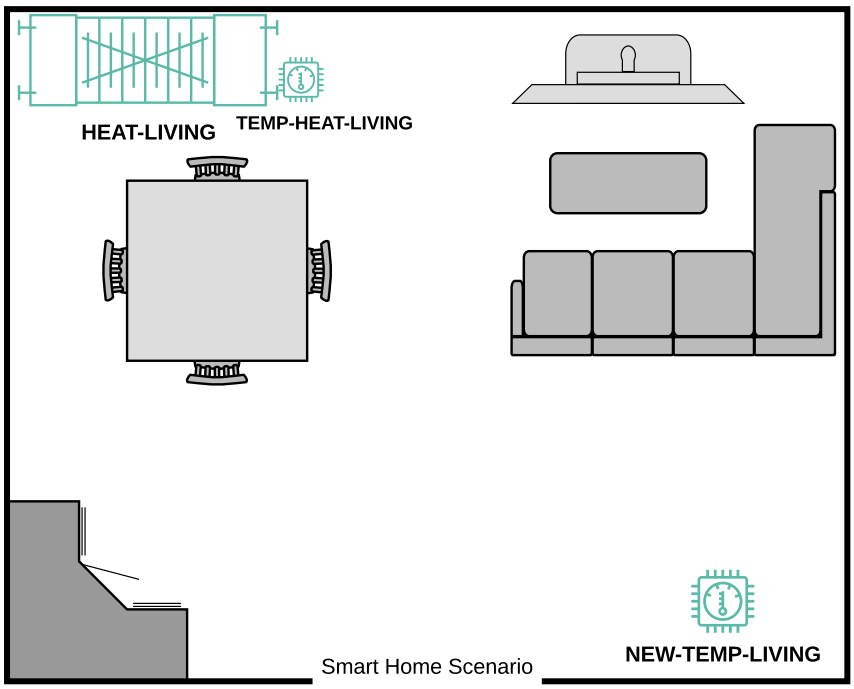

Figure 3.3: Additional external sensor resource added.

In order to accomplish this adaptation, there is no need to write a new script for the new temperature source. T-Res allows this reconfiguration by replacing the previous sensor (TEMP-HEAT-LIVING) with the new temperature sensor (NEW-TEMP-LIVING) as input to the processing function of the task TEMP-CONTROL. Thus, all that is needed is to set its */is* sub-resource to

the new sensor device:

```
/is:
    coap://[aaaa::3]/sens/NEW–TEMP–LIVING
```

Finally, consider that the new external sensor is provided with an alarm that can be switched "on" or "off" remotely, designated TEMP-ALARM. To control the alarm we create another T-Res task, namely ALARM-CONTROL. This task should read from all sensors available, in this case TEMP-HEAT-LIVING and NEW-TEMP-LIVING, and set the alarm if all sensors are above a predefined threshold TMAX. Likewise, if all sensors are below TMAX, then the alarm should be reset. We deploy this task in the external sensor by issuing a PUT command to the following URI:

```
coap://[aaaa::3]/tasks/ALARM–CONTROL
```

Then we set the sub-resources of the task as follows:

```
/is:
    coap://[aaaa::1]/sens/TEMP–HEAT–LIVING "T_LOC"
    coap://[aaaa::3]/sens/NEW–TEMP–LIVING "T_REM"
/od:
    coap://[aaaa::3]/act/HEAT–LIVING
```

Note the tags specified after the URI of the sensors, which are used as alias of each sensor, to allow distinguishing the inputs received in the processing function.

The processing function */pf* can then be set to the byte code of the following Python script. Note the use of the *getInputTag()* function to identify the input that triggered the task function execution. Here we use a state variable to hold the last input from the sensor that was received in the previous function invocation. This way, at each invocation the function receives input from one sensor and reads the last value of the other sensor stored in the state, allowing a joint comparison with the predefined constant threshold. Nevertheless, we also use the state variable to store the previous state of the alarm, whether "on" or "off", so that commands to set or reset are sent only on the state transitions.

```
from tres import *
class state:
    def __init__(self):
        self.loc = 0
        self.rem = 0
        self.prev = "off"

s = getState(state)
tag = getInputTag()
if tag == "T_LOC":
    s.loc = getInput()
elif tag == "T_REM":
    s.rem = getInput()
if (s.loc > TMAX) and (s.rem > TMAX) and (s.prev=="off"):
    setOutput("on")
    s.prev="on"
elif (s.loc < TMAX) and (s.rem < TMAX) and (s.prev=="on"):
    setOutput("off")
    s.prev="off"
saveState(s)
```

Finally, note that the current state of the heater and alarm are available in the respective */lo* sub-resources (last output). Suppose we wish to develop a new task that logs the state of these devices and its transitions. Such a task could simply have its */is* sub-resource set as follows. If the task was not expected to generate any action, its output destination sub-resource */od* could be left empty.

```
/is:
    coap://[aaaa::1]/tasks/TEMP_CONTROL/lo
    coap://[aaaa::3]/tasks/ALARM_CONTROL/lo
/od:
```

3.2 Extending T-Res with Mobility

To understand the limitation of T-Res with respect to mobility and context changes, we resort to another example from (Alessandrelli et al., 2013), similar to the one we examined before. Consider again an application that controls the temperature of a room, this time using one heater and two external nodes, with addresses *aaaa::1*, *aaaa::2*, *aaaa::3*, respectively. We define one T-Res task, namely *avgtemp*, that computes the average temperature from the sensors available (two in this case) and actuates on the heater. This task is deployed on the sensor with address *aaaa::2* as shown in Figure 3.4. The figure also shows another T-Res task, namely *control*, deployed on the heater node, which is not relevant for now. More importantly is the sub-resources structure of the T-Res *avgtemp* task. The */is* sub-resource is set to read from the local sensor in *aaaa::2* and from the other sensor in *aaaa::3*, while the */od* sub-resource is set to the heating actuator in *aaaa::1*.

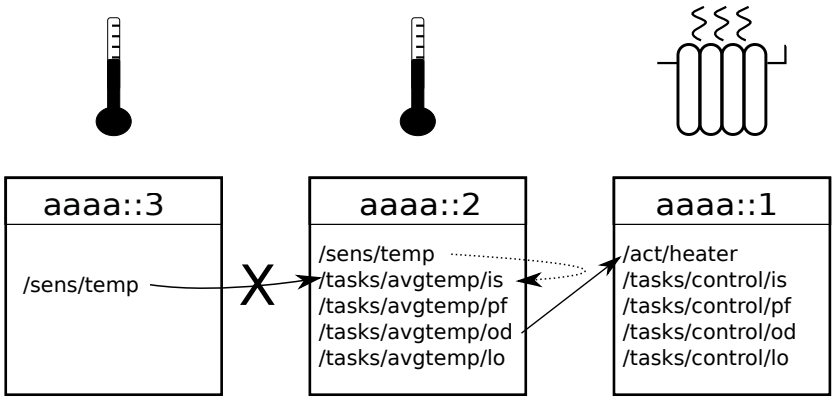

Figure 3.4: T-Res task structure for a room temperature control application.

Now, suppose that the external temperature sensor in *aaaa::3* is part of a mobile node that, at some point, moves out of reach and becomes unavailable. This could also be caused by energy depletion or communication failure. Thus, the *avgtemp* task loses one of its */is* resources and, to continue generating consistent control values, it needs to be reconfigured so that it now reads from the local sensor in *aaaa::2*, only. An obvious reconfiguration is removing the address of the absent sensor from the */is* of *avgtemp*. Second, it is important that the processing function */pf* is designed to handle this situation and continue generating correct commands to the heater. Alternatively, consider that the *control* task could already be prepared to control the heater from a single sensor. Thus, we could suspend *avgtemp* and redirect the input and output of *control* appropriately.

This apparently simple modification, in T-Res, has to be done by the user, providing manual instructions using a CoAP agent. Although there may be many other possible solutions to support an automatic reconfiguration of this type, we opted to extend T-Res with this capability, because of the high level of abstraction it provides, which is in-line with what we need to support context-awareness. We called this extension mT-Res (Gaur, 2015).

Therefore, our goal when developing mT-Res is to detect situations that represent changes of context, like the *silent* sensor failure in the previous example, be it due to mobility, energy, communications or other, and perform the necessary changes in the tasks sub-resource structures, automatically.

This is what we refer to as mobility support and it is the core of the mT-Res extension to T-Res. We define *mobility* as a feature where the resources and the processing function can be moved around to satisfy system context requirements, without needing explicit user inputs. In the previous example, the failure of a temperature node should be detected by the system that should be able to reconfigure the *avgtemp* task */is* sub-resource as also shown in Figure 3.4.

One final note to acknowledge that PyoT (Azzara et al., 2014) is also an evolution of T-Res that includes some support for mobility. It also has a resource monitor that tracks the current status of all resources and can trigger reconfiguration events. However, it had in mind multiple application support and distributed sensor data storage, which is a different direction than the context-awareness in heterogeneous networks that we pursue. Therefore, we decided to extend

T-Res while keeping its main features, just by adding two architectural elements as we will see further on, in mT-Res. Then, we left the extension for context-awareness to be developed on top of mT-Res.

3.2.1 mT-Res Architecture

To achieve the goals referred above, dynamic management elements must be added to the system, capable of detecting context changes and carry out the corresponding reconfigurations.

To achieve this goal we opted for an architecture with two centralized management elements, namely the *Resource Administrator* and the *Application Manager*, as outlined in Figure 3.5. The *Resource Administrator* deploys the code to host devices, assigns the input and output devices and keeps track of any changes in the system. The *Application Manager* allows defining applications and their requirements in terms of resources needed, both for input and output as well as for processing. All other features of T-Res are kept unchanged.

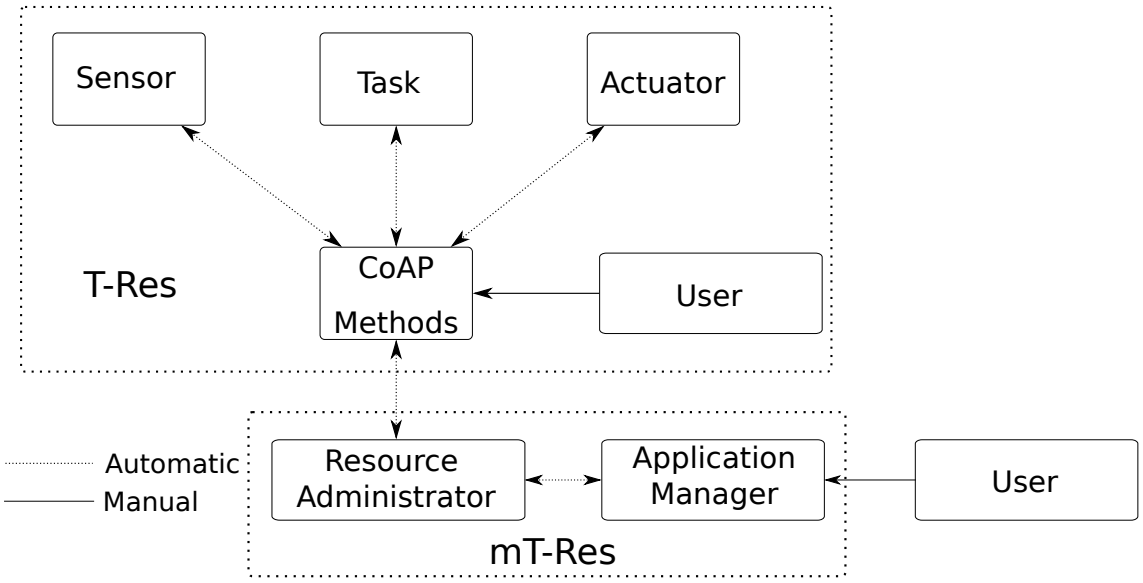

Figure 3.5: mT-Res, a simple middleware that extends T-Res with mobility support.

3.2.2 The Resource Administrator

This component is enabled via Python scripts, which provide automated CoAP operations (such as PULL, PUSH, GET, and using OBSERVE), in form of Python functions. Once applications are deployed and execution starts, the *Resource Administrator* updates the status of all active resources regularly. At any point in time, if any deployment operation returns with an error, the framework will once again execute the process to allocate resources. However, this time it will use another available resource and not the one corresponding to the error received earlier.

3.2.3 The Application Manager

This component uses a Django-based web framework to provide the so-called application form, i.e., an application form where the user can provide the task and its parameters. The *Application Manager* also reacts online to events generated by the *Resource Administrator* to configure all applications whenever the context changes. The application form contains four fields, with some resemblance to the T-Res sub-resources structure. These are: input, output, host and code. In the code field, the user can provide the task processing function code, the same as in T-Res. However, in the Input/Output/Host fields the user is asked to do an abstract selection from available resources, by selecting the type of resource the user wants to use for input or output or to host the code. The user does not need to provide URI addresses of specific resources. However, there can be more complex scenarios where more details are required from the user, such as spatial information, time bounds, etc.

3.2.4 Early Validation

We tested our implementation (Gaur, 2015) on the Cooja simulator in the Contiki Operating System. Cooja provides emulated motes based on the MSP430 microcontroller such as WiSMotes. The simulation in Cooja is cycle accurate for each device and also bit-level accurate for the radio transceivers of each device. This allows to have the same behavior in the simulator as on the actual hardware. IPv6 and CoAP support are provided by Contiki itself. Our implementation also provides support for the CoAP operations in Python with the help of *txthings* (Wasilak, 2015).

For our experiments, first we consider the same example as provided by T-Res. This simple example has four WiSMotes as shown in Figure 3.6. The functions of each mote are as follows: mote 1 is a border router; mote 2 is a host sensor mote; mote 3 is an input sensor mote; and mote 4 is an output actuator mote. Both sensor motes 2 and 3 can measure the same physical parameter, temperature in this case. The host sensor mote 2 takes input from the sensor mote 3, performs some computations with it and provides output to the actuator mote 4.

In T-Res, these three devices have to be connected issuing CoAP PUT requests. The compiled code of the task is also deployed using another PUT request to the URI address of host mote 2. To complete the deployment, a POST request to host mote 2 is required. In T-Res the user is required to issue all these CoAP requests via the Copper CoAP (Kovatsch, 2011) user agent for Firefox. In mT-Res, the user can provide the same code using the application form provided by the *Application Manager*. The *Resource Administrator* takes care of all CoAP operations.

In this example we take a look at a change in context due to energy failure. We demonstrate the actions of mT-Res on failure of two motes, both input (Figure 3.6-a) and host (Figure 3.6-b), respectively.

First, let us assume that after some time of operation, input sensor mote 3 fails due to energy depletion. mT-Res will automatically reinitialize the deployment by replacing mote 3 with mote 2, which possesses a sensor similar to that in mote 3. The *Resource Administrator* performs a PUT

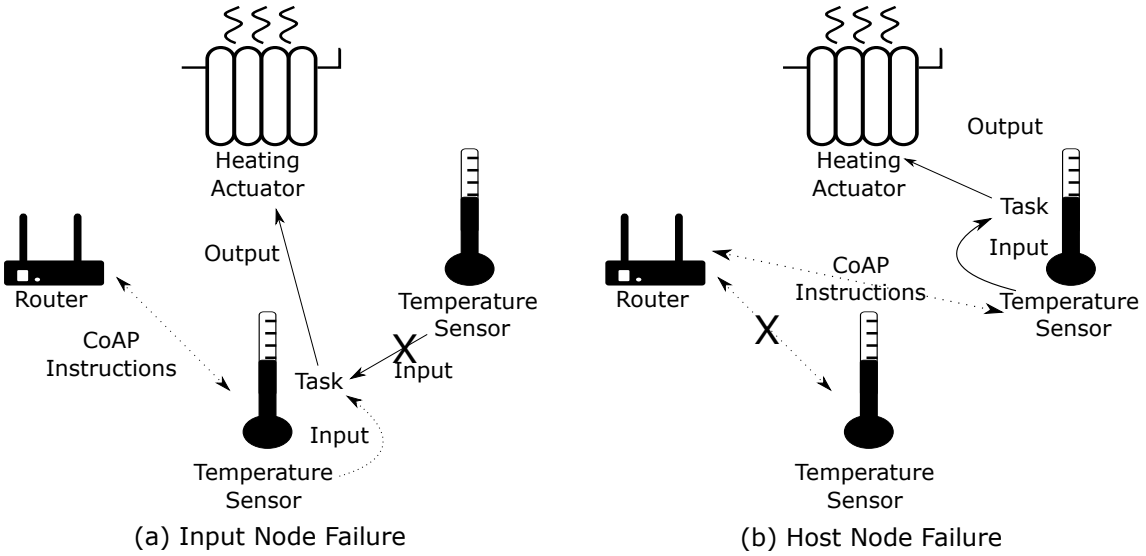

Figure 3.6: Four motes with a simple T-Res application

request to update the task input source as mote 2. After this, normal execution of the application resumes.

In the second case, it is the host sensor mote 2 that fails, instead, and not the input sensor mote 3. In this case, mT-Res reinitializes the deployment replacing mote 2 with mote 3 as host mote for the task, and assigning itself, mote 3, as the input source, too. Once again, this is done by the *Resource Administrator* that performs two PUT requests for both code and input, respectively.

3.3 Summary

In this chapter, we have discussed an existing macroprogramming solution to support adaptation in WSN that also provides abstraction, namely T-Res. We have demonstrated T-Res using illustrative examples. In addition, we have contributed a new feature to T-Res to support application mobility with automatic reconfiguration, which we called mT-Res, hence removing the reconfiguration responsibility from the user. mT-Res is a cornerstone for our main contribution, namely Context-Aware Programming, and was integrated in it. Thus, we opted for excluding the detailed description of its mechanisms from this chapter and integrate it in the following chapter, consistently with the description of CAP. This work demonstrates the need of programming solutions for context-aware sensor networks, particularly explicit context management, which is discussed in next chapter.

Chapter 4

Context-Aware Programming (CAP)

This chapter brings focus back to the concept of context-awareness in WSN. Specifically, we propose a new middleware for WSN that provides an adequate programming framework to support context-awareness, which we name Context-Aware Programming, or simply CAP. This middleware builds on top of mT-Res, the T-Res extension to support mobility (Chapter 3). To provide context-awareness, CAP goes one step beyond and provides a comprehensive solution resorting to an architecture that manages automatically not only applications and resources but also contexts. This architecture allows a user to program and interact with mobile sensing systems in a simplified way, leveraging context-awareness capabilities efficiently.

CAP is a core contribution of our thesis as briefly introduced in Section 1.7, and we believe it is a keystone to build context-aware WSN, which we referred to as CASE.

This chapter examines CAP essential design features. The following section revisits the concept of programming for context-awareness and its relevance. Section 4.2 discusses the CAP solution and its essential features to enable context-awareness. Section 4.3 presents CAP middleware architecture, its components and implementation. Section 4.4 discusses an experimental validation of CAP, its results and the benefits of CAP features. Section 4.5 provides a summary of the chapter.

4.1 The Core Concepts of CAP

The main driver of CAP is to support context-aware WSN by providing the user with appropriate tools to write and deploy context-dependent applications. As we saw before, there have been many efforts to provide better programming support for WSN users (Mottola and Picco, 2011). Some proposals, e.g., by Alessandrelli et al. (2013), already provide support for adaptations, letting the user express the desired goals without requiring knowledge about specific resources. Note that a programming framework that provides independence of low-level details of resources and allows for the mobility of applications over multiple resources is, in principle, capable of supporting context-awareness.

However, to develop context-awareness programs, it is important to refine the understanding of *context* itself. This can be achieved by defining the *context* for different scenarios, e.g., related to security, energy, communication or user behavior. For any system, these context scenarios can change with time or multiple scenarios can exist at the same time. To adapt to new context scenarios the system must take actions. These actions can be the deployment of new applications across the network or re-configuration of existing applications for different resources.

In CAP we propose to take a declarative approach to program user applications. In this approach, a user is able to write self-contained blocks of code that can process a predefined type of input and provide a certain type of output. We also propose three essential features for CAP, as follows:

- *Abstraction* allows the user to write code without worrying about low-level details.
- *Modularity* allows the user to write code that is reusable in modules to provide specific functions.
- *Mobility* allows the user to write code which can be moved around the network across any suitable device.

To understand these features in a better way, let us take a look at the example of an HVAC system operation shown in Figure 4.1. *Application A* takes input from the two temperature sensors and provides the average temperature as output. *Application B* checks for user presence using the input from a motion sensor and the average temperature from *Application A* and, if conditions are satisfied, actuation for appropriate heating or cooling takes place. *Application A* is deployed on one of the temperature sensors and *Application B* is deployed on the HVAC actuation device. There are two motion sensors available which can be used by *Application B*.

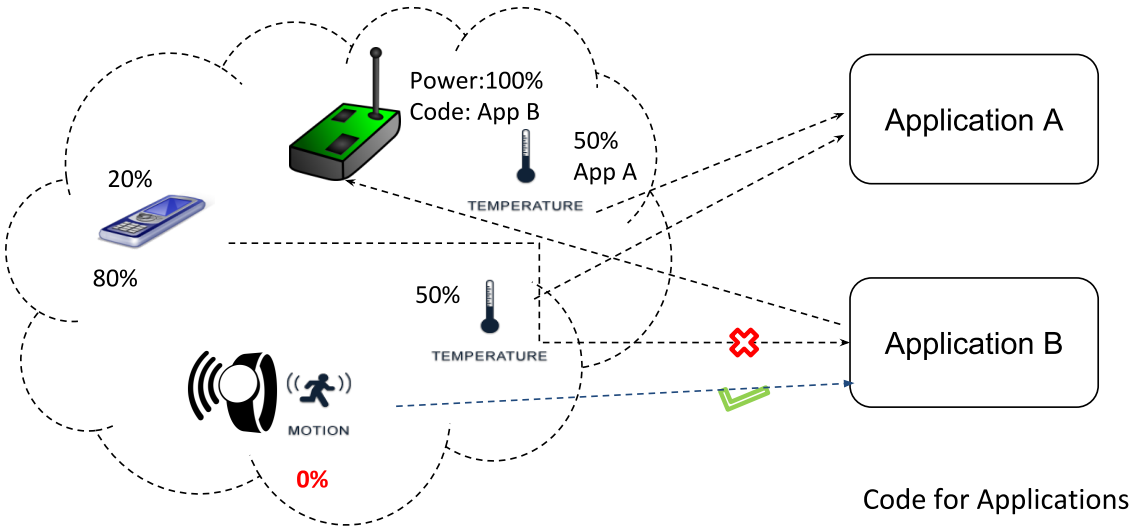

Figure 4.1: Features for Context-Aware Programming

Assume the motion sensor provided by the wearable device becomes unavailable in the network, either due to low energy levels or connectivity. *Application B* can still use another motion sensor available via the smartphone.

To accommodate this change, the code for *Application B* must not be bound to the hardware address of one motion sensor. The user should be able to write the required input without specifying each device to be used, but just the input data required. This is where the *Abstraction* feature of CAP is required. Similarly, the complete HVAC application to detect user presence and calculate an average temperature could be written all together, which would be the usual way using traditional WSN programming tools such as Contiki. However, dividing this objective into two self-contained applications provides the ability to change the input sensors for one application of the whole objective without interrupting another application. *Modularity* helps in providing such ease of access for the system. In another scenario, one of the temperature sensors hosting *Application A* may become unavailable in the network. In that case, *Mobility* allows the system to deploy *Application A* on the other device such as the smartphone to get the temperature as an input.

Abstraction, *Modularity*, and *Mobility*, altogether enable the user to write applications for different contexts. When a context change occurs, the system can make appropriate adaptations without any manual configurations by the user.

4.2 Implementing CAP

In order to implement CAP, we take inspiration from some of the existing programming frameworks for WSN and add the necessary adaptations to accommodate context-awareness in mobile sensing systems. Two such frameworks stand out for their influence on the way we implemented CAP. One, already explained in the previous chapter, is T-Res (Alessandrelli et al., 2013). T-Res is able to keep input and output parameters separate from the application code. This is an extremely useful feature that we also take advantage of, providing a similar abstraction between the code written by the user and devices on which the code executes. The other work that inspired the implementation of CAP is PyFuns (Bocchino et al., 2015), which also provides similar abstraction.

However, differently from those frameworks, CAP strives to provide the three features needed for context-awareness altogether with full autonomy, namely *Abstraction*, *Modularity* and *Mobility*.

To achieve this purpose, CAP uses an architecture composed of three management components as shown in Figure 4.2. These are the *Application Manager*, the *Resource Administrator* and the *Context Manager*.

The *Application Manager* collects code from the user for each application and also helps in keeping track of active applications. The *Resource Administrator* keeps track of the current system composition concerning devices, resources and their status, and detects any changes in the system. It also deploys the applications on host devices, assigning the input and output devices, and keeps track of the current applications deployed in the system. Finally, the *Context Manager* collects

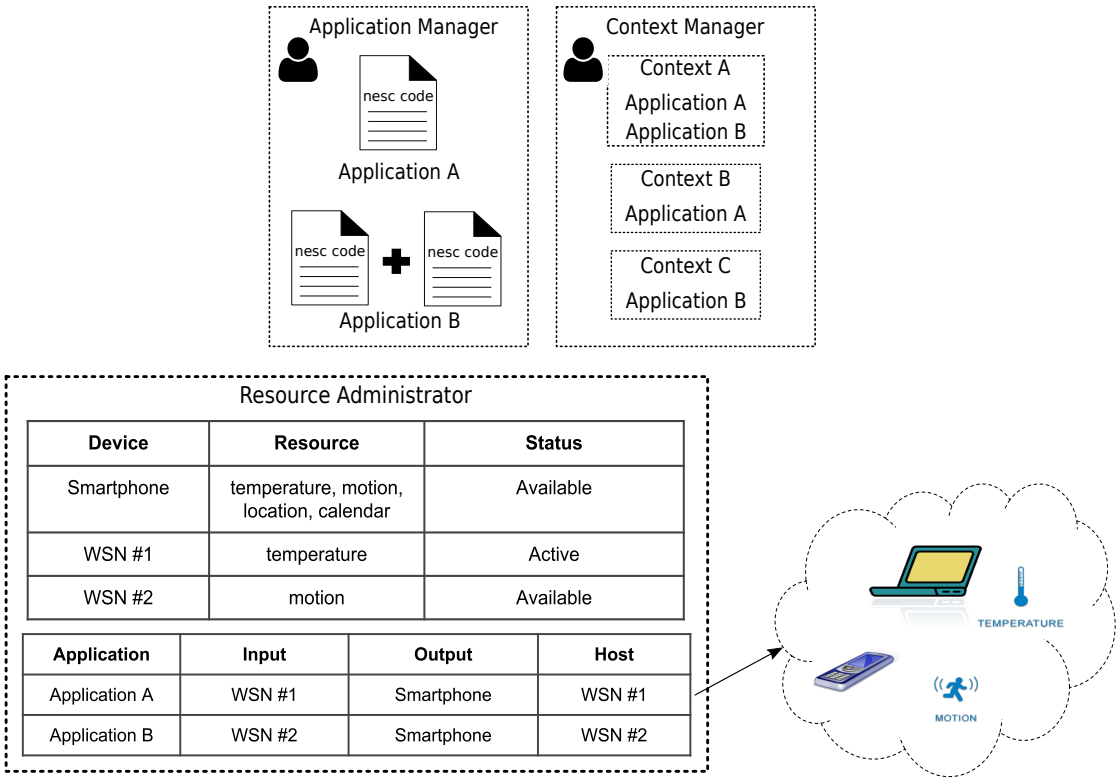

Figure 4.2: Components of Context-Aware Programming

information on the contexts defined in the system and compiles a list of applications associated with each context.

The CAP components are implemented using Python and Django programming languages. The user applications are coded with *nesC*, similarly to T-Res and many other popular programming frameworks. Due to its popularity, nesC allows familiar users to use CAP without learning another new programming language or syntax.

4.2.1 Application Manager

The *Application Manager* utilizes a web form similar to that used in (Azzara et al., 2014) to collect code from the user for each task, as shown in Figure 4.3. This form is built using Django and has a backend in Python to refresh the list of applications. This form has four input fields, namely *input*, *output*, *host* and *code*:

- *Input* is the type of input required by the application. The user can perform an abstract selection from all the available resources using a drop-down list. There is no need to provide an address or details of a particular device, just the type of resource is required. For example, if there are two temperature sensors, three motion sensors, and one pressure sensor available for the user, the web form will show the user a drop-down menu with these three choices: *temperature*, *motion* and *pressure*.

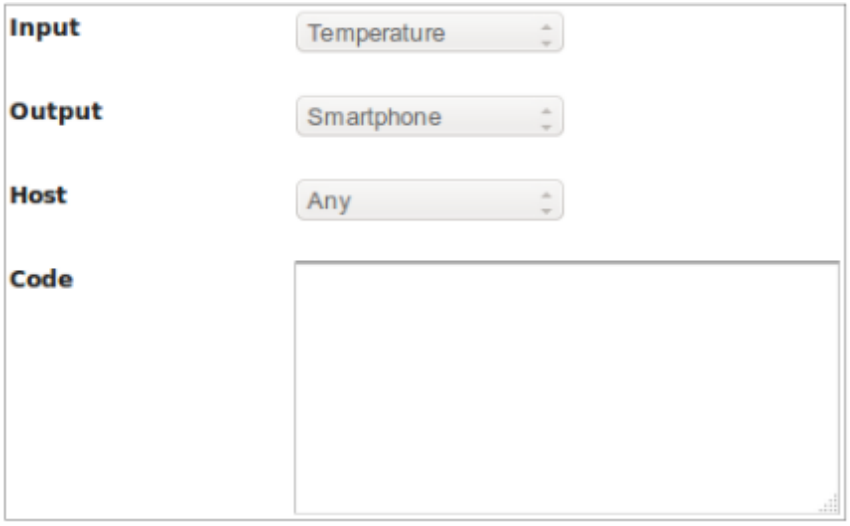

Figure 4.3: Programming web form similar to that used in (Azzara et al., 2014) provided by the Application Manager

- *Output* is the output destination for the application. Similarly to the *Input* field it allows selecting the destination device of this application output using a drop-down menu.
- *Code* is the nesC code written in an abstraction similar to T-Res.
- *Host* is the devices where the *Code* can be executed. Similarly to the *Input* and *Output* fields, this is also selected from a drop-down menu that also provides keywords that represent groups of devices, e.g., *any*.

These four fields are enough to specify a resource-independent application. Sometimes more information about the devices maybe required such as time bounds, spatial limits, etc.

4.2.2 Resource Administrator

The *Resource Administrator* is also enabled via Python scripts (*resource_track.py* and *resource_table.py*). These scripts also provide automated CoAP operations, namely PULL, PUSH, GET, and OBSERVE. CAP utilizes an open source library, namely *txthings* (Wasilak, 2015), to make use of CoAP operations in Python.

As shown in Figure 4.2, the *Resource Administrator* creates and regularly updates two tables implemented on a *Current Resource Database*, the first of which contains the status of all the devices available in the system and resources provided by those devices[1].

The *available* status indicates that the node is available for use by any application and *active* status indicates that the node is already in use by a certain application. More status types can be easily defined if convenient to provide more information on resources. For example, *inactive*

[1]From here on *resource* defines a *sensing resource* in a physical device, e.g. a smartphone has multiple *resources* such as temperature, location, motion, etc.

status could be used to indicate that a node is no longer in the network, e.g., due to energy depletion, where as *unavailable* status could indicate that a node is temporarily incapable of running applications, but may become available in the future.

After the user submits an application via the provided web form using the *Application Manager*, the *Resource Administrator* takes decisions on which resources to use to host the code, to take input from and provide output to. It creates a dictionary with the list of acceptable resources for each decision, using the first table. If any of these lists are not created or are empty, the framework notifies the user that the desired resources are not available. Once the decision is finalized, the *Resource Administrator* creates a second table with these decisions, as shown in Figure 4.2.

Every time there is an update, the *Resource Administrator* will execute the CoAP operations needed to deploy the code on the selected host device and assign the input and output devices using URI addresses. For this it will choose one of the options from each list with adequate criteria and keep the selection saved for future references. The Resource Administrator can also track changes in the code available to be deployed and make decisions based on the changes in the availability of the code. Such functionality can allow CAP to make decisions based on new code added by different users.

This functionality is implemented through additional flags that are automatically added by CAP to the nesC code. These flags are inserted appropriately in the code so that, whenever there is a context change, the code can be recompiled for and redeployed on another appropriate resource. Most importantly, these actions are performed autonomously by the CAP middleware to assure continuous execution of the applications, without intervention from the user.

The CoAP operations such as PUT and POST are executed to deploy the selections and run the application task, respectively. The GET operation is executed to make sure that selections are made correctly, as it returns the status of each resource. This can be seen in Listing 4.1, where *resource* is the dictionary with lists of all acceptable resources. The *host*, *input* and *output* are selected from this dictionary. The functions *assign* and *post* refer to the CoAP requests for PUT and POST respectively. The status of each assigned resource is changed from *available* to *active*.

```
hostdevice = resource[host_type][available]
inputdevice = resource[input_type][available]
outputdevice = resource[output_type][available]
assign(inputuri, inputdevice)
assign(outputuri, outputdevice)
assign(codeuri, open(code).read())
post(taskuri, "Start")
```

Listing 4.1: CoAP operations to assign resources.

As the application is deployed and execution starts, the *Resource Administrator* updates the status of all active resources regularly. This is done by performing GET operations via the Python function *checkresource*, as shown in Listing 4.2. At any point, if any operation returns with an error, the framework will once again execute the process to allocate resources. However, this time it will use another available resource for the corresponding error received earlier.

```
host_status=checkresource(hosturi, hostdevice)
input_status=checkresource(inputuri, inputdevice)
output_status=checkresource(outputuri, outputdevice)
if host_status!="Active" or input_status!="Active"
        or output_status!="Active":
        application.restart()
```

Listing 4.2: Check status of resources.

As mentioned earlier, CAP utilizes multiple Python scripts to iterate through the code provided by the user. These scripts add additional flags to the nesC code and compile it to generate the binary file to be deployed on the devices using CoAP operations. These flags are inserted appropriately in the code so that, whenever there is a context change, the code can be recompiled for, and redeployed on, another appropriate resource. Most importantly, these actions are performed autonomously by the CAP middleware to assure continuous execution of the applications, without intervention from the user. CAP utilizes an open source library, namely *txthings* (Wasilak, 2015), to make use of CoAP operations in Python.

4.2.3 Context Manager

The *Context Manager* is enabled by a set of Python scripts named *context-dict*, *context-input* and *context-track*. These scripts altogether allow defining, adding and removing contexts as well as keeping track of every context and the applications associated with each one. For an application to be associated with a context, it must have been already created using the Application Manager.

A context is created using the *dictionary* data type in Python. Each context has four keys:

- *id* is used for identification using an integer value.
- *group* denotes the ranking of individual context, which is used to resolve conflicts between multiple contexts.
- *applications* contains a list of all associated applications.
- *triggers* is a list of items which can affect the context, such as output of devices, application, or user inputs. A separate list is created with the conditions for each of these triggers.

Using these keys, the scripts are able to detect and inform other components about a change in context and trigger the associated adaptations.

The definition of a context is not limited by the keys defined in the early stages of the implementation. As the part of system design, it is possible to allow user to integrate complexities in context manager. However, each context must always be associated to one or more applications. Triggers for a context can be customized as long as it is also able to activate triggers using the resource manager.

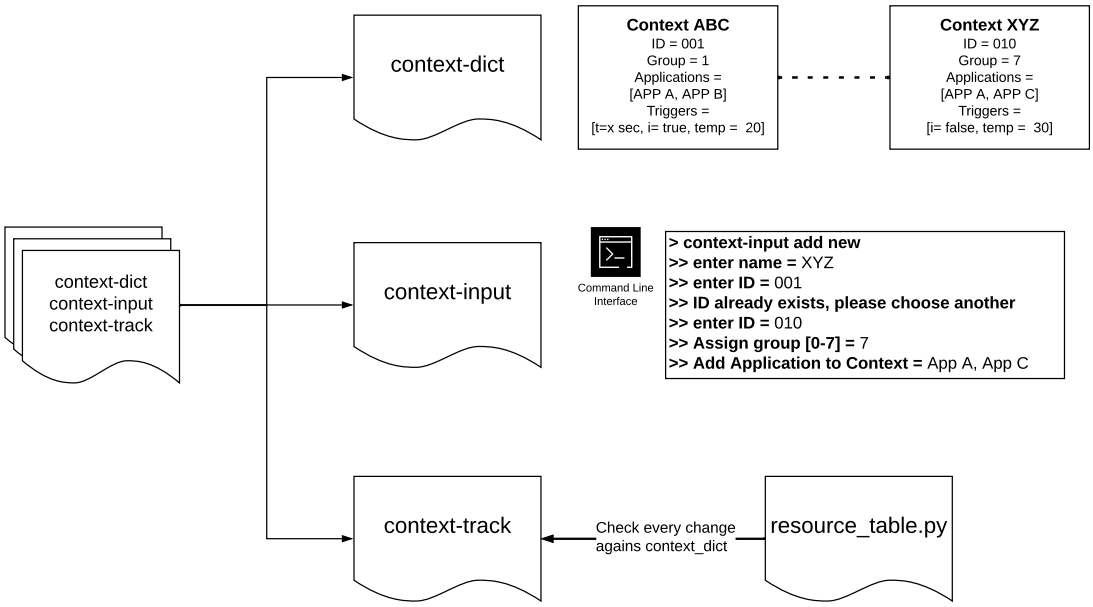

Figure 4.4: Components for Context Manager

4.2.4 Prototype Architecture

Figure 4.5 shows the implementation architecture used in a proof-of-concept prototype. Particularly, it shows how different scripts of CAP work together to provide the functionality of its three components as described before.

This architecture organizes the whole functionality in four blocks. One block includes the features managed directly by the user, another block contains CAP's operational information, a third block has the actual CAP management features and the last block holds the logging and redeployment features. The four blocks are described as follows:

- **Managed by the User**: This block contains the actions that are managed by a user directly. These actions allow a user to interact with different components of CAP according to its model (Figure 4.2). For the Application Manager, a user needs to submit the application source code through Django web forms. For the Context Manager, a user can provide different contexts by using command line inputs through the Python script named *context_input.py*. Lastly, to provide information regarding all the possible resources in the system, a user would have to use a form-based approach as done in the Application Manager. However, for the sake of simplification, this is currently done manually via a Python-based *dictionary*.

- **External Storage**: This block includes the storage needed to hold all the information referred above and provided by the user. It contains different data objects, possibly implemented separately, that must be accessible to both user and the CAP running on an embedded device in the system. Currently, this storage is implemented on a local disk in a

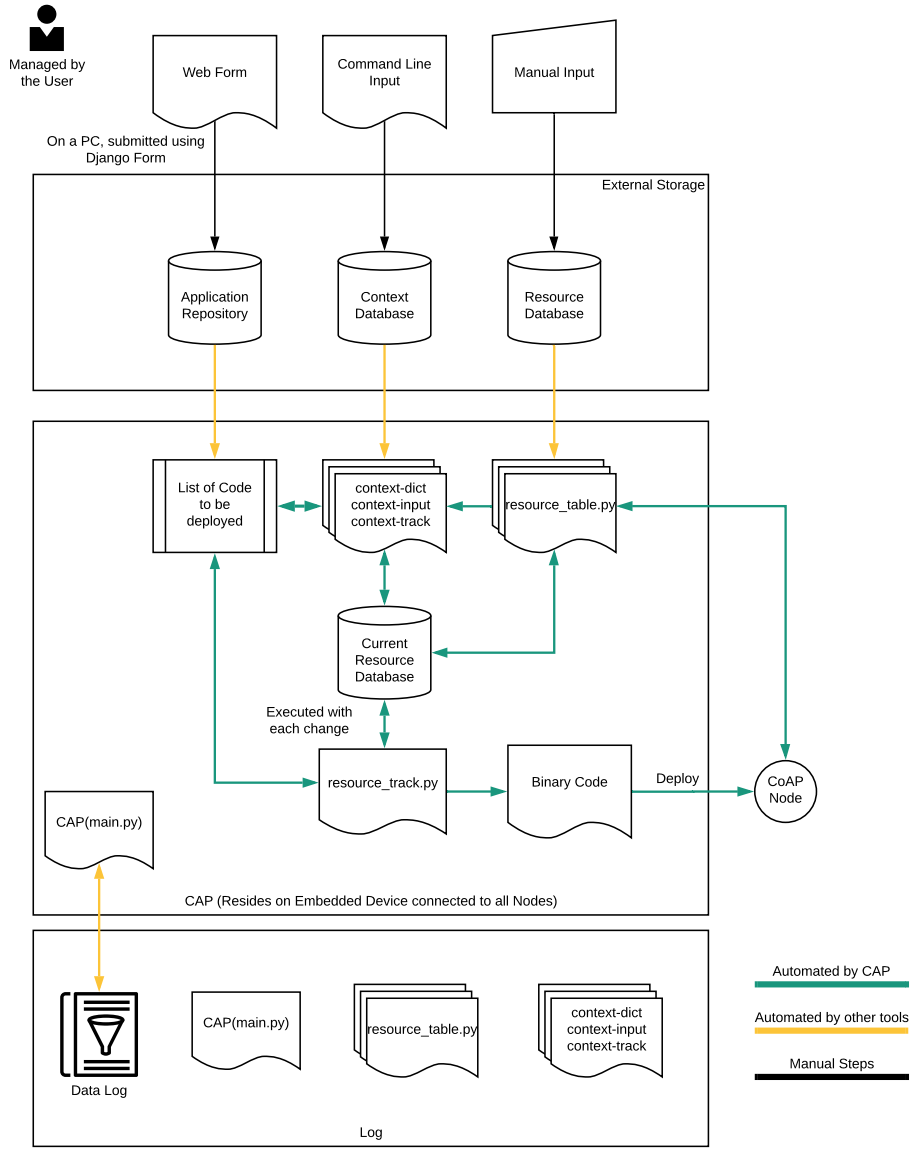

Figure 4.5: Architecture for Context Aware Framework

computer connected to same network as the CAP embedded device, but it can be a completely separate storage accessible through the same network. The information should be readily available to CAP including upon any changes made by the user.

- **CAP**: The CAP block contains all CAP management scripts that run on an embedded device connected to the WSN. These scripts cooperate to implement the functionality of the CAP components. The Context Manager uses a group of several scripts representing the different contexts. These scripts interact with both the Resource Administrator and Application Manager. Similarly, different scripts are used to track resources and implement the Resource Administrator component, interacting with the Application Manager and the Context Man-

ager, and maintaining the Current Resource Database. Upon any change in the applications, contexts or resources, the *resource_track.py* script parses the code, generates the binary and executes the corresponding deployment in the involved WSN nodes. A main script, named *main.py*, carries out a supervisory role, resetting the other scripts when needed and logging any data to the logging device.

- **Log**: The *Log* block includes the functionality needed to save a Data Log generated by the *main.py* script in CAP. It is implemented in a different embedded device, but in the same network as the CAP embedded device. Currently, the device running the Log also stores copies of all the CAP scripts which may be needed for redeployment of the CAP management layer on the same or other embedded device.

4.3 An Illustrative Example

This section explains the functionalities of CAP using an illustrative example based on a simple HVAC system. There are four sensor nodes as shown in Figure 4.6. The function of each node is as follows:

- Node 1 acts as a border router node
- Node 2 acts as a host temperature sensor node
- Node 3 acts as a input temperature sensor node
- Node 4 acts as an output heating actuator node

Both sensor nodes 2 and 3 can measure the same physical parameter. The host sensor node 2 takes input from the sensor node 3, submits this values to an internal application function (called *halve*) and provides output to the actuator node 4.

Similarly to T-Res, these three devices, host, input and output, have to be activated using CoAP PUT requests. The compiled code of the application task function (*halve*) is also deployed using another PUT request to the URI path of host node 2. To complete the deployment, a POST request to host node 2 is required. In T-Res the user is required to issue all these CoAP requests via the Copper CoAP (Kovatsch, 2011) user agent for Firefox. In CAP, the user can provide the same code using the application form provided by the *Application Manager*. The *Resource Administrator* takes care of all CoAP operations.

Table 4.1: Status table after application deployment.

Application	Halve Task	Running
Code	halve.c	Sent
Host	coap://[aaaa::200:0:0:2]/tasks/halve	Active
Input Source	coap://[aaaa::200:0:0:3]/sensor	Active
Output Device	coap://[aaaa::200:0:0:4]/actuator	Active

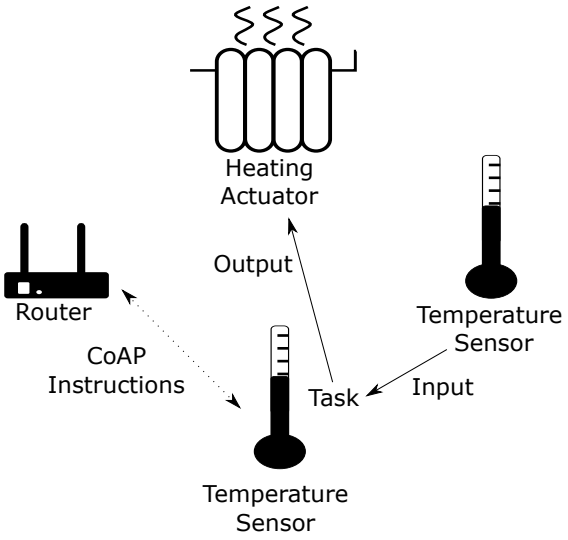

Figure 4.6: A simple HVAC application with four nodes.

As soon as the user submits the application, CAP will return with a deployment success message to the user and build a table with the status of nodes being used for the current deployment (Table 4.1). This table is refreshed regularly via GET requests in the background. A time bound for these requests can also be provided by the user. The user may also force a refresh via options provided by CAP.

4.3.1 Changes in Context

To illustrate the context adaptations supported by CAP, consider the two reconfiguration scenarios shown in Figure 3.6 (Chapter 3) triggered by an energy failure affecting either the sensor node 3 or the host node 2 (Figure 3.6). These reconfigurations are associated to different contexts that correspond, in this case, to different sets of nodes available, namely all nodes or just nodes 1, 2 and 4 or then nodes 1, 3 and 4. All contexts are associated to the referred application.

4.3.1.1 Failure of input node 3

First, let us assume that after some time of operation, input sensor node 3 fails due to battery depletion (Figure 4.7). This causes a switch in context, leading to an update in the system status as shown in Table 4.2.

Table 4.2: Status table upon failure of the sensor node 3.

Application	Halve Task	**Halted**
Code	halve.c	Running
Host	coap://[aaaa::200:0:0:2]/tasks/halve	Active
Input Source	**coap://[aaaa::200:0:0:3]/sensor**	**Inactive**
Output Device	coap://[aaaa::200:0:0:4]/actuator	Active

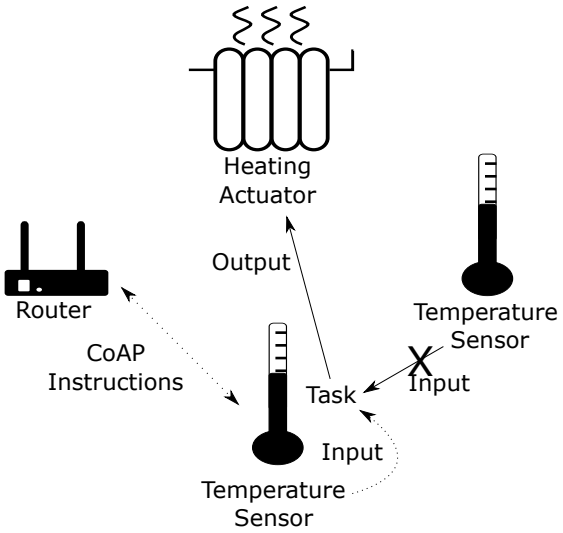

Figure 4.7: Failure of a sensor resource.

Upon the change of context, CAP will automatically reinitialize the deployment of task *halve* by replacing the input from node 3 with the local input from node 2. The *Resource Administrator* performs a PUT request to redefine the application input source as node 2.

After the reconfiguration, normal application execution resumes in a fully user-transparent way. The new system status is represented in Table 4.3.

Table 4.3: Status table upon reconfiguration and redeployment

Application	Halve Task	**Running**
Code	halve.c	Sent
Host	coap://[aaaa::200:0:0:2]/tasks/halve	Active
Input Source	**coap://[aaaa::200:0:0:2]/sensor**	**Active**
Output Device	coap://[aaaa::200:0:0:4]/actuator	Active

The sequence of operations is shown in Figure 4.8. Note that the status change caused by the input node failure is detected by the Resource Administrator. The comment boxes indicate the points in the sequence when the system status is the one shown in the status tables above.

4.3.1.2 Failure of host node 2

In a second case, instead of the failure in the input node 3, we now have a failure of the sensor node 2 that hosts task *halve*, again due to battery depletion. In this case, CAP will reinitialize the deployment by replacing node 2 with node 3 as host node and assigning the same node as the input source as well (Figure 4.9). This reconfiguration is done by the *Resource Administrator* with two PUT requests, for both code and input deployments, respectively. The respective sequence diagram is shown in Figure 4.10.

In both cases of context changes shown above, the user is not required to intervene since CAP is able to detect the context change and adapt accordingly. Moreover, the architectural components

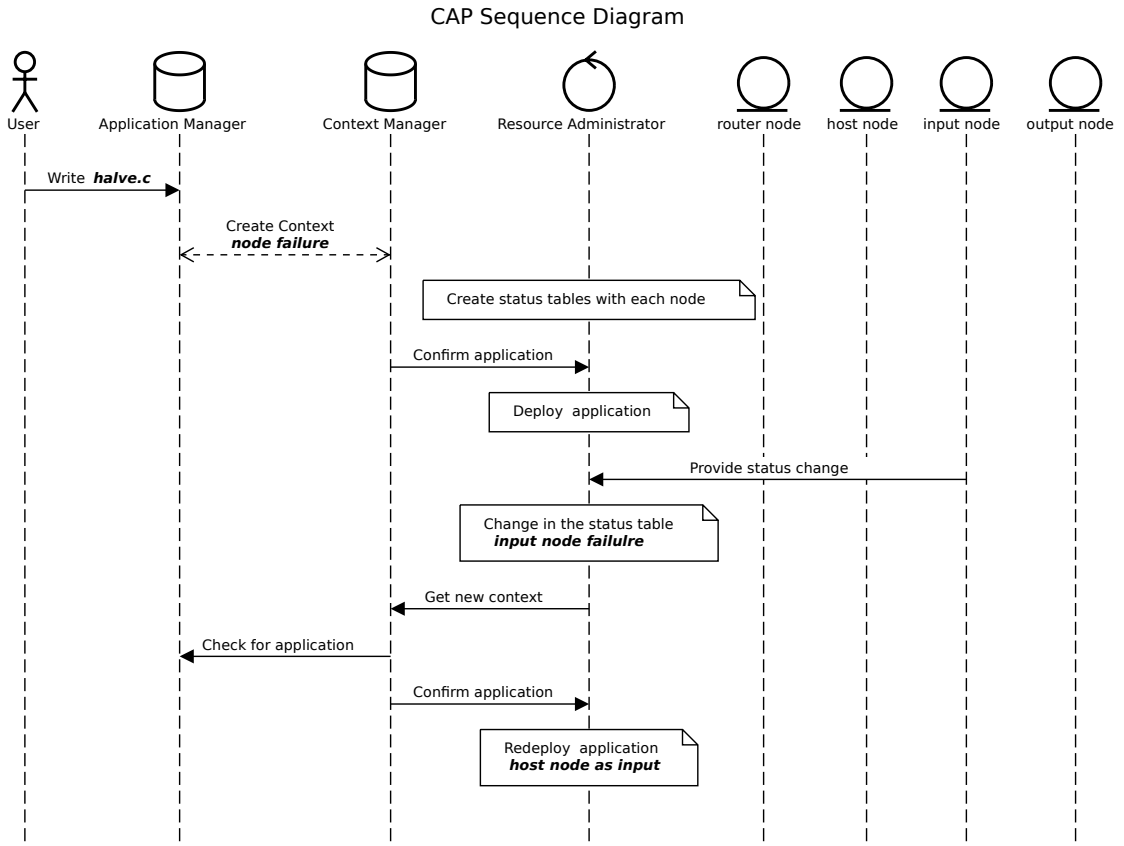

Figure 4.8: Sequence diagram with input node failure.

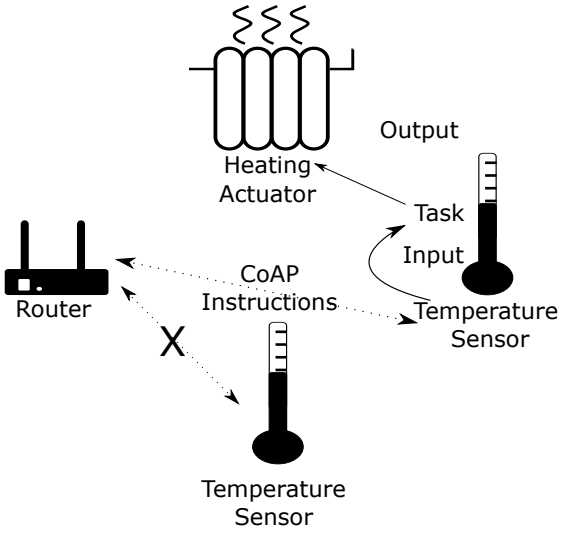

Figure 4.9: Failure of a task host resource.

of CAP necessarily need to run on an adequate node. In the example above, this could hypothetically be the router node 1, or another node that supports the application in the background, just for the purpose of running CAP management components.

Finally, there are a few parameters involved in CAP operation that we did not explicitly refer

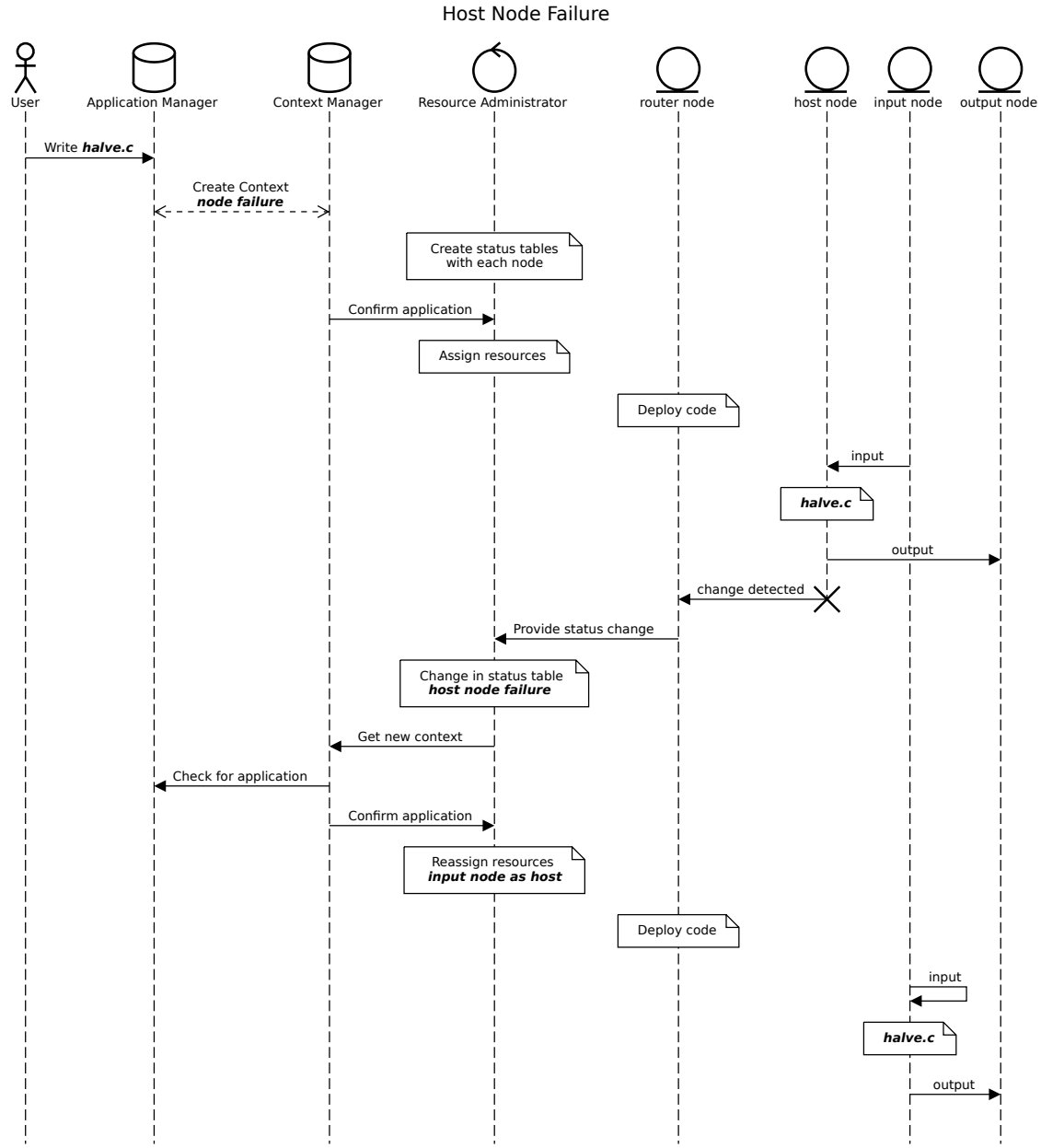

Figure 4.10: Sequence diagram with host node failure.

and which have an impact on the efficiency and reactivity of systems running on CAP middleware. For example the status table can be refreshed upon the occurrence of an event, but it also needs to be refreshed upon a timeout, to detect omission events, such as those caused by silent failures (like the energy failures referred before). The length of such timeout imposes a trade-off between reactivity to failures and overhead. We envisage that this timeout can either be pre-defined by the user or adapted dynamically by an enhanced resource management algorithm.

4.3.2 General Operations Flow in CAP

Generalizing the flow of operation in CAP, between WSN devices and the CAP management node, we can represent in a simplified manner both normal operations and interruptions with reconfigurations occurring in various stages of the WSN operation. This helps understanding how CAP adapts and executes the changes required for each context.

Initially, we have a list of applications ready to be executed on our run-time system, developed with the help of the Applications Manager. Then, the user provides the details of the contexts with help of the Context Manager. With this input, the applications are assigned to different contexts and possibly grouped under one or more contexts.

Simultaneously, the Resource Administrator builds a system status table with all the devices available. When applications are ready to be deployed, using the system status table the assignment of executable code and its deployment begins (distribution). Once applications are delivered their execution on specific devices can start. This is shown in Figure 4.11.

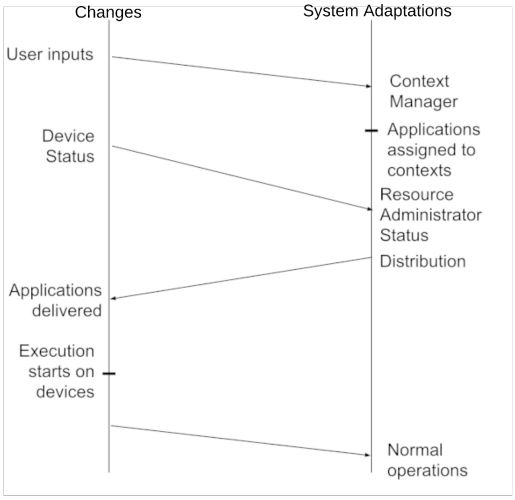

Figure 4.11: Initial applications deployment with CAP.

Under normal operations, the Resource Administrator monitors continuously the devices in the system, thus detecting node failures or additions of new nodes. This is shown in Figure 4.12 (left). The system status is updated and a new context is activated, potentially leading to a new deployment (adapted distribution). Once the redeployment is done, normal operation resumes.

It may also happen that a given application is suspended (interrupted) due to lack of adequate resources. In this case, when the Resource Administrator detects new resources in the system, a new context is activated, which may already allow the suspended application to resume operation. Thus, a new redeployment is done (new distribution) and the execution is restarted. This situation is shown in Figure 4.12 (right).

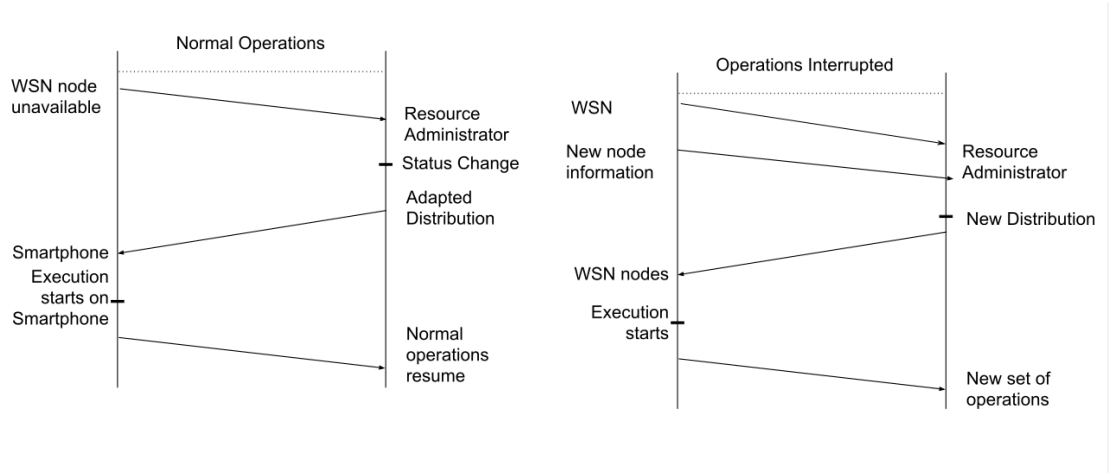

Figure 4.12: Adaptation operations in CAP.

4.4 Experimental Results

CAP was developed in an incremental fashion, building on technologies that already exist for WSN, such as T-Res, and re-purpose them for CAP. In this section we discuss several experiments that were carried out to validate CAP, namely its dynamic management mechanisms. We refer to some preliminary experiments which purpose was to debug CAP and validate specific features and mechanisms. Then, we discuss resilience experiments carried out in a remote IoT facility where it was possible to assess the autonomous adaptations of CAP for an extended period of time.

4.4.1 Preliminary Experiments

During the various implementation stages of CAP, most of the development effort was done to run applications using Contiki-OS on TelosB and MicaZ sensor motes. We used Cooja simulations for Sky motes to advance initial development and debugging of code for the CAP. The Sky mote is the basic mote available in Cooja that can be used within a WSN and does not require significant configuration efforts for simulations. Once a proof-of-concept simulation version of CAP was ready, we deployed simple applications using MicaZ motes and TelosB motes.

These experiments were essentially carried out on an IEEE 802.15.4 (Zigbee) network using the existing MAC layer and running on Contiki. As most of the CAP-related developments work at the application layer, there was no need to make modifications to lower layers.

As referred above, these experiments aimed at debugging CAP and its functionality but not at performance assessment. For this reason, the necessary adaptations were forced manually. For example, removal of nodes was forced by removing energy sources manually from one of the motes engaged in the WSN application and discovery was triggered by powering on a new mote.

Initially, the experiments with the core scripts of CAP were run on a virtual machine on the same computer that was running the Contiki-OS and Cooja for the motes. Once this initial debugging phase was completed successfully, CAP was moved to a Raspberry Pi, that we considered as

providing a more realistic target, better suited for higher autonomy systems.

4.4.2 Resilience Experiments

Once all the preliminary experiments were successfully completed, we looked for a suitable testbed that could support more realistic experiments during sufficiently large periods of operation to assess its resilience and autonomous adaptation. The chosen testbed was the IoT-LAB from the Future Internet Testing Facility and provided by OneLab in France, among multiple test sites. For these experiments we used the sensor nodes provided by testbed site itself, namely IoT-LAB M3 and IoT-LAB A8-M3. These are the most widely available boards in the testbed and designed specifically for that purpose.

The IoT-LAB M3 node, shown in Figure 4.13, is based on a 32 bit CPU with 64kB of RAM, 256kB of ROM, 2.4 GHz radio chip and four sensors, namely light, pressure and temperature, accelerometer and gyroscope). We used this type of nodes as sensor nodes in our experiments, running the applications as a replacement of TelosB or MicaZ motes that we had used in the preliminary experiments.

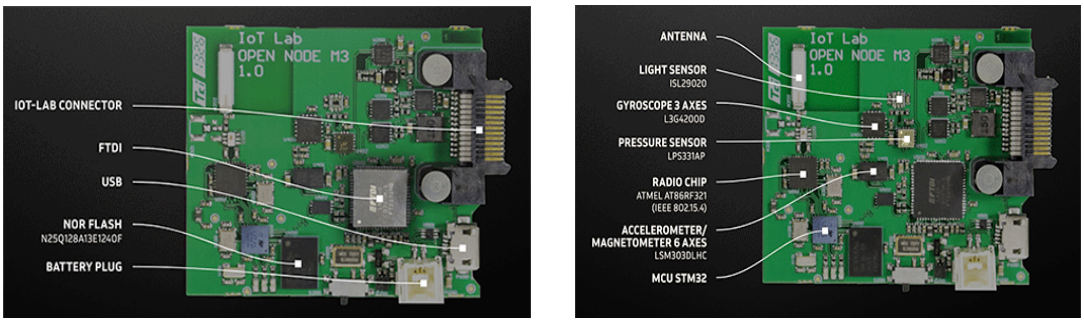

Figure 4.13: IoT-LAB M3 node with 32 bit CPU, 64kB RAM, 256kB ROM and 4 sensors.

The IoT-LAB A8-M3 node, shown in Figure 4.14, is a more advanced board with a high-performance ARM Cortex-A8 microprocessor. It is capable of emulating the behaviour of a smartphone or tablet. Some boards are also equipped with a GPS module. One of these nodes is used to run the core scripts of CAP, as a replacement of the Respberry Pi used in the preliminary experiments. Note that the IoT-Board A8 M3 runs a version of Linux. This simplified the porting of CAP from the Raspberry Pi version, requiring just very small modifications to support new sensors available in the testbed. Though CAP management components are all running on the same node, the node itself may need to be exchanged for another one multiple times since it may deplete its energy source. One interesting feature of the IoT-LAB A8-M3 node is that it includes a clone of the IoT-LAB M3 node attached to itself, hence enabling it to be used as a sensor node, too, if needed.

For the sake of simplicity and without risk of confusion, from here on we use just *M3-x* where $x \in [1,8]$, or simply *M3* node, to refer to the IoT-LAB M3 x node and *A8-y* where $y \in [1,3]$, or simply *A8* node, to refer to the IoT-LAB A8-M3 y node.

Figure 4.14: IoT-LAB A8-M3 with Cortex A8 high-performance CPU and a clone of the IoT-LAB M3 attached.

Concerning the programming of the testbed, it provides a REST API, hence facilitating the porting of the scripts that had already been developed before in the preliminary experiments. Using the REST API, CAP scripts can execute all the actions required for adaptation such as monitoring nodes, deploying code, collecting outputs, etc.

For the experiments, we defined our system as composed of eight M3 sensor nodes and three A8 nodes. On this system we deployed three applications. The CAP management components (scripts) are deployed on another A8 node and yet another A8 node is used for logging purposes. The initial selection of the nodes for each application is done manually by addressing each node's IPv6 address. However, at runtime CAP builds up its resource manager table, collecting IPv6 addresses of all the nodes. We also provide manually the information on the sensing services offered by each sensor through the respective IPv6 address. This information should be collected automatically in the future. The experimental setup is shown in figure 4.15 in which we can identify the following components:

- Testbed Local Storage: The main interface of the testbed allocates a local storage on a server with a front-end GUI through which the user can upload various files and save them for future usage in the experiments. This local storage is also used to save any data/output produced during the experiments.
- Remote Connection to Local Storage: A remote connection is available to access the local storage to upload or download files. It is also used to monitor the status of the running experiment via a dashboard.
- Testbed deployment: Through the Remote Connection the user can allocate different files to different nodes in the testbed and assign various parameters such as time duration of the experiment, location to store output data, etc.
- Nodes: Once the user determines the application assignment, the testbed will automatically connect to the nodes and deploy the respective scripts/application. In some cases, the testbed may also redeploy the whole image of a node if needed.

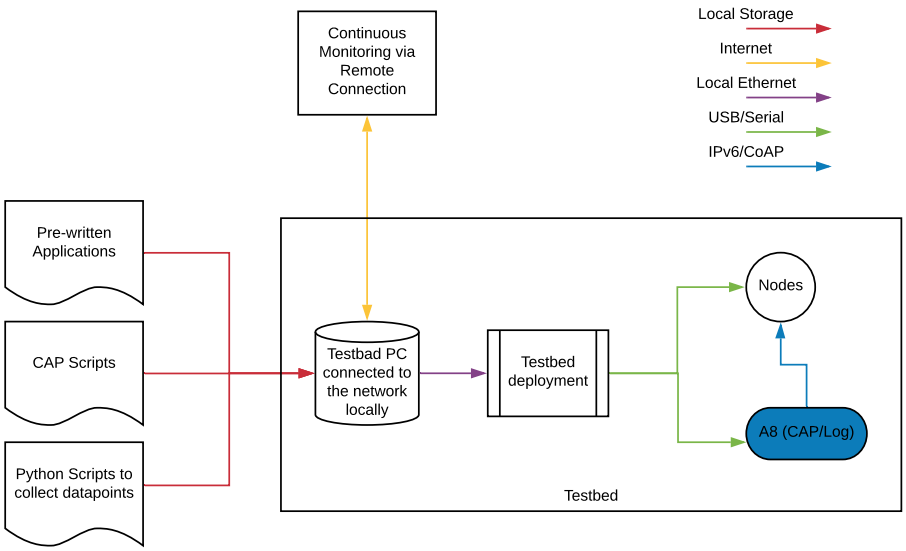

Figure 4.15: Experiments execution setup within the testbed.

The hardware architecture of the experimental setup is displayed in Figure 4.16. Note that we use a logical star topology for the management of the experiment, with the A8(CAP) management node communicating directly with all other nodes, both sensor nodes as well as the A8(log) logging node.

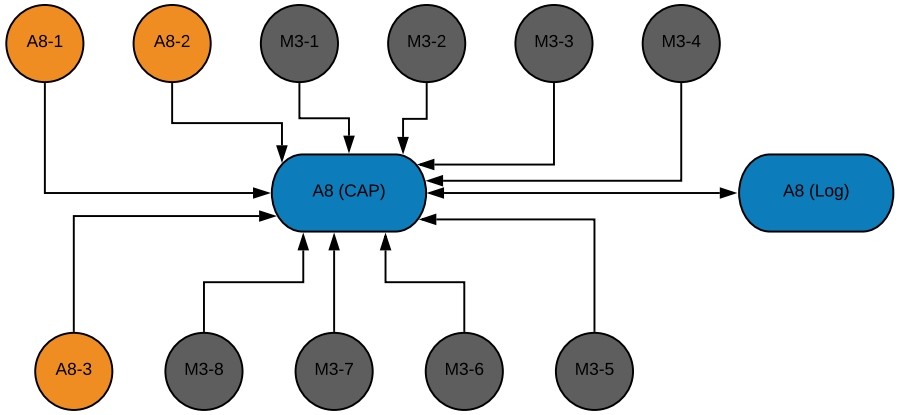

Figure 4.16: Hardware setup in the testbed, showing the network topology used for the management of the experiment. Note that A8(CAP) communicates with all sensor nodes, while A8(log) just communicates with A8(CAP).

In this testbed we deploy three applications, A, B and C, described and initially deployed (using IPv6 addresses assigned manually) as follows:

- **Application A**: Measures average temperature across multiple nodes.
  Deployed on: M3-1(host), M3-3(Input), M3-4(Input).

Table 4.4: Contexts used in the experiments for applications A, B and C.

Context 1	App. A: node M3-8(temperature) is ON, replace input node M3-4
Context 2	App. A: node A8-3(temperature) is ON, add input node
Context 3	App. B: node M3-6(temperature) is ON, add input node
Context 4	App. B: node M3-7 is OFF, relocate host
Context 5	App. C: node A8-2(GPS) is ON, add host node

- **Application B**: Measures average pressure and temperature across multiple nodes.
  Deployed on: M3-7(host), M3-5(Input)

- **Application C**: Monitors nodes GPS and logs it on an external base station.
  Deployed on: A8-1(host), A8-3(host)

On the referred applications we define five contexts (Table 4.4) that correspond to configurations that require certain actions. Specifically, we define two contexts for Application A, two for Application B and one for Application C. Therefore, as new nodes are automatically added or removed, context changes occur, triggering CAP to reconfigure the applications appropriately. This addition / removal of nodes occurs by turning them on or off, randomly, mimicking mobility, energy depletion, crashes, etc. All contexts used in the experiments are listed in Table 4.4.

These contexts and associated reconfigurations are all described and executed using a script. For example, Context 1 consists of replacing node M3-4 with M3-8 in Application A, while Context 2 consists in adding node A8-3 to the Application A. These changes must be duly accounted for in the respective processing function.

However, there will be additional context changes due to connectivity, energy and deployment issues. CAP will try to adapt to those as well, looking for alternative deployments. For example, if node M3-1 fails, CAP will look for another active node that can host Application A and, if so, transfer the application to it. Overall, CAP will adapt for the context changes included in one of the following three categories as we discussed earlier: (i) change of host; (ii) change in input node; and (iii) addition of new input node.

In the experiments, we only considered the three kinds of context changes and the results integrate all kinds. The Context Manager was also developed to satisfy these changes, however the design aspects allow for evolution as discussed in earlier section about Context Manager. The context changes were added incrementally, at a predefined rate. Then, after a period of 48h, the experimental procedure increased by one the number of context changes that would be generated at a time per application. This increment repeated every 48 hrs. Note that simultaneous context changes means that different changes occurred in a period shorter than the reconfiguration time. In this case, CAP serializes the handling of the context changes and performs them in sequence[2].

Finally, the A8(log) node would restart autonomously in case of a crash for any unforeseen reason, which could lead to temporary loss of data. The case of permanent failure of this node,

[2]Other approaches to handle simultaneous context changes are possible, too, e.g., ranking their importance and performing just the most important one, possibly aborting an on-going reconfiguration with lower importance.

e.g. due to energy depletion, was not considered, but it did not affect the experiments, either. The A8(log) node was also supervising the A8(CAP) node, which is responsible for the whole management operation of CAP. Its higher activity implied higher energy consumption, thus permanent crashes. In this case, this situation would be detected by the A8(log) node that redeployed all CAP scripts and functionality on another A8 node available in the testbed, which would take over as a new A8(CAP) node, thus testing a full restart of the whole system, too.

4.4.2.1 Experimental Results

We ran the experiments in the conditions referred above, continuously and autonomously, for nearly 16 days, with 8 cycles of 48h. We collected 563 individual context changes, which may have occurred separately or together with other context changes. These data points were collected by a log script that wrote them, regularly, on a text file on the A8(log) node. Out of the 563 data points, 100 represent incomplete reconfigurations due to various situations, e.g., changes or errors occurring during reconfiguration actions, that needed a restart of the reconfiguration process. These incomplete context changes are still included in the results reported ahead, whenever meaningful. Note that we do not intervene to check those specific situations, since we wanted to show CAP could manage to restart the corresponding processes autonomously, in case of failures.

In this section, we discuss several aspects of CAP performance arising from the experimental results, namely:

- periods (and variability) of continuous operation of the A8(CAP) node against number of simultaneous context changes;
- the number of trials to complete a single context change in the scope of an application;
- number of context changes occurring during different periods of A8(CAP) node continuous operation before redeployment on another node;
- simultaneous context changes processed to completion;
- time taken by CAP to prepare and deploy new code in one context change.

Figure 4.17 shows the average hours of continuous operation of the same A8(CAP) node, which runs all CAP management scripts, against the number of simultaneous context changes. The number of hours are obtained as follows. Each context change data point occurred within a period of continuous operation of the CAP management node. The values displayed in the figure are the average of the duration of such periods, for all data points grouped by number of simultaneous context changes.

Overall, the figure shows that, when there are single context changes, only, CAP runs on average for 25 hours continuously. This value is sustained approximately until four simultaneous context changes, but for higher numbers of simultaneous context change events, there is a clear trend towards shorter average periods of A8(CAP) node continuous operation. Our interpretation is that higher numbers of simultaneous context changes imply, by design of the experiments, higher

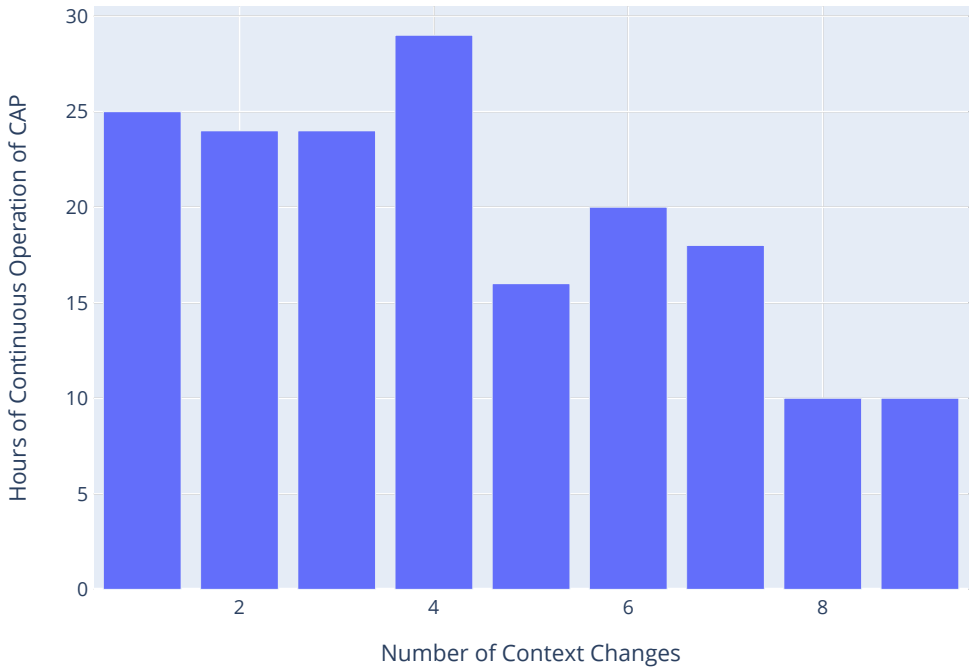

Figure 4.17: Hours of continuous operation of the CAP management node, A8(CAP), with increasing number of simultaneous context changes.

total number of context changes and, consequently, higher execution load on the A8(CAP) node, increasing its energy consumption.

Figure 4.18 shows the variability observed in the periods of continuous operation of the A8(CAP) node for different numbers of simultaneous context changes, using a box-plot diagram. This variability is expected to be relatively small, which is consistent with the observations, since most context changes occur within the same periods of CAP continuous operation, or even when they occur in different periods but with similar number of simultaneous context changes, the computing load should be similar and so should the longevity of the management node.

We believe that an important source of variability is the restart of reconfiguration processes caused by communication errors or topology changes, both explicitly generated, as when nodes are switched on or off, and spontaneous, as when nodes crash or run out of energy. To check this hypothesis we looked into the concrete case of Application A and we plotted the number of trials to enforce each of its context changes. This is shown in Figure 4.19, where we can observe that most context changes were applied at once (single trial, no restart needed) and a few more took one restart, only (2 trials), followed by a relatively long tail of few sporadic longer restart counts.

Still concerning the periods of consecutive operation of the A8(CAP) node, Figure 4.20 shows the number of context changes (Y-axis) that occurred in a given period of CAP consecutive operation (X-axis). The figure clearly shows that the majority of the context changes occurred in

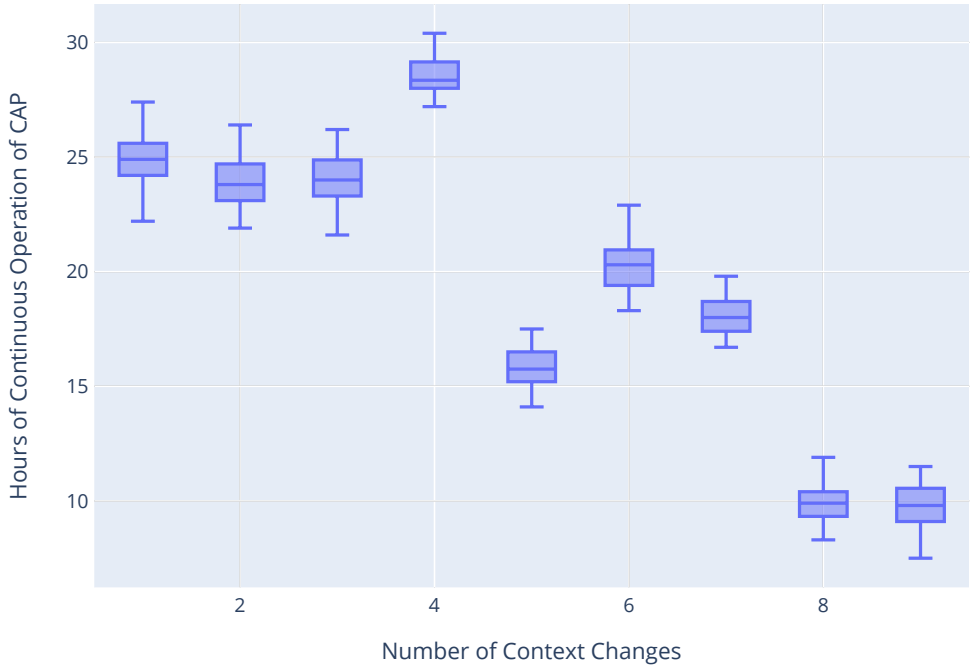

Figure 4.18: Variability of periods of continuous operation of the CAP management node, A8(CAP), with increasing number of simultaneous context changes.

periods of 23h to 27h of consecutive operation, which, as we saw before, correspond to periods with up to 3 simultaneous context changes. This figure also shows that the averages of continuous operation in Figure 4.17 and associated variability in Figure 4.18 are built with significant differences in the numbers of data points considered. This is particularly relevant for the cases of 4 or more simultaneous context changes, that show operation intervals that were observed with fewer data points.

The referred asymmetry in the number of data points of context changes occurring in different intervals of CAP continuous operation is explicitly shown in Figure 4.21 in which we can see the distribution of simultaneous context changes that were processed to completion. Here we see that approximately 1/3 of all context changes are single changes (150), and approximately 2/3 are changes with up to 3 simultaneous context changes, while the remaining 1/3 are changes with 4 or more simultaneous context changes.

Finally, Figure 4.22 shows the time taken by CAP on the A8(CAP) node to redeploy code for the cases of a single context change. This is the time taken from the point when CAP scripts are notified of the change, start parsing the code to be adapted, compile the code until the new code is deployed on the target node. The time is computed using the timestamps logged by CAP upon the detection of each context and upon the successful completion of the deployment. Even with a relatively simple computing node running on limited energy, the average time for such interval

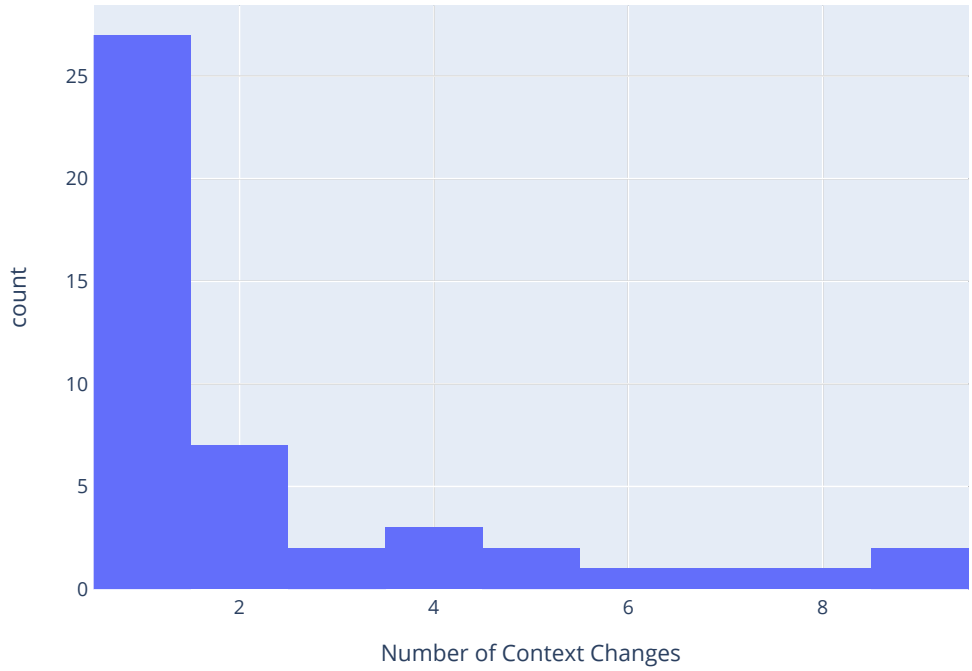

Figure 4.19: Frequency of the number of trials needed to complete each context change in Application A.

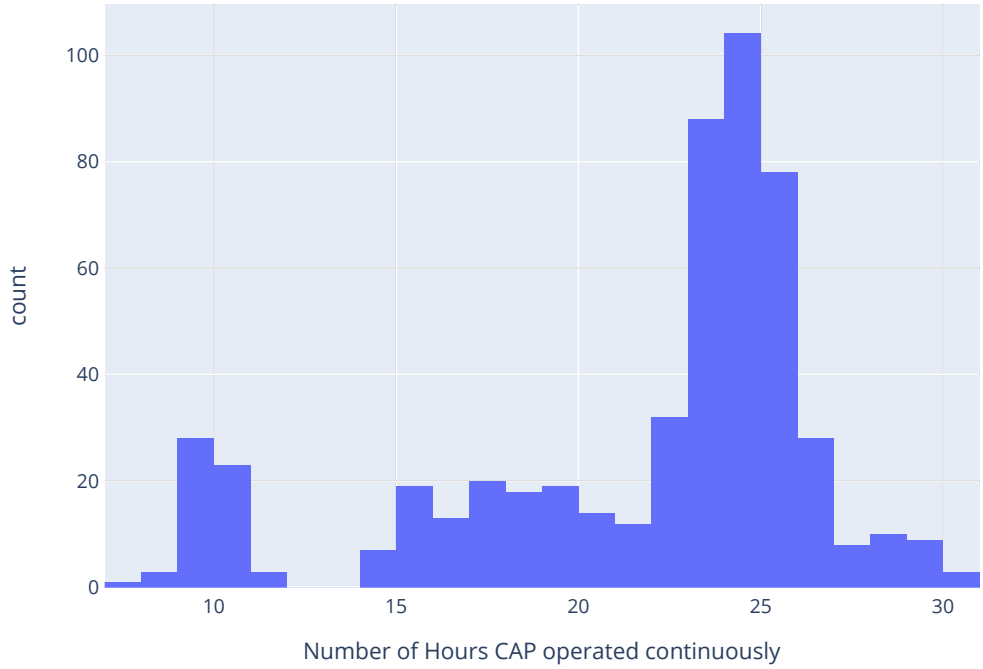

Figure 4.20: Number of context changes against the duration of the period of CAP continuous operation in which they occurred.

varied between 40ms and 50ms, with an average at approximately 45ms, which we consider a rather small time. However, the time reported above applies to single context changes, only, and

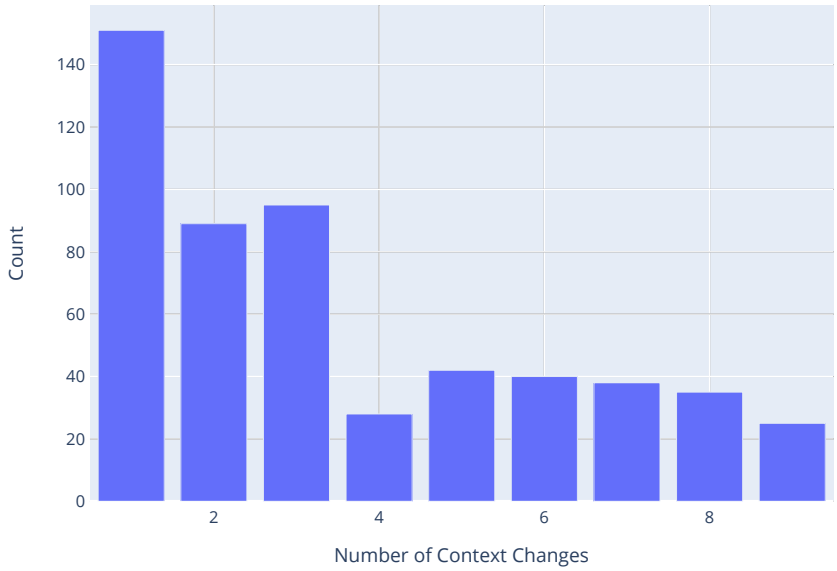

Figure 4.21: Histogram of simultaneous context changes processed to completion.

does not include the time to resume the application in the target node. Moreover, it should naturally vary with the size of the code to be adapted, too.

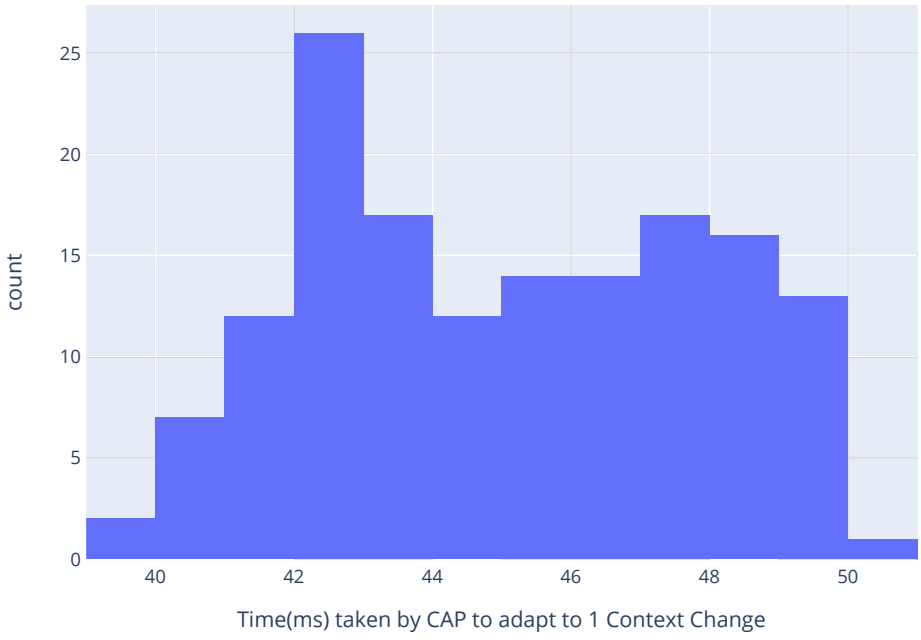

Figure 4.22: Mean time in ms for CAP to deploy new code for one context change

4.4.2.2 Summary of Resilience Results

Probably the most relevant result from these experiments is the validation in practice of the capacity of CAP for continued full autonomous operation, even under considerable stress of context changes. In this sense, CAP showed resilience and adapted autonomously to all the changes within the declared contexts as well as any other changes that, in spite of not being in the context list, occurred and could be classified in one of the three groups that CAP handles (change of host, change of input and addition of input). Other changes beyond these classes were ignored by CAP, as expected. CAP also resisted consistently to spurious communications errors, asynchronous crashes, and other similar situations that occasionally prevented the correct completion of the respective context change.

Another interesting observation, though intuitive, is the impact of the intensity of context changes, i.e., rate and number, on the system overhead and energy consumption. Naturally, the higher the rate and number, the higher the overhead and energy consumption by the CAP management.

During all experiments, it is assumed that the cost and quality of communication between different nodes is maintained by the infrastructure. We also assume that the limitations of such costs would not affect quality of context-awareness. For example, we assume that the nodes would have reported their aliveness status to a certain centralized system already. CAP only utilizes that aliveness status for the desired adaptation. However, if any system is not built with those assumptions, deploying CAP will increase cost of communication and impact energy usage of each node in significant way.

4.5 Summary

In this chapter, we have proposed a framework to support Context-Awareness on WSN and described essential features required for such framework. The framework, called Context-Aware Programming, or CAP, combines multiple Python scripts with a Django-based web application to provide autonomous adaptations for different contexts. This proposed framework and its capabilities were tested and debugged using local and remote experiments using TelosB sensor nodes and TinyOS software. These experiments validated the three essential features that we needed for the framework, namely: Abstraction, Modularity and Mobility. Further experiments with fully autonomous operation during an extended period of time, using the IoT-LAB facilities, tested the resilience of CAP and its capability to withstand strong loads of context changes, consistently.

The proposed framework is a proof of concept of Context-Aware Sensor Networks, or CASE, enabling context-awareness in WSN and mobile sensor systems. To our best knowledge such a complete framework has not been done by any of the previous works reported in the related literature.

The current implementation of the CAP is limited, though. In the future, CAP can be extended to include more advanced features. A key feature would be to provide faster adaptation with

switching between multiple contexts or a group of contexts. Another future goal could be to include complex data from multiple sources such as web services (e.g. calendars and SMS) and recognize context with such data. Basic machine learning algorithms can help in recognizing such context with more reliability. Efforts to evaluate such contribution against existing context-aware work in mobile computing would be critical, as well.

The implementation explained in this chapter can serve as a building block for more complete systems or advanced tools. In the same way that we took T-Res and extended it to mT-Res and then CAP, a similar approach is possible with the framework proposed in this chapter. Such a framework would include various aspects such as application management, resource management, and network management. Each aspect entails its own research problems. For example, to design an efficient resource management, it may be required to create a dynamic schedule to provide assurance on deadlines for different applications with the switch between contexts. We will address this issue later on, in Chapter 6.

Chapter 5

Adaptation Policies for Context-Awareness

The Context-Aware Programming introduced in the previous chapter provides basic features for writing code by different users for various goals. However, CAP as presented earlier, supports context changes just by re-connecting and re-locating to different resources the applications tasks provided by the user. The code of such tasks remains immutable. Thus, writing WSN applications that can adapt to changes in operational scenarios, be it changes in physical parameters, such as location, or system changes, such as availability of nodes, energy levels, etc., requires that programmers anticipate all possible future scenarios and thus program actions to adapt accordingly. This is limiting and, in practice, constrains adaptability to topological adaptability, i.e., adaptations are achieved with changes in the set of tasks and their deployment on the WSN resources.

This limitation can be strongly attenuated by supporting dynamic and more adaptable code to face context changes, providing code adaptability, i.e., the tasks code can itself change. However, despite the efforts already described in Chapter 2 to improve support to WSN programmers and end-users and to context-awareness (Afanasov et al., 2014), these efforts still lack an easy way of writing adaptations policies, particularly lacking an adequate abstraction that could foster reusability.

In this chapter we study how a programmer would establish such adaptation policies and we show the difficulties in achieving so. In addition, we demonstrate the need for writing such adaptation policies considering a couple of use cases from different domains. With the help of these use cases, we also examine essential features to build a generic model. Based on that model, we propose a new programming abstraction, named AdaptC, for writing adaptation policies that can be reused and extended, to support continuous and complex adaptations. This is the second most relevant contribution of this thesis as shown in Section 1.7.

Along this chapter we specifically provide the following:

- Design features for setting continuous and complex adaptation policies.

- A novel programming abstraction for adaptations that can be used with languages such as nesC.
- Examples of the ability to reuse and extend the adaptation policies written using the proposed abstraction.

5.1 Design Features for Code Adaptations

In this section, we identify the design features which are critical in defining the desired adaptation policies. These design features can also express the complexities in writing these applications from a programmer's perspective. To express the design features we propose the generic model in Figure 5.1.

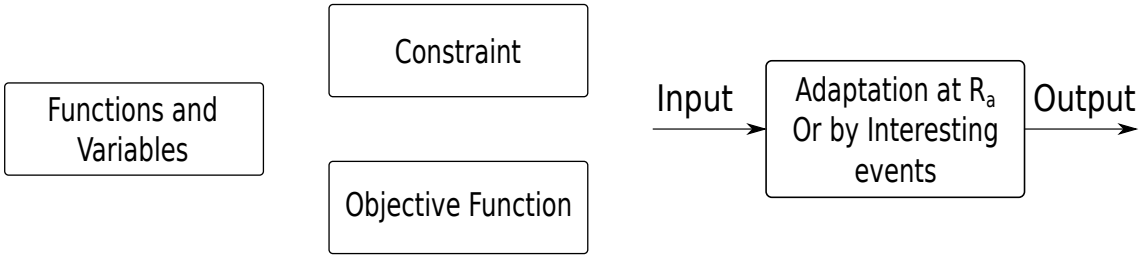

Figure 5.1: Generic model for design features

This generic model is composed of several elements. *Functions and Variables* represent all the relationships between different variables of the application that are relevant to the desired adaptation. These relationships can either exist from design or can be obtained with the progress of the application. *Constraint* describes the restrictions for the adaptations. *Objective Function* represents the desired outcome for the user. There must be some relationship between *Constraint* and *Objective Function*, but it is optional to have a direct relationship.

The component on the right expresses the dynamics of the adaptation policy. Adaptations can either be carried out periodically at a *Rate of Adaptation* or triggered by other events taking place in the system. These other events are called *Interesting Events* in the generic model. The user can tag different elements of the program as Interesting Event and the adaptation will happen every time that element of the program is triggered. These Interesting Events can also impact on the *Functions and Variables* component. All these changes are included in the next adaptation cycle.

To better explain the design features, we consider two different use cases and try to apply the proposed model to these use cases.

5.1.1 Usecase 1: tracking the GPS of a wild animal

Consider a use case in which a wildlife animal is tracked by a GPS-enabled sensor node attached to the animal itself. Base stations are deployed across the forest to collect the GPS data from the sensor node on the animal. The primary objective is to sample the GPS at a fixed rate and store the movements of the animal locally at the node itself. Whenever a base station is encountered

all the recorded data can be transferred and then delivered to the user through the base station. After this, the user also wants to assure that the sensor node maintains sufficient battery level to encounter the next base station. To achieve that, the GPS polling rate must be adapted according to various situations such as the speed of the animal, other applications running on the same node, the amount of energy left or any encounter with other animals. The programmer can not anticipate the speed of the animal and program a GPS sensor polling rate for all the possible situations. Also, some new events which may affect such adaptation may be added later on. Hence, there is a need to support continuous adaptation capabilities while programming the goals. This adaptation policy can be expressed, using pseudo-code, as in Listing 5.1.

```
For Speed (S) and Battery Level (B):
    Poll GPS with Sampling Rate (R)
    keeping Battery Level (B) above Threshold (B_T) after Time (t)
```

Listing 5.1: Pseudo-code for the GPS adaptation policy.

The variables in this use case are Speed of the animal (S), Battery Level (B), Sampling Rate (R) and Time (t). There is a direct relationship between how frequently the GPS is sampled (R) and the level of the battery (B). This can be either pre-defined by the user based on earlier studies (Ben Abdesslem et al., 2009; Carroll and Heiser, 2010) or studied by the system over time. Here, we can express it as $B = f(R)$. This relationship helps to learn how much B will be affected by changes in R. There is a constraint on how much B can be affected. B should always be above a threshold battery level (B_T), i.e. $B \geq B_T$. While the system must follow the constraint, the user may desire that the polling rate must be maximized in order to get as much fresh GPS data as possible. That would require a complex and continuous adaptation for this application and maximizing R would be the objective function for this adaptation.

The user can decide a fixed sampling rate at which this objective function should be solved, which will be the rate of adaptation (R_a). Also, some interesting events within the program can also trigger the adaptation. For example in this use case, these events can be *An encounter of another base station, Change in the number of other applications, Not able to get GPS position, Encounter of another wildlife animal,* etc. Figure 5.2 shows the design features for this use case.

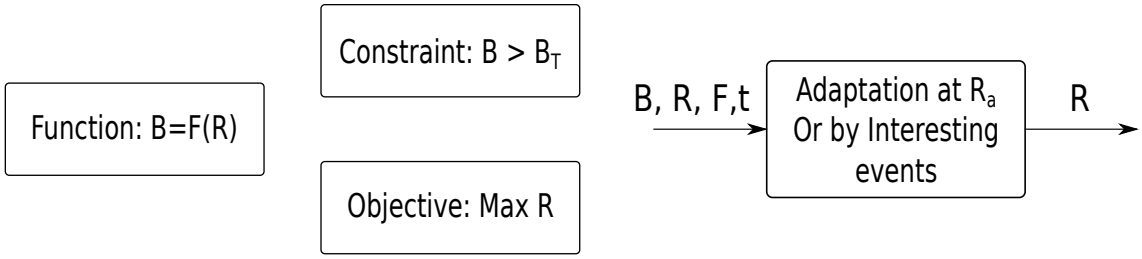

Figure 5.2: Design features for the GPS use case.

5.1.2 Usecase 2: controlling an HVAC system

We consider a Heating, Ventilation, and Air-Conditioning system for building automation like those discussed by Erdelj et al. (2013); Deshpande et al. (2005). HVAC systems are a common utility in modern infrastructures, such as offices, shopping malls or industrial buildings. Typical HVAC systems have different sensor nodes to monitor physical conditions such as temperature, pressure, humidity in the various parts of the building. According to user requirements for those physical parameters, certain actuation is performed on various cooling or heating devices or any other actuators. In some HVAC systems, there is only one user requirement such as maintaining a certain temperature. But in the case of more complex buildings, the user might want to minimize the cost of operations while satisfying multiple user requirements. During the operation, the HVAC system should adapt to different events to minimize the cost and provide sufficient performance. Also, adaptation is expected in other cases such as offices where different occupants might have different requirements and their behavior might affect the HVAC performance. Hence, it becomes difficult for a programmer to write an application which can keep up with all the above-mentioned factors and achieve its main goal. Again using pseudo-code, we can express this adaptation policy as in Listing 5.2.

```
For Volume (V) and time (t):
    Provide Power (P) to maintain Temperature (T_F)
    with Cost of operation (C) minimized over time period (t)
```

Listing 5.2: Pseudo-code for the HVAC adaptation policy.

The variables in this use case are Volume of the space (V), Power Consumption (P), Temperature of the space (T_F), Operational Cost (C) and Time (t). There is a direct relationship between the temperature being maintained (T_F), the volume of the space (V) and the power consumption (P). There is also a direct relationship between the power consumption (P) over time (t) and the operational cost (C). These can be either pre-defined by the user based on statistics or studied by the system over time. Here, we can express these as $T_F = f(V,P)$ and $C = g(P,t)$. These relationships show which variables can affect operational cost and how changing them can help in achieving the objective of the user, i.e. minimizing operational cost ($min(C)$). The constraint is on temperature since the system must always maintain a suitable temperature as well. If the required temperature is T_S, then there can be some tolerance (δ) to minimize operational cost. Hence, that would be the constraint for this use case, i.e. $T_S - \delta \geq T_F \geq T_S + \delta$. This would require a complex and continuous adaptation and minimizing cost would be the objective function for this application.

Concerning the dynamics of the adaptation, the user can decide a fixed sampling rate at which this objective function should be solved, which will be the rate of adaptation (R_a). Also, some interesting events within the program can also trigger the adaptation. For example, in this use case, these events can be *Change in the occupied volume in the space, Body temperature of the occupants, Location of the occupants, Interaction with outside environment when doors open or close,* etc. Figure 5.3 shows the design features for this use case.

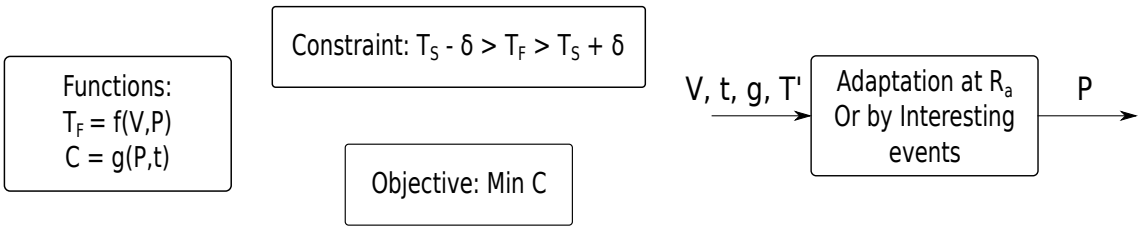

Figure 5.3: Design features for the HVAC system use case.

5.2 Programming Adaptation Policies

Let us focus on how a programmer would write adaptation policies for WSNs, with current state-of-practice approaches, using the C language. In particular, let us examine the use case 1 (GPS) presented in the previous section, for which we provide the code in the Listing 5.3. The relationship between the BatteryLevel and the GPS polling rate is described by the Function in line 5, which takes the Rate as input. In this function, we assume *alpha* is the factor by which the GPS sampling rate affects the battery level. In order to maintain the normal operations and the adaptation simultaneously, the programmer must create multiple threads. In line 9, `sensing_thread` is the function that describes the normal behavior of the application using all the parameters provided by the user. Another thread is required for adaptation, which is called `adaptation_thread` in line 23. This function checks the BatteryLevel using the earlier functions for current Rate. Then it gets a new polling rate under the constraint of keeping BatteryLevel above a threshold defined by the user. It uses *haversine* function to calculate the speed from the GPS coordinates in line 28.

One of the design features is that the adaptation must either occur at a fixed rate or triggered by some interesting events across the application. Hence, a third thread has to be created to trigger adaptation at a fixed rate by using timers, which is defined in line 16. In addition, the `adaptation_thread` must be called back inside all the functions that may generate interesting events.

As shown in Listing 5.3, the pseudo-code for the adaptation includes threads, timers, function dependencies, etc. With dynamic and complex adaptation policies it becomes difficult for the programmer to write the code. For example, if the programmer wants to initiate the adaptation at a fixed time every day or according to the time stamps of particular data in addition to the rate of adaptation, then the programmer must start a new timer for that purpose. That creates additional complexity when the programmer wants to add new interesting events over time, change the relationships between variables, or change the solution itself. For all these changes, the programmer must dive into the low-level details which are not always necessary for each change. If the programmer just wants to add a new time-stamp to trigger the adaptation, it should not require extensive knowledge of timers and threads in the whole program. Hence, there is a need for an abstraction that can enable the programmer to achieve goals without delving for every low-level detail. Such an abstraction did not exist until now, to the best of the authors' knowledge, and it has further implications since it also brings the ability to reuse and extend adaptation policies.

```
1  int Solve(Speed, BatteryLevelcurrent ) {
2      // Calculate R using the solution for optimization
3      Return R; }
4
5  int Function(Rate){
6      BatteryLevel = alpha*Rate;
7          return BatteryLevel; }
8
9  void sensing_thread () {
10         while() {
11                 timer = clock();
12                 sleep(Rate);
13                 GPS[time] = getGPS();}
14         Return 0; }
15
16 void timer_thread () {
17         while(){
18                 timer_set (timer2, Ra);
19                 if ( timer_expired (timer2)){
20                         adaptation_trigger = 1;}
21                 Timer_reset(timer2); }}
22
23 void adaptation_thread () {
24         While(){
25         If ( adaptation_trigger == 1) {
26                 BatteryLevel = Function(Rate);
27                 If ( BatteryLevel < Batterythreshold ){
28                         Speed = haversine (GPS[time-1:time]);
29                         Rate = Solve(Speed, BatteryLevel );}
30                 Timer_reset(timer2) // reset the timer
31                  adaptation_trigger = 0; }}}
```

Listing 5.3: Pseudo-C code for the GPS use case.

5.3 Proposed Abstraction

In this section we propose, AdaptC, a high-level programming abstraction that can enable not only ease in programming but also ease in debugging and re-usability of the adaptation policies written by the programmer. One of the examples of the complexities with the current state of practice, shown in the code in Listing 5.3, is that the interesting events are not integrated into the

code. Hence the programmer would have to always remember which functions to check for the adaptation. AdaptC can provide a better structure to write complex adaptation policies for various applications.

```
1  Block Function f {
2          Use variables b, c, d
3          // operation
4          return b }
5  Block Function g {
6          Use variables a,c
7          // operation
8          return a }
9  Block Constraint B {
10         // define the constraint
11         return true / false }
12 Block Adaptation {
13     // Adaptation here
14         If Trigger = Active:
15                 Solve a
16         return a }
17 Block Solution S {
18         Use Function f
19         Use Function g
20         Use Constraint
21         Uses Variables a b c d
22         // solve
23         return a }
24 Block Trigger T {
25         // Combination of different triggers
26         // a fixed rate
27         Use consecutive_time 10s
28         // at fixed system time
29         Use time_stamp 00:00
30         // use different flags or events
31         Use flags }
```

Listing 5.4: Abstract pseudo-C code of AdaptC.

Listing 5.4 shows an implementation of AdaptC. We divide the code into five blocks based on the proposed generic model of the design features described in Section 5.1. These blocks are named *Function, Constraint, Adaptation, Solution* and *Trigger* and are explained next:

- The **Function** block, or possibly a set of blocks, expressed in line 1, define(s) all the associated variables and their relationships. The number of Function blocks depends, by convenience, on how to define multiple relationships between the different variables. There can also be local variables used in these blocks and this allows ease of access in defining these relationships.
- The **Constraints** block in line 9 allows the programmer to define one or many constraints that affect the adaptation. This block will return a Boolean result of true or false depending on whether the constraints are satisfied or not, respectively.
- The **Adaptation** block in line 12 checks for the trigger and if the trigger is active it invokes the Solution block.
- The **Solution** block in line 12 solves the objective function subject to the defined constraints. Once a solution is achieved, this block returns it to the Adaptation block.
- The **Trigger** block in line 24 allows the programmer to define when the adaptation must occur. In this block, the programmer can have fixed timers, periodic timers, or flags across the application. Hence, the programmer can have a flag named *interestingEvent*, use it across the complete application and whenever that flag is raised the adaptation is triggered.

As an example, Listing 5.5 shows AdaptC applied to the HVAC system use case and Listing 5.6 shows AdaptC applied to the GPS use case.

```
1  // The solution for desired goals
2  Block Function_TEMP {
3          Use Volume, Power
4          //calculate Temperature
5          return TEMP }
6  Block Function_COST {
7          Use Time, Power
8          //calculate COST
9          return COST }
10 Block Constraint_TEMP {
11         //define the constraint
12         if T_{S}+\delta<TEMP<T_{S}-\delta:
13                 return true
14         else
15                 return false}
16 Block Adaptation_Power {
17     // Adaptation here
18         If Trigger = Active:
19                 a = Solution_COST()
```

```
20                  return a }
21 Block Solution_COST {
22          call Function_TEMP()
23          call Function_COST()
24          //calculate required power
25          return Power }
26 Block Trigger_HVAC {
27          //Combination of different triggers
28          // a fixed rate
29          Use consecutive_time 10s
30          // at fixed system time
31          Use time_stamp 00:00
32          // use flags or events across application
33          Use flags }
```

Listing 5.5: Abstract pseudo-C code for the HVAC system use case.

```
1  Block Trigger_GPS{
2  //Combination of different triggers
3  // a fixed rate
4  Use consecutive_time 10s
5  // at fixed system time
6  Use time_stamp 00:00
7  // use flags or events across application
8  Use flags
9  }
10 // The soltuion for desired goals
11 Block Solution_FRESHNESS{
12 call Function_BatteryLevel()
13 //calculate remaining BatteryLevel
14 return Power
15 }
16 Block Constraint_BatteryLevel{
17 //define the constraint
18 if B is greater than B_{T}:
19 return true
20 else
21 return false
22 }
23 Block Function_TEMP{
24 Use Polling Rate and BatteryLevel
```

```
25 //calculate BatteryLevel
26 return B
27 }
28 // Adaptation here
29 Block Adaptation_RATE{
30 If Trigger = Active:
31 R = Solution_FRESHNESS()
32 return R
33 }
```

Listing 5.6: Abstract Pseudo-C code for GPS Use Case

5.4 Early Assessment of AdaptC

To validate the feasibility of the proposed abstraction, AdaptC, we implemented it for Contiki using nesC (Gay et al., 2003) and Python to support only basic functionalities as a proof-of-concept. For a concrete example, we implement the HVAC application using AdaptC and we compare it against an implementation following state-of-practice approaches. In this section, we discuss the technical details of that implementation and highlight some characteristics that help in understanding the benefits of AdaptC. The same characteristics can also apply to many other applications such as smart homes, smart irrigation systems, etc.

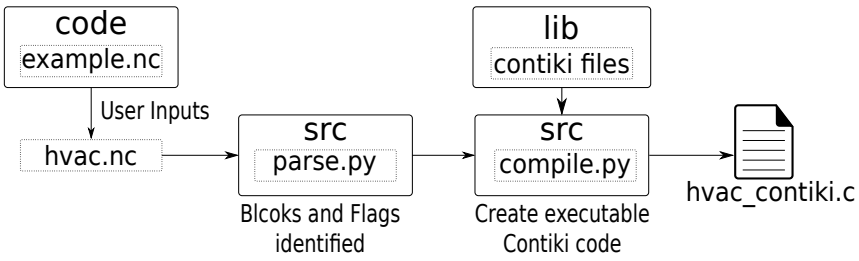

Figure 5.4: Implementation of the AdaptC abstraction.

AdaptC is implemented to parse the nesC code provided by the user and identify different *blocks* as mentioned in Listing 5.4. Then those are compiled into Contiki code. All of this is done using Python scripts. The implementation is explained in Figure 5.4. To use AdaptC, a programmer must obtain the repository. It is divided into three directories, *code*, *lib*, and *src*.

The *code* folder contains already written examples. The programmer can find files such as *example.nc* written in nesC equivalent to the abstract pseudo code showed earlier in Listing 5.4. This can be adapted for the intended application by the programmer. For example, in the case of the HVAC application, the *example.nc* code can be modified according to Listing 5.5.

The *lib* folder contains libraries built for the compilation of *example.nc* code into a workable Contiki code. That code can be executed inside Contiki or deployed using Contiki. These libraries are written in C and modified from the Contiki repository available on GitHub (Dunkels, 2003).

The *src* folder contains scripts written in Python. The *parser* script reads the code provided by the programmer such as *example.nc* and identifies each block in it. The *compile* script uses the files from the *lib* directory to create the Contiki code and add each block from *example.nc* file in it.

With the help of AdaptC it becomes easier to reuse the same adaptation policies in different scopes, by just changing input parameters. For example, in the HVAC application, the programmer sets a constraint on the temperature which is affected by volume and power. Later on, another programmer might want to use the same policy in an area where the temperature is affected by other parameters such as the number of people, movement, etc. Hence, the second programmer can easily reuse the same adaptation policy by just slightly changing the input parameters. In addition to being reused, the applications are required to evolve with new requirements as well. For example, in wildlife monitoring using GPS sensor, the programmer might wish to monitor the health condition of the animal. This can add more interesting events such as any rest taken by the animal, elevation reached during the day, etc., and that could be easily achieved by adding these new events in the Trigger block.

A programming language must meet a few basic demands of the user community. We evaluate our abstraction, AdaptC, for those basic demands provided in (Borning, 2002) as shown in Table 5.1. We have already discussed the re-usability earlier. The ability to extend the code implies the ease of maintenance. Also using AdaptC the programmer is able to write diverse goals quickly since it takes away the complexities of timers, threads etc. Hence that enables rapid development. In addition, AdaptC allows adaptation policies to work on different nodes, hence the support of portability is provided. The design features and their modularity contribute to the ability to learn as it is easier to understand the role of each feature and their dependencies. The remaining properties, namely *Reliability* and *Efficiency* still remain to be evaluated.

Requirements	Development	Easy to Update	Reliable	Portable	Efficient	Learnable	Reusable
AdaptC	Yes	Yes	–	Yes	–	Yes	Yes

Table 5.1: AdaptC and common requirements for programming languages.

This is a preliminary evaluation based on software engineering concepts. Since we are not aware of any existing abstractions for adaptation policies to build WSNs, a direct comparison was not possible. Further evaluation of the performance requires implementing it with different applications, which we leave for future work. With those implementations, a more detailed evaluation of software engineering elements such as the impact on variables, lines of code and functions will be possible.

Despite the absence of macroprogramming work specifically for adaptation policies, there is still a great amount of work to support the programmer. Thus, we try to compare AdaptC with some of the relevant state of the art work as shown in Table 5.2. We selected two related frameworks aiming at providing some sense of adaptation in WSNs and re-usability for the applications, namely T-Res (Alessandrelli et al., 2013) and PyFUNS (Bocchino et al., 2015). Both aim at providing support to the programmer to write applications without the knowledge of low-level

features, i.e., they are hardware agnostic. However, there is no support to write adaptations for the applications that change at run-time, i.e., evolvable applications, while AdaptC is able to support both changes in the node hardware and application code.

Programming Abstractions	Hardware Agnostic	Evolvable Code
T-Res	✓	**x**
PyFuns	✓	**x**
AdaptC	✓	✓

Table 5.2: Comparison with existing abstractions.

5.5 Experimental Validation of AdaptC

To provide a simple practical validation of AdaptC, we reused the IoT-LAB setup used earlier for CAP in Chapter 4. We used the same applications as described in Section 4.4.2, except that the code for the application is replaced by the code mentioned in this chapter.

We ran the experiments for a shorter duration to observe the correct execution of adaptations and we also analyzed cross-applications impact. In fact, since the adaptation policies are directly involved in detecting the adaptation conditions, which in turn can be involved in multiple concurrent applications, we wanted to verify whether having adaptations detected in one application could interfere with their detection in another one.

Figure 5.5 showcases the cross-application impact that we observed. We have two applications, A and B, that share some resources in certain contexts, either as input or host nodes. We then observe if a context change in one application led to parsing code of another application and making changes in it, too. This indicates that the adaptation policies are being correctly applied even when multiple applications share resources. In the figure we see, in a total of 23 context changes, that 13 triggered adaptations in a single application, while 10 triggered adaptations in the two applications. These situations were traced and verified to be correct. Thus, we observed that CAP enforced the adaptations correctly in two applications written according to AdaptC, even when sharing resources.

5.6 Summary

In this Chapter we discussed the problems associated to writing complex adaptation policies for Wireless Sensor Networks. We exposed these problems using two different use cases with different user requirements. Drawing from those use cases, we have built a generic adaptations model that systematizes and clarifies the adaptation process.

Following the generic model for adaptation policies, we defined a new programming abstraction, AdaptC, that allows a programmer to write such policies without explicit dependence on low-level node features. In addition, we have implemented one of the use cases with and without

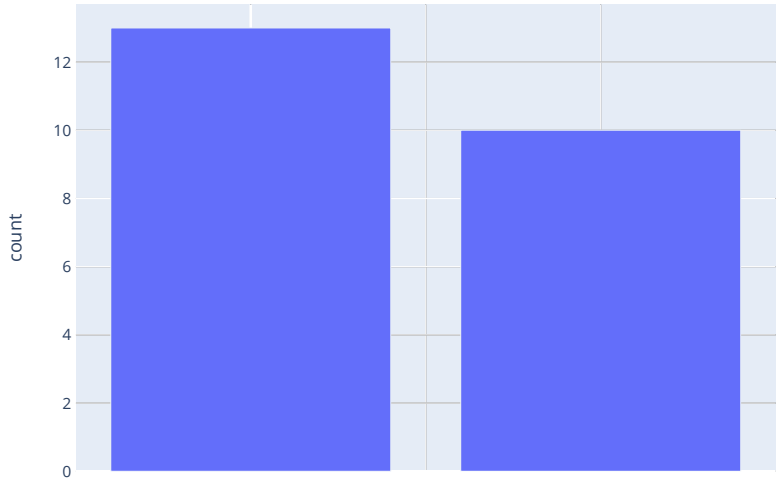

Figure 5.5: Changes with AdaptC on CAP affecting one application (left) and two applications sharing resources (right).

the AdaptC abstraction, in Contiki and nesC, and we could verify the desirable properties that AdaptC meant to provide.

Finally, we used AdaptC in two applications running concurrently in the IoT-LAB testbed with CAP and we could verify that the adaptations of both applications were triggered correctly, even when sharing resources. We believe that the use of AdaptC makes it easier for CAP to detect and isolate the code related each context change. This is the basis of its distinguishing capacity of supporting evolvable code, by allowing dynamic swapping of blocks in the applications and their respective redeployment at runtime.

Chapter 6

Network QoS Support for CAP

In previous chapters we discussed the requirements and the construction of CAP as a middleware to support Context-Aware Sensor Networks, as well as a programming framework, AdaptC, that allows users to write tasks for WSN applications that can easily adapt to different contexts at run-time. However, we focused on the programming support to develop such applications and associated middleware to run them. Conversely, we did not address the important aspect of Quality-of-Service, particularly in what concerns the timeliness of the communications, which in turn impacts strongly on the timeliness of the applications.

In this chapter we address the issue of network QoS in CAP, particularly using a traffic scheduling technique proposed previously in the context of real-time packet scheduling in WSN, namely Network Harmonized Scheduling (Gupta et al., 2014), and adapting it to fit our framework. NHS determines the communication resource, i.e., the network bandwidth, allocation, defining specific slots for packet transmission organized in a compact schedule. The main feature of NHS is the use of strictly harmonic periods for recurrent transmission slots and proper offsets to reduce the multi-hop propagation delay of data from sources to a designated sink, i.e., the end-to-end communication delay.

We will first revisit the resource allocation aspect and how it can be used within WSN to improve separation between applications, thus reducing mutual interference and improving their timing behavior. Then we will describe the NHS technique and, in particular, the construction of the NHS schedule. Finally, we will discuss the adaptation of NHS to CAP.

6.1 Resource Allocation

As referred earlier in this dissertation, WSN were initially developed with a single application in mind. Later on, different techniques, as referred in Chapter 2, provided multi-application support and allowed running multiple applications concurrently. One of these technique is resource reservation and allocation, which consists is allocating to each application a fraction of the physical resources involved, namely a fraction of the network and the nodes bandwidth. An adequate middleware allows doing the allocations and enforces the reservations to provide mutual isolation.

However, when considering Context-Aware Sensor Networks, the requirements of the applications vary as changes in context occur, potentially impacting the allocation of resources to applications [1]. This impact can take multiple forms. For example, a change in context can enable different applications to use the same resource, thus needing new resource reservations and applications allocation. A change in context may also lead to a structural change in an application, requiring a resource reservation to be modified with different parameters. Figure 6.1 shows two applications sharing a given resource. These applications react differently to two contexts. Application X runs under both context A and B. In turn, Application Y runs under context B, only. Thus, depending on which context(s) is(are) active at each moment, the resource is shared in a different way, possible requiring different reservations.

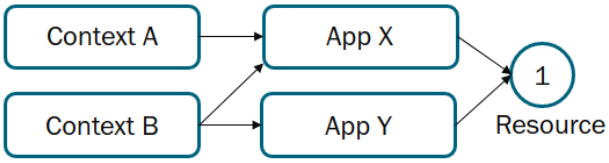

Figure 6.1: Resource allocation with multiple contexts.

To better illustrate this situation, let us take as example a system that tracks wild animals with GPS sensor nodes. The tracking of each concrete animal is an application. Different contexts may correspond to different locations, proximity to base stations, proximity to other animals, etc. In each of these contexts an application records the respective animal location using the GPS sensor, but may need to poll the sensor at a different sampling rate or even pause the polling. If the animal is in close proximity to a base station, the application can pause polling the GPS since the base station can provide location data for the animal. The polling rate also implies the rate of network communications. Thus, the applications of multiple animals may need different fractions of network bandwidth depending on the respective contexts.

Hence, in a context-aware framework, resources must be allocated and managed dynamically to satisfy applications requirements and constraints, as discussed next:

- ***Application Priority***. Priorities can be used to support resource sharing with asymmetric isolation, only. This means that higher priority applications are isolated from lower priority ones, but the opposite is not true. In a dynamic scenario, priorities may change depending on the active context. For example, an application might be critical in a certain context (higher priority), but not in other (lower priority). In the presence of multiple applications in the same resource, each context may imply different applications priorities, which may be generated dynamically according to an adequate function executed upon entrance in the respective context.

[1] In practice we allocate resources to the applications to enable their execution. However, we may also consider that we allocate the applications to the resources, so the applications can make use of the resources. Thus, without risk of confusion, we will use both directions interchangeably.

- ***Application Constraints***. Each context may imply a different set of constraints on the applications. For example, a given context may limit the rate of application execution, or determine its temporary suspension.

- ***Resource Constraints***. Similarly to the previous case, each context may also determine different availability of resources with their own constraints, be it in terms of memory, processing capacity, communication bandwidth or sensors/actuators available. For example, a node may become unavailable and the applications that were using it need to be re-allocated. In certain contexts, more powerful nodes may be available while in other contexts the nodes available can be severely resource-constrained, or lacking certain sensing capabilities.

Resource reservation/allocation is one major issue in WSN, but also generally in the IoT or in CPS. Certain techniques were already discussed in Chapter 2, such as virtualization. One technique available in the literature that is particularly interesting in our scope, and also as example, is DepSys (Munir and Stankovic, 2014). This is because DepSys allows resolving conflicts among different applications using priorities. DepSys is a dependency-aware system for specifying, detecting and resolving conflicts among different applications, e.g., multiple applications trying to control a light simultaneously. DepSys works with these conflicts using additional meta-data called *effect*, *emphasis*, and *condition*. However, two strong limitations prevent the direct use of DepSys in a context-aware setting. On one hand, the priorities are provided at design time and are static. On the other hand, the developers need to provide, using XML and for each application, information about the device to be used to run it.

Hence, despite solving the problem of application co-existence, DepSys has to be adapted to cope with the dynamism of context-aware applications. To resolve resource allocation conflicts during runtime as the constraints change with new contexts, an adaptation layer is needed. Listing 6.1 shows the pseudo-code of a possible algorithm to carry out such adaptation.

```
// R = set of resources available
// A = set of running applications
// C = set of active contexts
Resource_allocation (R,A,C) {
    for each application a in A
        Ca = set of active contexts affecting a
    Q = define_contexts (A, {Ca, for all a}, R)
    allocate (R,A,Q)
    }
```

Listing 6.1: Adaptation of the resource allocation at run-time.

The function *define_contexts()* decides which is the specific context to be used for each application, in case several of the active contexts affect the same application. This is important to define the actual priority of an application and the resources it needs. This can be done separately per

application, for example using the active context in which the application has the highest priority, or the highest criticality, or the lowest resource requirements, etc, but it can also be done holistically, for example, using the active contexts whose resource requirements better fit the resources available.

Then, once the specific contexts per application are defined, the function *allocate()* can resort to an allocation strategy like the one of DepSys, in which applications priorities are well defined so as the target resources.

Considering the example we used above with the system to track wild animals, we could pick at each moment the active context per application (per animal) that has the lowest use of GPS, to save energy and increase the lifetime of each application. Thus, if an animal would be in the vicinity of a base station, that specific context used would be the one with the GPS suspended and location extracted via base station. Another context could be defined when the battery level runs below a threshold, using a specifically low GPS poll rate, if not in the vicinity of a base station.

6.2 Network Harmonized Scheduling

In WSN, one of the most important resources, with a significant impact on the QoS of the applications, is the network bandwidth resource. Thus, in this section we will look into the scheduling of this resource, with the aim of supporting bandwidth allocation to multiple applications that leads to improvements in QoS with respect to end-to-end delays.

In order to allocate network bandwidth to the applications we need a network scheduling approach to manage the transmission of all concurrent flows triggered by the applications. For this purpose we are going to make use of Network Harmonized Scheduling (Gupta et al., 2014), which was proposed earlier in this thesis, in a joint research work, and which creates a single task in each node that combines all packet transmissions from the respective child nodes and forwards them up the network towards the sink. However, NHS was not implemented for multi-application WSN, nor to support dynamic contexts. In this section we will first describe NHS and how it can support multiple applications running on a multi-hop WSN and then we will discuss the use of NHS to support CAP and Context-Aware Sensor Networks in general.

6.2.1 NHS Basics

NHS is a simple and effective approach to build compact periodic schedules that is inspired from Rate-Harmonized Scheduling (Rowe et al., 2008) and applied to the context of multi-hop networking. When considering WSN, with multiple sensors transmitting recurrently their samples, even if these transmissions were originally periodic, the global pattern may not be periodic due to packet collisions, RF interference, packet losses and retransmissions, etc. If the WSN runs multiple applications, the total number of packets released by a sensor node grows proportionally to the number of deployed applications. Internal interference in each node may contribute to further degrade the periodic transmissions pattern and the additional network load may lead to increased contention at

different hops in the network. If the underlying MAC layer is based on a carrier-sense mechanism, additional load may, in turn, contribute to wide variations in network access delay.

Therefore, multiple nodes releasing packets independently in a WSN may increase the overall resource (network bandwidth) consumption and generate prohibitively large and non-deterministic communication delays. To overcome these issues, NHS aligns packet transmissions from different nodes in global harmonic periodic boundaries and leverages this periodic framework to consolidate the transmissions in the network in a global periodic pattern, but without any global state maintenance as we will show.

6.2.1.1 Multi-hop Data Forwarding

NHS groups periodic transmissions in batches from different devices per layer of neighbors in a cluster-tree topology to reduce the time that receiver nodes need to keep their radios turned on to receive the packets. Figure 6.2 shows an example topology where circles represent clusters. Each cluster head is shown as the node in a cluster that belongs simultaneously to the next cluster towards the sink (or root).

Note that in a WSN, most of the network traffic consists of flows of sensor data from the sensor nodes up to the sink node. Thus, it is a desirable target to schedule the communications so they start from the leaves of the network, i.e., the nodes that are farther away from the sink, and propagate upwards until reaching the root. In each cluster, the respective cluster head collects the transmissions of the cluster members, packs them in a batch and forwards them. For minimum end-to-end delay of the sensor transmissions, it is important that the sequence of transmissions is bottom up, i.e., the transmissions of the cluster with a hop count of n (assuming the sink cluster has a count of 1) should occur immediately before the transmissions of the previous cluster, i.e., with hop count $n-1$. This way, the head of cluster n, which is a member of cluster $n-1$, will have just received the data from cluster n nodes at the time of transmitting within cluster $n-1$, thus forwarding it with lower latency than other sequencing.

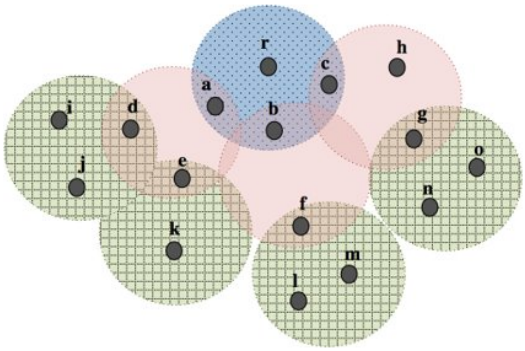

Figure 6.2: A multi-hop cluster-tree network.

6.2.1.2 Structuring Communications

Organizing the communications in the network according to strictly monotonically decreasing hop count leverages the periodic harmonic framework making transmissions fit in a global cycle with period T_H, the harmonizing period [2]. Figure 6.3 shows the communications schedule corresponding to the topology in Figure 6.2. Generally, this is a TDMA schedule with a sequence of time slots that are allocated to nodes for transmission.

The sequence shown in the figure focuses on the bootstrap of the global periodic synchronization and communications, i.e., the interval of time following start up until the regular periodic pattern of the network transmissions begins. The labels in the slots show the nodes to which they are allocated to. Note that we consider the slot duration σ to be constant and equal for all slots. This facilitates significantly the schedule construction and manipulation and it is a common feature of WSN protocols applied in practice such as WirelessHART.

Another aspect that simplifies the scheduling significantly, is the allocation of network slices, i.e., whole fractions of the TDMA schedule, to the communications in different hops (h_{max} being the maximum hop count, i.e., the depth of the cluster-tree). Thus, inside each slice, the slots scheduling is local and can be accomplished by hearing the transmissions of neighboring nodes. As we explain later in this section, we call the number of slices the *cadence-factor* ω.

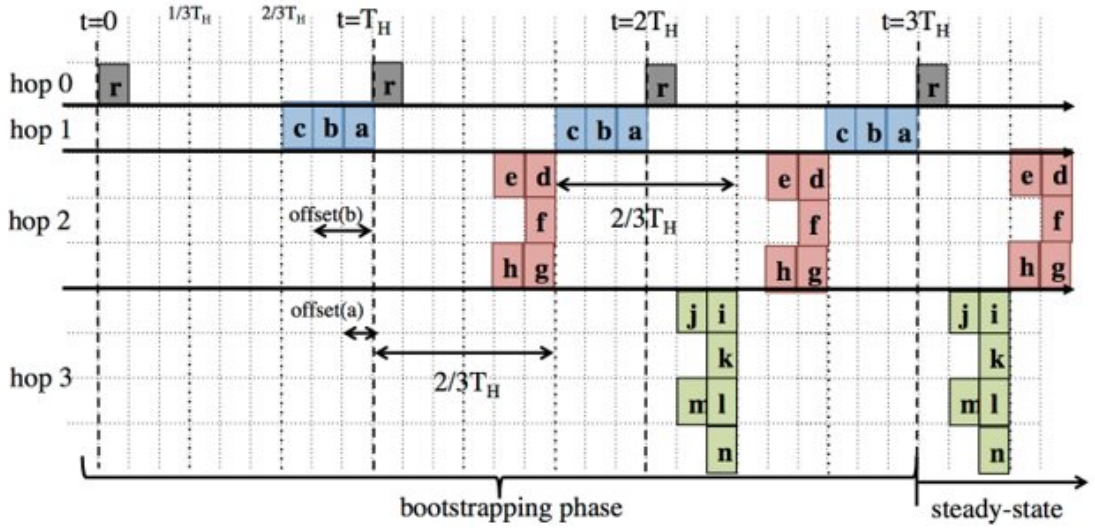

Figure 6.3: Timeline of transmission in NHS, from start up to steady periodic activity.

6.2.1.3 Network Synchronization and Scheduling

One interesting aspect of NHS is that global synchronization is achieved progressively, using the reception of messages as time references. In fact, in a multi-hop scenario, there is no single message that can reach all nodes and synchronize them together. Thus, NHS uses the transmissions

[2]Note that harmonizing periods may imply a certain level of resource over-provisioning, but this is a price to pay to cut the size of transmissions schedule and achieve minimum end-to-end delays. If different periods are needed, they must be integer multiples of the harmonic cycle.

of each cluster head, starting from the root node, to synchronize the nodes in the respective cluster, including the heads of the next clusters in hop count. Then, once receiving a message from the cluster head, all nodes in the cluster compute their transmission instants based on specific offsets that follow the order of their identifiers (ID).

This offset-based approach is then followed by all nodes in the network. For this purpose, each node transmits an NHS-tuple as a part of the packet header. The NHS-tuple, denoted λ, consists of η, the number of hops the transmitter is from the root, i.e., the hop count of the respective cluster, and ϕ, its offset counted in number of slots:

$$\lambda \equiv < \eta, \phi >$$

The example in Figure 6.3 considers $\omega = 3$, thus with the global harmonizing period T_H divided in three equal slices, one per hop, of duration $1/3T_H$. The root node r transmits at time $t = 0$ advertising the harmonizing period T_H and $\lambda =< 0,0 >$, synchronizing nodes *a, b* and *c*. However, these nodes do not transmit immediately, they compute a reference T_r to the end of the harmonizing period ($T_r = T_H$), also coinciding with the end of the 3^{rd} slice, just before the next root beacon transmission, and then compute offsets to the slots (T_s) allocated from that reference backwards by ID order. Therefore, their λ will be $< 1,1 >$, $< 1,2 >$ and $< 1,3 >$, respectively.

For all other levels in the network (hop count > 1), when receiving the parent packet, the nodes compute the T_r offset to the end of their slice, with respect to the start of the parent slot. This is shown in the following expression, where ϕ_p is the offset of the parent:

$$T_r = \frac{2}{3}T_H + \phi_p$$

The offsets of the slots in each cluster are again applied backwards in ID order. The slot s has offset $T_s = T_r - \phi$ For example, nodes *d* and *e* that are in the same cluster will have for their λ $< 2,1 >$ and $< 2,2 >$, respectively. In the next hop, nodes *j* and *i* in the same cluster will have λ equal to $< 3,1 >$ and $< 3,2 >$, respectively.

The factor $2/3$ that appears in the expression above results from the cadence-factor $\omega = 3$ and the policy of ensuring that the nodes in successive hops transmit in the slice that precedes the one of their parents, in this case by T_H/ω. Looking at the slots schedule in Figure 6.3, we can see that the sensor data collected by the nodes in any hop is propagated to the sink (root) in at most one harmonizing period (T_H). Conversely, the synchronization, which propagates in the opposite direction, takes approximately 3 (i.e., h_{max}) harmonic cycles to go from the root to the network leaves.

Note that we have considered just three hops for convenience of representation and explanation. However, NHS copes with an arbitrary depth of the cluster-tree. If the hops are created at the boundaries of the communication range, then it is safe to assume that communications in two clusters that are three hops away of each other do not mutually interfere and can reuse the channel, transmitting in parallel. Thus, if in the example above we had another layer of external clusters, it would be hop 4, which could reuse the same slice reserved for hop 1.

In other words, the protocol ensures at least ω-hop distance between clusters that transmit simultaneously in the network. Assigning slots to transmissions in a TDMA-based network is typically accomplished by applying distance-ω vertex coloring graph. To maximize the throughput, the problem is equivalent to choosing the minimum number of colors (Ramanathan, 1999). However, we are not aiming at maximum throughput, but at simplicity of deployment, low energy and low delay, instead. We achieve the required ω-hop distance by dividing each harmonizing period into ω equal slices. Nodes at consecutive hop-levels transmit only in non-overlapping slices. The number of slices, i.e., the cadence-factor, can be chosen to be greater or equal to 3.

Designing a protocol with a cadence-factor of less than three ($\omega < 3$) can result in collisions at the receivers. On the other hand, if $\omega > 3$, data from deeper hops can reach the root within one harmonizing period. However, given the fixed slot size, a larger ω may also imply a longer harmonizing period or a stronger limitation on the number of nodes per cluster. Therefore, the choice of ω provides a trade-off between the latency suffered by a packet to reach the root from a leaf-node and the maximum number of children a node can have.

In general, for slot scheduling purposes, the reference time T_r can be calculated with respect to ω as follows:

$$T_r = \frac{\omega - 1}{\omega} T_H + \phi_p$$

In steady state, the network operation is harmonized with respect to the cadence-factor (ω) and the harmonizing period (T_H). The expression above is used to allocate to the nodes in the current hop the slots in the last slice ($\omega - 1$) of the harmonizing period that starts with the slice of the parent node. Note that the next parent transmission occurs in the following slice. As discussed before, this approach creates a pipeline propagation of sensor data up the network to the root, improving the respective end-to-end latency.

In the general case of a network designed to operate with any value of ω and T_H we need to update the definition of λ, the NHS-tuple that we presented before and which was introduced in (Gupta et al., 2014). In this case we should include these parameters in the NHS-tuple so they are passed from cluster leaders to cluster nodes, allowing them to compute the offset of their slots properly.

$$\lambda \equiv < \eta, \phi, \omega, T_H >$$

6.2.1.4 Slots Allocations

Finally, concerning the slots allocation we need to consider the allocation order per cluster, which is based on the IDs of the nodes in the cluster as referred before, and whether they create interference (collisions) in the cluster leader. Looking again at both Figures 6.2 and 6.3 we can see that the slots for nodes inside the same cluster are always serialized (represented in horizontal lines). Conversely, clusters at the same depth may extend in different spatial directions that will keep

their transmissions out of the range of each other, as referred before. In such case, as shown in Figure 6.3, their slots can be allocated in parallel (represented vertically).

To enforce unambiguous allocation of slots to nodes, avoiding collisions caused by hidden-nodes, all cluster heads (parent nodes) piggyback in their transmissions (in the NHS header) the list of nodes from whom they successfully received packets during the previous slice. This way, they share the list of all nodes that must use exclusive slots. With this list the nodes compute their slots offsets. The simplest approach is to have the parent node sorting the list according to the nodes IDs and the nodes use the offset of their ID in the list as slot offset.

Different cluster heads will provide different lists to their nodes, which can be allocated the same set of slots, in parallel, as long as the lists are disjoint (this is the case in Figure 6.3). If there are overlapping lists, it means that some nodes receive from more than one parent node. In this case, each such node chooses the parent with the highest signal strength of the received packets with whom it stays logically connected. However, the offset of the slot needs to consider the different lists of the multiple parents. A reasonable criterion is to use the slot corresponding to the highest offset among all the lists. We call this a *forced offset* because it will have to be enforced in the other clusters where its offset would have been smaller.

To support forced offsets, all nodes include their offset in the NHS header of their packets and signal if it is forced. When a parent receives a packet from a node with a forced offset, it adjusts the schedule of its nodes to respect the forced offset. The easiest way is to shift the position of this node in the list (inserting idle slots) until its position matches the forced offset.

This phase of having each parent scheduling the nodes it hears in a compact sequential set of slots is called in (Gupta et al., 2014) *compressing the cluster schedule*. To bootstrap this scheduling it is necessary that cluster heads first receive from the nodes. In the same work, this problem is solved by proposing the use. initially, of absolute offsets that are globally matched to the nodes IDs. In this case, the first transmissions of all nodes would be using these absolute offsets, while from the following harmonic cycle onward, the nodes could already use the slots scheduled by each cluster head. A more efficient, and probably more practical, alternative to the absolute offsets would be to start transmissions with CSMA access arbitration on and switch it off after the slots scheduling is done.

The bootstrap process is also shown in Figure 6.3. Note that the final schedule for hop 1 would be conveyed in the second root message, for hop 2 in the second message of their cluster heads in hop 1, and for hop 3 in the second message of the respective cluster heads in hop 2. This means that at $t = 3T_H$ (or in general at $t = h_{max}T_H$) the slots schedule of the three (all) hops is defined.

The situation of having nodes that are heard by multiple cluster heads was left open in (Gupta et al., 2014), but it requires another harmonic cycle to be addressed in the bootstrap as referred above. In this situation, the nodes would keep the CSMA arbitration switched on for two harmonic cycles after receiving their first cluster head synchronization message. In the same example of Figure 6.3 the final schedule of all hops would be defined at $t = 6T_H$, only, or in general at $t = 2h_{max}T_H$.

Field	Description
ID	ID of the transmitter
Slot	Offset (ϕ), transmitter's slot backwards from end of slice
Parent	ID of transmitter's parent
Hopcnt	Hop count (η), Transmitter's Hop level
Cycle	Current cycle in number of harmonizing periods
N_child	Number of transmitter's child nodes
{Child_k}	Set of IDs of the N_child nodes
NHS Data	Data from all the deployed applications, delimited by application ID and length.

Table 6.1: Structure of the NHS packet.

6.2.2 NHS Preliminary Implementation

To test the feasibility of the protocol, we implemented the Network Harmonized Scheduling on the Contiki operating system replacing the Radio Duty-Cycling (RDC) and the Medium Access Control (MAC) layers of the Contiki network-stack. The core of the NHS implementation is a simple state-machine as shown in Figure 6.4.

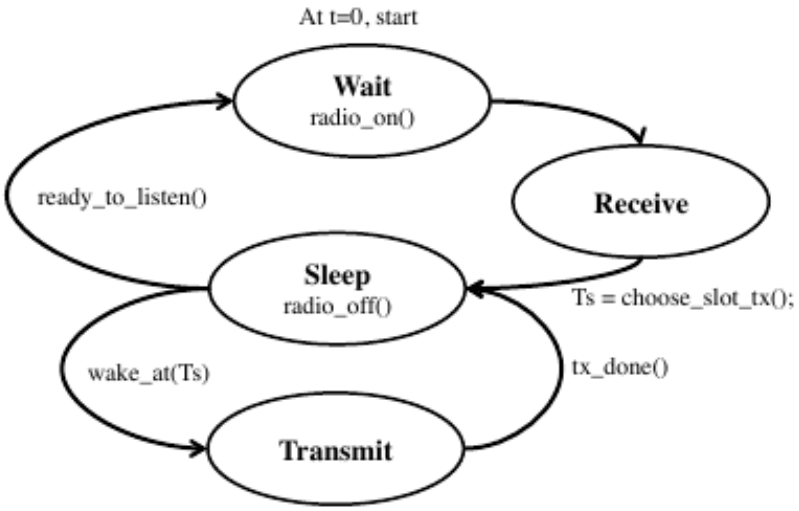

Figure 6.4: State machine showing the core of implementation of the NHS protocol at each node.

The state machine is the same for all the nodes except the root node, which is essentially a gateway with other networks. The NHS packet, structured as shown in Table 6.1, contains all the information needed to support the proper operation of the state machines in all nodes in a distributed fashion. It contains a header part with control fields and a data part, the *NHS Data* field, which carries all the flows of the applications currently deployed. A node may send multiple NHS packets if needed to support larger amounts of information. This is particularly relevant at higher layers of the cluster-tree that need to support the convergence of the data coming from all the sensors.

The implementation of the bootstrap phase was simplified, using initial offsets that were already similar to the steady state ones. We focused on the regular operation. At the beginning the node is in the network Wait state, waiting for a parent packet. This packet will allow defining the transmission slot offset T_s, by invoking the function *choose_slot_tx()*, as well as the offset of the start of transmissions of the respective child nodes $T_r - 2T_H/\omega$ (note that T_r is the offset to the end of the slice in which the node transmits). Then the node moves to the Sleep state. At $T_r - 2T_H/\omega$, the *ready_to_listen()* function expires and the node moves to the Wait state to receive its child nodes transmissions (next hop). This is done iterating between the Receive and Wait states (through Sleep).

Once the slice of the child nodes hop is over, at $T_r - T_H/\omega$, the node returns to the Sleep state. At T_s the *wake_at()* function expires and the node moves to the Transmit state to send its information (a batch with its locally generated information plus the information to be forwarded). When the transmission ends, the node moves again to the Sleep state until the harmonic period T_H expires and the node restarts the cycle, moving again to the Wait state to receive the next parent message.

One important detail that is hidden in the figure is the routing of information. This is done implicitly in the Receive state. Data coming both from the parent and the child nodes is aggregated and transmitted in a batch. It is the receiver that discards the information that is flowing in counter-direction. The parent discards its own information in the messages it receives and keeps just the information from the child nodes, while the child nodes discard all information except that coming from the parent. In the implementation reported here, nodes may add their own data, generated locally, to the communication batch and it will be taken as if it was coming from the child nodes. This implies that, currently, nodes can send their own information upwards, only,i.e., to the sink, which is consistent with typical WSN that do sensor data gathering.

The protocol was implemented on TMote Sky sensor nodes and with networks up to 10 nodes, with diverse topologies including line and cluster-tree (Gupta et al., 2014). The results not only validated the feasibility of the protocol, but they also confirmed that the duty-cycle needed for diverse sets of communication requirements was lower with NHS than with ideal TDMA, confirming the capacity of NHS to generate compact schedules, thus reducing overhead and energy.

6.2.3 Using NHS to Support CAP

Coming back to our aim of supporting CAP, we need to revisit the requirements that need to be met to achieve such capacity. In particular, we need to support multiple applications running on the network, bidirectional and node-to-node communication and dynamic topology, and these requirements may also change over time, i.e., the requirements themselves are dynamic. One particular aim of using NHS to support CAP is to provide some level of QoS control in the timing domain by means of adequate resource (network bandwidth) management. We will also address this issue at the end of this section.

6.2.3.1 Multiple Applications

NHS intrinsically supports multiple applications by separating applications and network and providing a data interface between both. Applications read and write data to this interface. Transmission and reception to and from the network is carried out by a specific management task that is part of the network stack and implements the state machine in Figure 6.4. This data interface naturally multiplexes the flows of multiple applications in the node communication channel. This separation allows keeping control of the transmission instants, according to network objectives, like compressing schedules and reducing duty cycle requirements, independently of the applications running in the nodes. However, the service provided to the applications naturally depends on the properties of the communication channel, namely the harmonizing period T_H, the cadence-factor ω and the slot width σ. Applications are expected to run recursively with periods that are larger than T_H, normally asynchronously with respect to the network. Thus, a random *access latency* with uniform distribution between virtually 0 and T_H can build up in the interface, adding to the end-to-end communication delay (from sensing to delivery in sink).

6.2.3.2 Bidirectional and Node-to-node Communication

As discussed previously, NHS supports communication in both directions, from the sink to the network leaves and vice-versa. However, the network scheduling that was adopted and described above is such that favors the upward flows, i.e., from the leaves to the sink, granting them expedite forwarding with a latency that is upper bounded by T_H as long as the network depth h_{max} is not more than ω. In general, the upward forwarding latency is bounded to $\lceil h_{max}/\omega \rceil T_H$. Accounting for the access latency referred in the previous section, the upward end-to-end delay upper bound d^{up} can be determined as:

$$d^{up} = T_H(1 + \lceil h_{max}/\omega \rceil)$$

On the other hand, the downward forwarding, from root to the network leaves, is significantly slower, taking approximately T_H per hop minus one. If we consider that there can be an access delay of T_H in the root caused by asynchronous application data generation, the downward end-to-end delay upper bound d^{down} can be determined as:

$$d^{down} = h_{max}T_H$$

This asymmetry is adequate to networks that are sensing-oriented, as most WSN. This is also adequate to Context-Aware Sensor Networks, in which the low latency of sensing can help in keeping track of contextual information. However, in WSAN, it could be the case that a certain network is used essentially to connect actuators. If this is the case, NHS can be easily reconfigured to switch the asymmetry to favour downward flows, instead. This can be accomplished simply by programming the slice allocated to hop n to occur immediately after the slice of hop $n-1$, instead

of immediately before as it is now. This would change the reference time to compute the slots offsets to:

$$T_r^a = \frac{1}{\omega} T_H - \phi_p$$

The slots would then be allocated from the beginning of the slice, using offset $T_s^a = T_r^a + \phi$, where ϕ would start from 0 onward. This change would essentially lead to switching the delay upper bounds shown above for the two direction of end-to-end communication.

Concerning node-to-node communication, this is not currently supported by NHS. In the current implementation, all nodes communicate with the sink/root, only. Thus, node-to-node communication would require a full upward and downward cycle, with a significant overhead. However, as it was clear in the examples provided in Chapter 4, CAP can take advantage of direct sensor to controller, or sensor to actuator, or even sensor to sensor communication. Fortunately, supporting direct node-to-node communication in NHS requires a relatively small change in the routing of data, namely the addition of routing tables in all nodes with the set of nodes that are accessible through each child node.

When receiving a packet, a node can know whether a give piece of information is meant to itself, thus moving it to internal buffers and removing it from the communication batch, or whether it is meant to a child node and keeps it in the batch, in the child nodes area, or the destination is unknown among all child nodes and the data is kept in the batch to be transmitted to the parent node. The use of node-to-node communication, however, has different end-to-end delay upper bounds than those referred above, since it may involve a small fraction of the network, only. For this reason, it also complicates the analysis of the network delays in general since this kind of communication affects differently each network link, thus requiring global information to support such analysis. Finally, the extra amount of memory to hold routing tables in the nodes might be prohibitive and disallow this feature.

6.2.3.3 Dynamic Topology

This is a fundamental requirement in Context-Aware Sensor Networks, since an important source of context changes arises from topology changes, e.g., caused by mobility or node failures. Unfortunately, this aspect has relatively low support in NHS. By design, NHS is more suitable and reliable for static and nomadic sensor deployments than mobile and intermittent networks.

Nevertheless, NHS can still cope with certain changes in the network topology. For example, removal of nodes can be accomplished in a fairly simple way. If a parent node does not receive from a child node for a pre-configured number of communication cycles, it may assume the child and corresponding branch has left the network, thus freeing the corresponding slots and possibly compressing the resulting schedule in that cluster. On the other direction, if a node does not receive a packet from its parent for a preset number of cycles, it chooses a different parent within its neighborhood, if available, and keeping the same slots. Again, a compression of the cluster schedule may be required after the reconfiguration.

To avoid removing a whole branch and reattach a part of it immediately after, the number of cycles to detect the loss of a parent should be smaller than to detect the loss of a child. Thus, the loss of an intermediate node in the tree would be first detected from below, the child nodes side, triggering a re-connection of the associated branches. In the case of loss of parent node and lack of alternative parent, it is possible that there is still another path that allows recovering the cluster-tree. However, this would need building the whole cluster-tree from scratch, launching the bootstrap process.

The situation is somewhat more complicated when admitting new nodes due to the need to provide a mechanism to allow new nodes to announce themselves or explicitly request integration. As proposed in (Gupta et al., 2014) one possible solution is to allocate a few slots for asynchronous announcements with CSMA access control enabled. New nodes can hear for a certain period to acquire fundamental network parameters, such as T_H, ω, h_{max} and the potential parents in range, and synchronize with the network schedule. Then, the new node could pick one parent node and send a join request in one of the available CSMA slots. The parent would confirm in the following cycle and update the cluster slots schedule already including the new node, or could reject or ignore the new node that could, then, try another available parent node.

Note that the solution above may imply an extra complexity in the presence of overlapping neighbor clusters due to slot sharing and the need to consider joint slot allocation, including forced offsets (see slots allocation in the previous section). If several clusters need to update their schedules in the course of a joining process, the updates should be consistent and simultaneous (same cycle) across all clusters at the involved hop. This can be achieved with a two cycle process similar to the one proposed for the bootstrap process. In a first cycle the different parents that heard the new node request publish updated cluster schedules but with a flag telling the cluster members to keep the previous schedule. Parent nodes hear each other and determine conflicting slot assignments, solving them with forced offsets. In the following cycle all cluster schedules are updated accordingly and can be applied to the cluster nodes.

6.2.3.4 Dynamic Requirements

Concerning the application requirements, NHS already provides support for dynamic definition of certain parameters. For example, the applications working rate, or recurrence period can be changed at run-time without impact on the network given the separation between applications and network enforced by NHS. Such a change would have implications on the utilization of the slots of the respective nodes, only. In the same way, applications can be launched and removed without impact on the network, but again on the utilization of the slots of the involved nodes, only. This is very convenient since context-awareness frequency leads to changes in the set of running applications or in their frequency of operation.

In the course of these adaptations, it may happen that the capacity of a slot is not enough for the new requirements. In this case, NHS can support the dynamic addition of slots with a small modification. Adding slots can be done in a similar way to adding a new node as described in the previous section. It is possible to use a two-cycle protocol in which the first cycle is used to

communicate the slot request to the respective cluster head and to all other cluster heads that hear it. The cluster heads will independently compute and then transmit proposals for cluster schedules with the new slot, which are not applied, yet. In the second cycle, all cluster heads hear each other and agree on possible forced offsets for the slots in their schedules and build new compressed schedules accordingly. These new schedules are announced and enforced in the following cluster head messages.

Beyond the application requirements, there may be system requirements that may also change, such as the harmonizing period or the cadence factor. NHS directly supports some of these changes by having the root node disseminating and enforcing these values through the network, hop by hop until the leaf nodes. When the new configuration can co-exist with the current configuration, then NHS can apply the changes dynamically without any modification. For example, increasing the harmonizing period will cause the next hop to transmit later while following hops will still transmit earlier according to the previous period, thus without collisions. The same happens when increasing the cadence-factor, which reduces the length of the hop slices, thus the next hop will transmit less, while the following hops still have current slice width. The situation is more complex when decreasing the harmonizing period or decreasing the cadence-factor, because during the propagation of the new values there will be moments in which there will be collisions due to overlapping communications caused by hops that were already updated and other that are still with the previous configuration. Fortunately, this can also be easily sorted out by first communicating the new values throughout the whole network and deferring the change in configuration to a later cycle, when all hops have already received the updated values.

6.2.3.5 Fitting NHS in the CAP Architecture

Finally, when addressing the use of NHS in the scope of CAP it is necessary to see how NHS could be integrated in the middleware architecture. NHS is a scheduling method that manages an associated WS(A)N. In terms of CAP, this network can be just a subset of the whole network managed by the middleware, connected through the root node that works as gateway. Nevertheless, CAP's Application Manager may support the design of applications for the WSAN nodes. As referred earlier in this section, NHS currently supports upward flows to the sink/root and downward flows from the root, only. Thus it is well suited to include sensors and actuators, only, that communicate with controllers that rest outside the WSAN.

CAP's Resource Manager can also keep track of the current WSAN topology by talking to the respective root. When instantiating applications, it can also keep track of the utilization of the nodes as well as their WSAN slots. Moreover, it can now interact with the WSAN root to compare possible timing requirements of the applications with the guarantees that NHS can provide. In this way, the Resource Manager can inform applications that use the WSAN when NHS cannot meet their requirements. On the other hand, the Resource Manager can also trigger the reconfiguration of the WSAN itself, such as the harmonizing period or cadence-factor as discussed above. A concept architecture of this integration is shown in Figure 6.5.

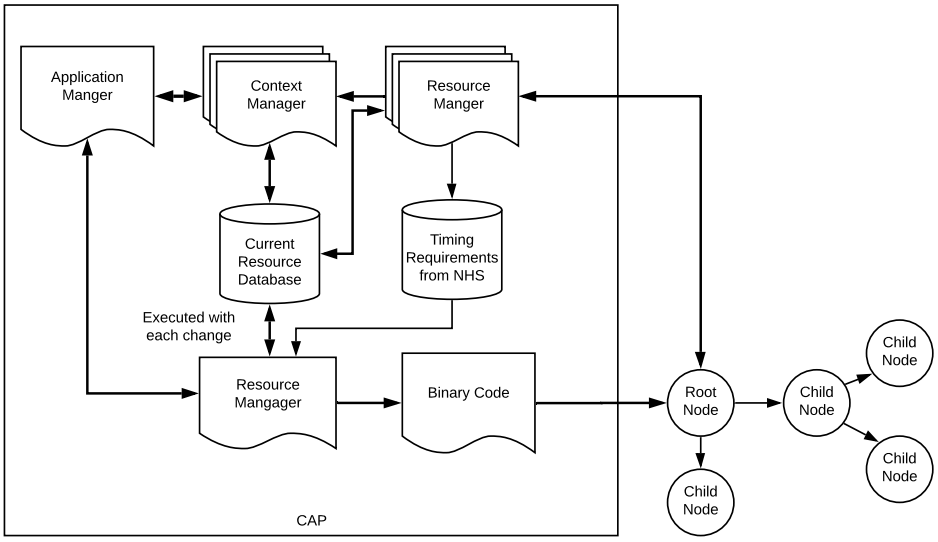

Figure 6.5: Concept architecture of CAP enhanced by NHS

Finally, the sensors in the WSAN can feed CAP's Context Manager to drive contextual updates and trigger associated reconfigurations. Some of these reconfigurations may include changing the applications that currently use the WSAN itself, as discussed earlier.

In terms of topology, by inserting the WSAN in a larger network under CAP, it is reasonable to expect that nodes with higher mobility will be outside the WSAN, while this subnetwork can be kept just for a set of nodes embedded in the environment, e.g., sensors and actuators in a building, which do not change often. This is in-line with the target WSN for which NHS was designed and allows using NHS directly in the scope of CAP, even with its limited support to dynamic topology.

6.3 Summary

In this chapter, we have discussed the interest in using resource allocation in the network to provide some level of QoS guarantees in terms of communication latency in the scope of WSN and generally in WSAN. In this scope, we made use of a scheduling method for multi-hop cluster-tree WS(A)N, namely NHS, which was partially developed in the scope of this thesis.

Overall, the NHS protocol is distributed by design and maintains very little state. The primary benefit from this approach is that the transmissions are harmonized around periodic boundaries and packets do not suffer from contention. The radios on the nodes only need to be turned on in a periodic manner for a short time-span, which considerably reduces the radio switching overhead. The proposed protocol reaches a good compromise between simplicity of deployment (favored by its distributed clockless character) and timeliness (favored by its TDMA-based compressing slot scheduling).

Most importantly, NHS allows decoupling the timing behavior of applications running in the nodes from the communications in the network, which enables improving the network timing

behavior with benefits for latency, reliability and energy consumption, while supporting flexibility at the applications level.

Finally, the chapter ends with a discussion of how NHS can be used in the scope of CAP. This discussion left it clear that NHS can be readily applied, in spite of some limitations in what concerns its flexibility. The whole WSAN managed with NHS can be incorporated in the larger network running CAP, connecting mobile nodes and controllers in general to more stable sensors and actuators embedded in the environment. It is also reasonable to assume that CAP can actually integrate several WS(A)Ns together using different non-overlapping channels, all managed by its Resource Administrator, thus improving the gathering of sensing information and, by that means, the perception of contextual information and the support of the Context Manager towards more effective CASE.

Along this line, we can envisage the use of a separate cluster-tree to connect sensors, only, a true WSN, with NHS configured to provide low-latency upward sensor readings. Then, another cluster-tree could be used to connect actuators exclusively, with NHS configured to provide low-latency downward actuation commands. This would allow implementing control loops with optimal low sampling-to-actuation delay, consequently enabling performance improvements of global feedback control.

Chapter 7

Conclusion and Future Work

In this chapter we revisit the thesis statement proposed in the Introduction and the work developed towards supporting that statement. We first comment on how our contributions and how they provide the desired support for CASE, i.e., Context-Aware Sensor Networks. This allows us to validate our thesis. We then end this chapter presenting some lines for future work.

7.1 Summary of Contributions

In the Introduction we stated the following contributions of this work, which emerged while researching to support our thesis and which are the following:

1. *CAP - Context-Aware Programming*, a middleware that allows building CASE - Context-Aware SEnsor networks with high-level programming tools suitable for non-experts.

2. *AdaptC*, a set of basic adaptation policies that programmers can use to write context-aware applications.

3. *mT-Res*, a proof of feasibility of context-awareness in WSN that consisted in adding features to an existing macroprogramming approach, namely T-Res.

4. *NHS - Network-Harmonized Scheduling applied to CAP*, a scheduling technique, developed in collaboration within another work, that manages the communications of multiple applications in a WSN efficiently and timely.

The two first contributions, namely CAP and AdaptC, are the core contributions of this thesis, putting forward novel concepts to provide Context-Aware Sensor Networks that can be easily developed by non-expert users. The remaining two contributions are complementary works that were either needed to developed the previous ones (mT-Res) or that constitute an enhancement to them (NHS).

7.1.1 CAP - Context-Aware Programming

This is a novel middleware that offers native support to *Mobility, Modularity*, and *Abstraction*, and with these design features it allows developing Context-Aware Sensor Networks. This middleware is the topic of Chapter 4, where we propose the concept and an architectural plan. The middleware relies on multiple tools and technologies, particularly CoAP instructions in Python scripts. We also used the Django project with the help of some existing libraries of *node.js* to support the applications design process. A preliminary validation was carried out on both Contiki-OS and TinyOS, and several practical experiments were carried out on a deployment in the IoT-Lab. These results showed the resilience of CAP, having persisted in constant operation for over 15 days, constantly adapting to forced contextual changes.

7.1.2 AdaptC - Adaptation Policies

This is a novel way of structuring the development of applications that require frequent adaptation. With these adaptation policies, presented in Chapter 5, the users can write adaptable code that can change the application logic based on the context. These policies are part of the Context Manager and allow activating/deactivating contexts previously defined by the users and are then applied by the middleware on the set of running applications. We demonstrate the implementation of the adaptation policies and also showcase their benefits in terms of effort required by the users to write such adaptable code.

7.1.3 mT-Res - Macroprogramming with Mobility

This work can be seen as a first step in the development of CAP. It is described in Chapter 3 and consisted in adding the needed architectural elements to a well-known Macroprogramming solution for IPv6-based WSN, namely T-Res, to support *Mobility*. This was achieved with Python-based scripts. This work validated the possibility of providing context-awareness by extending an existing WSN programming approach. However, given that several other works built on T-Res in similar ways as we did, despite with other purposes, we consider this a secondary contribution.

7.1.4 NHS - Network Harmonized Scheduling and CAP

In our early work, we contributed to the development of a scheduling technique to manage the execution of multiple applications in a WSN by scheduling the associated traffic in a way that reduces communication latency. However, of interest to this thesis is the application of NHS in the scope of CAP as a means to achieve network QoS support, particularly in what concerns low-latency communications. Thus, the main contribution is the discussion of how NHS can be deployed within CAP. However, in spite of the extended discussion and the experimental validation of both CAP and NHS separately, their joint operation was not validated in practice. For this reason we also consider this a secondary contribution.

7.2 Validating the Thesis

We believe that the contributions we just summarised directly validate the thesis. We claimed that

by providing i) programming models with mobility, modularity, and abstraction features; ii) a resource manager that tracks computing resources available and their capabilities; and iii) a context manager that tracks collected data to activate/deactivate contexts then a WSN middleware can automatically adapt the execution of applications to changes in contexts while enabling high-level applications programming with efficient operation. This middleware is a cornerstone for the wide adoption of context-aware sensor networks (CASE).

In fact, we have shown experimentally that our CAP middleware leverages the design features in i) and the architectural elements in ii) and iii) to provide **automatic adaptation** of WSN applications as reaction to contextual changes, while offering an application programming interface and adaptation policies that are easy to use by non-expert users. Thus, we strongly believe that CAP is an adequate tool to build and promote the adoption of CASE.

7.3 Future Work

With Mobility, Modularity, and Abstraction as essential features available, we could focus on improving **interoperability** and support more platforms, computing and communications, to make Context-Aware Programming sustainable for WSNs. CAP was implemented on two programming platforms popular in WSNs, namely Contiki-OS and TinyOS. While TinyOS has not been updated for a while, Contiki-OS remains an up-to-date and reliable solution, majorly due to its expansive library. It would be interesting, though, to study the problems associated to supporting more hardware nodes and networking protocols and verify the capacity to handle full heterogeneity.

In addition, CAP allows us to target some key WSN problems with a fresh perspective. One of these problems is resource allocation. WSNs have always been a resource-oriented paradigm due to their dependence on energy. In a typical WSN, if a node runs out of energy the application running on that node is interrupted alongside that node. Either a new node needs to be deployed to replace the older node or the whole WSN application needs to be re-written. With CAP, there are alternative ways to deal with such problems. A **smart Resource Allocation Manager** could be designed to anticipate context changes such as low energy levels and make changes sufficiently ahead of time before any WSN node loses power completely.

Another key problem to consider is the **conflict** between multiple **applications and contexts**. Multiple contexts may require the same resources at the same time and, to meet demands, some applications may take priority. To resolve these conflicts a certain algorithm can be designed based on features associated with the application such as critical importance to the user, impact on the performance of WSN nodes and other QoS parameters. Machine Learning could potentially be used to learn about regular WSN operation and make decisions on behalf of the users to sort such conflicts.

When **multiple users** are using the same WSN nodes in different contexts, the **privacy of data** may become a challenge. Especially if the nodes are connected to the Internet or any other external service. It would be interesting to look at where the WSN data must be saved and for how long it is saved and carry out a full security risks analysis together with proposing mechanisms to mitigate the main risks.

Appendix A

Brief Setup Guide for the IoT-Lab Deployment

This appendix introduces a user with the requirements to run Context Aware Programming (CAP) in the IoT-Lab. This also provides hints on applying CAP in other execution environments. This guide was prepared with certain assumptions, the primary one being that the user is knowledgeable in Contiki-OS and related popular WSN tools.

A.1 Software

Following is the software needed to use CAP:

- Python and all its dependencies: https://www.python.org/downloads/
- CLI tools: https://www.iot-lab.info/docs/tools/cli/
- Txthings for CoAP: https://github.com/mwasilak/txThings
- Contiki-OS: https://github.com/iot-lab/contiki
- CAP Source Code: https://bitbucket.org/shashankgaur_/CAP_V2
- Testbed Webportal: https://github.com/iot-lab/testbed-webportal

Instruction for installing each software are provided in their own pages, please make sure you follow those according to the version of the software you install.

A.2 Hardware

Currently CAP works on following hardware:

- TelosB Motes
- MicaZ Motes

- IoT-lab Testbad Boards: www.iot-lab.info/docs/boards/iot-lab-m3/

A.3 Setup Step-by-Step

1. Concerning the IoT-Lab, to start using the testbed you must follow all the steps mentioned at the testbed website: https://www.iot-lab.info/docs/getting-started
2. Once you have setup your account, please upload CAP scripts to your workspace using the CLI tools by following the command:
   iotlab-experiment submit [...] –site-association <site>[,<site>],script=/CAP_V2/CAP.py
3. Now select the boards you would like to use in the testbed. At that point, there are two alternatives available to you to select nodes, by properties or by ID. More info on this is available here:https://www.iot-lab.info/docs/getting-started/resources/
4. Once you have selected resources and your boards, you must schedule the experiment using the CLI tools. This is explained here: https://www.iot-lab.info/docs/tools/run-script/

References

T. Abdelzaher, B. Blum, Q. Cao, Y. Chen, D. Evans, J. George, S. George, L. Gu, T. He, S. Krishnamurthy, L. Luo, S. Son, J. Stankovic, R. Stoleru, and A. Wood. Envirotrack: towards an environmental computing paradigm for distributed sensor networks. In *24th International Conference on Distributed Computing Systems, 2004. Proceedings.*, pages 582–589, 2004. doi: 10.1109/ICDCS.2004.1281625.

Pooyan Abouzar, David G Michelson, and Maziyar Hamdi. Rssi-based distributed self-localization for wireless sensor networks used in precision agriculture. *IEEE Transactions on Wireless Communications*, 15(10):6638–6650, 2016.

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 *International symposium on handheld and ubiquitous computing*, pages 304–307. Springer, 1999.

Traian E Abrudan, Zhuoling Xiao, Andrew Markham, and Niki Trigoni. Imu-aided magneto-inductive localization. April 2014.

Mikhail Afanasov, Luca Mottola, and Carlo Ghezzi. Context-oriented programming for adaptive wireless sensor network software. In *2014 IEEE International Conference on Distributed Computing in Sensor Systems*, pages 233–240. IEEE, 2014.

Mikhail Afanasov, Luca Mottola, and Carlo Ghezzi. Software adaptation in wireless sensor networks. *ACM Trans. Auton. Adapt. Syst.*, 12(4), January 2018. ISSN 1556-4665. doi: 10.1145/3145453. URL https://doi.org/10.1145/3145453.

Petteri Alahuhta, Henri Löthman, Heli Helaakoski, Arto Koskela, and Juha Röning. Experiences in developing mobile applications using the apricot agent platform. *Personal Ubiquitous Comput.*, 11(1):1–10, October 2006. ISSN 1617-4909. doi: 10.1007/s00779-005-0058-z. URL https://doi.org/10.1007/s00779-005-0058-z.

Abrar Alajlan and Khaled Elleithy. Programming models for wireless sensor networks: Status, taxonomy, challenges, and future directions. *International Journal of Scientific and Engineering Research - IJSER*, 6(5), 2015. ISSN 2229-5518.

Daniele Alessandrelli, Matteo Petraccay, and Paolo Pagano. T-res: Enabling reconfigurable in-network processing in iot-based wsns. In *2013 IEEE International conference on distributed computing in sensor systems*, pages 337–344. IEEE, 2013.

Hitha Alex, Mohan Kumar, and Behrooz Shirazi. Midfusion: An adaptive middleware for information fusion in sensor network applications. *Information Fusion*, 9(3):332–343, 2008. ISSN 1566-2535. doi: https://doi.org/10.1016/j.inffus.2005.05.007. URL https://www.sciencedirect.com/science/article/pii/S1566253505000679. Special Issue on Distributed Sensor Networks.

ZigBee Alliance. Zigbee specification. *ZigBee Document 053474r06, Version*, 1, 2006.

Isaac Amundson and Xenofon D Koutsoukos. A survey on localization for mobile wireless sensor networks. pages 235–254, 2009.

Tomoyuki Aotani and Gary T Leavens. Towards modular reasoning for context-oriented programs. In *Proceedings of the 18th Workshop on Formal Techniques for Java-like Programs*, page 8. ACM, 2016.

Tomoyuki Aotani, Tetsuo Kamina, and Hidehiko Masuhara. Featherweight eventcj: a core calculus for a context-oriented language with event-based per-instance layer transition. In *Proceedings of the 3rd International Workshop on Context-Oriented Programming*, pages 1–7, 2011.

Malte Appeltauer, Robert Hirschfeld, Hidehiko Masuhara, Michael Haupt, and Kazunori Kawauchi. Event-specific software composition in context-oriented programming. In *International Conference on Software Composition*, pages 50–65. Springer, 2010.

Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.

A. Azzara, D. Alessandrelli, S. Bocchino, M. Petracca, and P. Pagano. Pyot, a macroprogramming framework for the internet of things. In *Industrial Embedded Systems (SIES), 2014 9th IEEE International Symposium on*, pages 96–103, June 2014. doi: 10.1109/SIES.2014.6871193.

Amol Bakshi, Viktor K Prasanna, Jim Reich, and Daniel Larner. The abstract task graph: a methodology for architecture-independent programming of networked sensor systems. In *Proceedings of the 2005 workshop on End-to-end, sense-and-respond systems, applications and services*, pages 19–24, 2005.

Paolo Bellavista, Antonio Corradi, Rebecca Montanari, and Cesare Stefanelli. Context-aware middleware for resource management in the wireless internet. *IEEE Trans. Softw. Eng.*, 29 (12):1086–1099, December 2003. ISSN 0098-5589. doi: 10.1109/TSE.2003.1265523. URL https://doi.org/10.1109/TSE.2003.1265523.

Fehmi Ben Abdesslem, Andrew Phillips, and Tristan Henderson. Less is more: Energy-efficient mobile sensing with senseless. In *Proceedings of the 1st ACM Workshop on Networking, Systems, and Applications for Mobile Handhelds*, MobiHeld '09, pages 61–62. ACM, 2009.

Bernhard Firner, Robert S. Moore, Richard Howard, Richard P. Martin, and Yanyong Zhang. Smart buildings, sensor networks, and the internet of things. In *Proceedings of the 9th International Conference on Embedded Networked Sensor Systems, SenSys 2011, Proceedings.*, November 2011. doi: 10.1145/2070942.2070978.

Arne Bestmann and Rönne Reimann. Easypoint–indoor localization and navigation low cost, reliable and accurate. April 2014.

Sangeeta Bhattacharya, Abusayeed Saifullah, Chenyang Lu, and Gruia-Catalin Roman. Multi-application deployment in shared sensor networks based on quality of monitoring. pages 259–268, 2010.

Stefano Bocchino, Szymon Fedor, and Matteo Petracca. Pyfuns: A python framework for ubiquitous networked sensors. In *European Conference on Wireless Sensor Networks*, pages 1–18. Springer, 2015.

Alan Borning. Evaluating programming languages, 2002. URL https://courses.cs.washington.edu/courses/cse341/02sp/concepts/evaluating-languages.html.

Peter J Brown, John D Bovey, and Xian Chen. Context-aware applications: from the laboratory to the marketplace. *IEEE personal communications*, 4(5):58–64, 1997.

Michael Buettner, Gary V Yee, Eric Anderson, and Richard Han. X-mac: a short preamble mac protocol for duty-cycled wireless sensor networks. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 307–320, 2006.

Aaron Carroll and Gernot Heiser. An analysis of power consumption in a smartphone. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 21–21. USENIX Association, 2010.

A. T. S. Chan and Siu-Nam Chuang. Mobipads: a reflective middleware for context-aware mobile computing. *IEEE Transactions on Software Engineering*, 29(12):1072–1085, 2003. doi: 10.1109/TSE.2003.1265522.

Pascal Costanza, Robert Hirschfeld, and Wolfgang De Meuter. Efficient layer activation for switching context-dependent behavior. In *Joint Modular Languages Conference*, pages 84–103. Springer, 2006.

Carlo Curino, Matteo Giani, Marco Giorgetta, Alessandro Giusti, Amy L. Murphy, and Gian Pietro Picco. Mobile data collection in sensor networks: The tinylime middleware. *Pervasive and Mobile Computing*, 1(4):446–469, 2005. ISSN 1574-1192. doi: https://doi.org/10.1016/j.pmcj.2005.08.003. URL https://www.sciencedirect.com/science/article/pii/S1574119205000453. Special Issue on PerCom 2005.

Julius Degesys, Ian Rose, Ankit Patel, and Radhika Nagpal. Desync: self-organizing desynchronization and tdma on wireless sensor networks. In *Proceedings of the 6th international conference on Information processing in sensor networks*, pages 11–20, 2007.

Linda G DeMichiel and Richard P Gabriel. The common lisp object system: An overview. In *European Conference on Object-Oriented Programming*, pages 151–170. Springer, 1987.

D. Deniz, F. Barranco, J. Isern, and E. Ros. Reconfigurable cyber-physical system for lifestyle video-monitoring via deep learning. In *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, volume 1, pages 1705–1712, 2020. doi: 10.1109/ETFA46521.2020.9211910.

Amol Deshpande, Carlos Guestrin, and Samuel Madden. Resource-aware wireless sensor-actuator networks. *IEEE Data Eng. Bull.*, 28(1):40–47, 2005.

Adam Dunkels. The official git repository for contiki, the open source os for the internet of things, 2003. URL https://github.com/contiki-os/contiki.

Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki-a lightweight and flexible operating system for tiny networked sensors. In *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, pages 455–462. IEEE, 2004.

M.A. El Khaddar, M. Chraibi, H. Harroud, M. Boulmalf, M. Elkoutbi, and A. Maach. A policy-based middleware for context-aware pervasive computing. *International Journal*

of Pervasive Computing and Communications, 11(1):43–68, 2015. doi: doi.org/10.1108/IJPCC-07-2014-0039.

Milan Erdelj, Nathalie Mitton, Enrico Natalizio, et al. Applications of industrial wireless sensor networks. *Industrial Wireless Sensor Networks: Applications, Protocols, and Standards*, pages 1–22, 2013.

Anand Eswaran, Anthony Rowe, and Raj Rajkumar. Nano-rk: an energy-aware resource-centric rtos for sensor networks. pages 10–pp, 2005.

Claudio M. De Farias, Wei Li, Flávia C. Delicato, Luci Pirmez, Albert Y. Zomaya, Paulo F. Pires, and José N. De Souza. A systematic review of shared sensor networks. *ACM Comput. Surv.*, 48 (4), February 2016. ISSN 0360-0300. doi: 10.1145/2851510. URL https://doi.org/10.1145/2851510.

Federico Ferrari, Marco Zimmerling, Lothar Thiele, and Olga Saukh. Efficient network flooding and time synchronization with glossy. In *Proceedings of the 10th ACM/IEEE International Conference on Information Processing in Sensor Networks*, pages 73–84. IEEE, 2011.

Federico Ferrari, Marco Zimmerling, Luca Mottola, and Lothar Thiele. Low-power wireless bus. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*, pages 1–14, 2012.

Carmelo di Franco, Enrico Bini, Mauro Marinoni, and Giorgio C Buttazzo. Multidimensional scaling localization with anchors. In *2017 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, pages 49–54. IEEE, April 2017.

Yasuhiro Fukuju, Masateru Minami, Hiroyuki Morikawa, and Tomonori Aoyama. Dolphin: An autonomous indoor positioning system in ubiquitous computing environment. pages 53–56, 2003.

S. Gaglio, G. Lo Re, L. Giuliana, G. Martorella, D. Peri, and A. Montalto. Interoperable real-time symbolic programming for smart environments. In *2019 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 309–316, 2019. doi: 10.1109/SMARTCOMP.2019.00067.

Shashank Gaur. T-res extension, 2015. URL https://bitbucket.org/shashankgaur_/tres_extension.

David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesc language: A holistic approach to networked embedded systems. *SIGPLAN Not.*, 38(5): 1–11, May 2003. ISSN 0362-1340. doi: 10.1145/780822.781133. URL http://doi.acm.org/10.1145/780822.781133.

Carlo Ghezzi, Matteo Pradella, and Guido Salvaneschi. Programming language support to context-aware adaptation: a case-study with erlang. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 59–68, 2010a.

Carlo Ghezzi, Matteo Pradella, and Guido Salvaneschi. Programming language support to context-aware adaptation: a case-study with erlang. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 59–68. ACM, 2010b.

Nam K. Giang, Rodger Lea, Michael Blackstock, and Victor C. M. Leung. On building smart city iot applications: A coordination-based perspective. In *Proceedings of the 2nd International Workshop on Smart*, SmartCities '16, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450346672. doi: 10.1145/3009912.3009919. URL https://doi.org/10.1145/3009912.3009919.

Omprakash Gnawali, Rodrigo Fonseca, Kyle Jamieson, David Moss, and Philip Levis. Collection tree protocol. In *Proceedings of the 7th ACM conference on embedded networked sensor systems*, pages 1–14. ACM, 2009.

Sebastian Andres Gonzalez Montesinos, Nicolas Cardozo Alvarez, Kim Mens, Alfredo Jaime Cadiz Rodriguez, Jean-Christophe Libbrecht, and Julien Goffaux. Subjective-c: Bringing context to mobile platform programming. In *International software language engineering conference*, 2011.

Ramakrishna Gummadi, Omprakash Gnawali, and Ramesh Govindan. Macro-programming wireless sensor networks using kairos. In *Proceedings of the First IEEE International Conference on Distributed Computing in Sensor Systems*, DCOSS'05, 2005.

Vikram Gupta, Junsung Kim, Aditi Pandya, Karthik Lakshmanan, Ragunathan Rajkumar, and Eduardo Tovar. Nano-cf: A coordination framework for macro-programming in wireless sensor networks. In *2011 8th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks*, pages 467–475. IEEE, 2011.

Vikram Gupta, Nuno Pereira, Shashank Gaur, Eduardo Tovar, and Ragunathan Rajkumar. Network-harmonized scheduling for multi-application sensor networks. pages 1–10, 2014.

Morten Tranberg Hansen, Raja Jurdak, and Branislav Kusy. Unified broadcast in sensor networks. In *Proceedings of the 10th ACM/IEEE International Conference on Information Processing in Sensor Networks*, pages 306–317. IEEE, 2011.

G. Hatzivasilis, I. Askoxylakis, G. Alexandris, D. Anicic, A. Bröring, V. Kulkarni, K. Fysarakis, and G. Spanoudakis. The interoperability of things: Interoperable solutions as an enabler for iot and web 3.0. In *2018 IEEE 23rd International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*, pages 1–7, 2018. doi: 10.1109/CAMAD.2018.8514952.

Ted Herman and Sébastien Tixeuil. A distributed tdma slot assignment algorithm for wireless sensor networks. In *International Symposium on Algorithms and Experiments for Sensor Systems, Wireless Networks and Distributed Robotics*, pages 45–58. Springer, 2004.

K. Heurtefeux and F. Valois. Is rssi a good choice for localization in wireless sensor network? In *2012 IEEE 26th International Conference on Advanced Information Networking and Applications*, pages 732–739, 2012.

Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3), 2008a.

Robert Hirschfeld, Pascal Costanza, and Oscar Marius Nierstrasz. Context-oriented programming. *Journal of Object technology*, 7(3):125–151, 2008b.

Jong-yi Hong, Eui-ho Suh, and Sung-Jin Kim. Context-aware systems: A literature review and classification. *Expert Systems with applications*, 36(4):8509–8522, 2009.

Xin Hu, Rahav Dor, Steven Bosch, Anita Khoong, Jing Li, Susan Stark, and Chenyang Lu. Challenges in studying falls of community-dwelling older adults in the real world. In *Smart Computing (SMARTCOMP), 2017 IEEE International Conference on*, pages 1–7. IEEE, 2017.

Stepan Ivanov, Kriti Bhargava, and William Donnelly. Precision farming: Sensor analytics. *IEEE Intelligent systems*, 30(4):76–80, 2015.

Maissa Ben Jamâa, Anis Koubâa, and Yasir Kayani. Easyloc: Rss-based localization made easy. *Procedia Computer Science*, 10:1127–1133, 2012.

Junghyun Jun, Yu Gu, Long Cheng, Banghui Lu, Jun Sun, Ting Zhu, and Jianwei Niu. Social-loc: Improving indoor localization with social sensing. pages 1–14, 2013.

Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. Eventcj: a context-oriented programming language with declarative event-based context transition. In *Proceedings of the tenth international conference on Aspect-oriented software development*, pages 253–264. ACM, 2011.

I. Khan, F. Belqasmi, R. Glitho, N. Crespi, M. Morrow, and P. Polakos. Wireless sensor network virtualization: A survey. *IEEE Communications Surveys Tutorials*, 18(1):553–576, 2016. doi: 10.1109/COMST.2015.2412971.

Joel Koshy and Raju Pandey. Vmstar: Synthesizing scalable runtime environments for sensor networks. SenSys '05, page 243–254, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 159593054X. doi: 10.1145/1098918.1098945. URL https://doi.org/10.1145/1098918.1098945.

Matthias Kovatsch. Demo abstract: Human–coap interaction with copper. In *Proceedings of the 7th IEEE International Conference on Distributed Computing in Sensor Systems*, 2011.

Ilias Leontiadis, Christos Efstratiou, Cecilia Mascolo, and Jon Crowcroft. Senshare: transforming sensor networks into multi-application sensing infrastructures. pages 65–81, 2012.

Philip Levis and David Culler. Mate: A tiny virtual machine for sensor networks. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, pages 85—-95. ACM, 2002.

Philip Levis, Neil Patel, David Culler, and Scott Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proc. of the 1st USENIX/ACM Symp. on Networked Systems Design and Implementation*, volume 25. USENIX/ACM, 2004.

Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, et al. Tinyos: An operating system for sensor networks. In *Ambient intelligence*, pages 115–148. Springer, 2005.

X. Li, M. Eckert, J. F. Martinez, and G. Rubio. Context aware middleware architectures: Survey and challenges. *MDPI Sensors, Switzerland*, 15(8):20570–20607, 2015. doi: doi.org/10.3390/s150820570.

Ting Liu and Margaret Martonosi. Impala: A middleware system for managing autonomic, parallel sensor systems. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, page 107–118, New York, NY, USA, 2003. Association for Computing Machinery. ISBN 1581135882. doi: 10.1145/781498.781516. URL https://doi.org/10.1145/781498.781516.

Elsa Macias, Alvaro Suarez, and Jaime Lloret. Mobile sensing systems. *Sensors*, 13(12):17292–17321, 2013.

Geoffrey Mainland, Greg Morrisett, and Matt Welsh. Flask: Staged functional programming for sensor networks. *ACM Sigplan Notices*, 43(9):335–346, 2008.

Guoqiang Mao, Barış Fidan, and Brian DO Anderson. Wireless sensor network localization techniques. *Computer networks*, 51(10):2529–2553, 2007.

Miklós Maróti, Branislav Kusy, Gyula Simon, and Ákos Lédeczi. The flooding time synchronization protocol. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 39–49, 2004.

Luca Mottola and Gian Pietro Picco. Programming wireless sensor networks with logical neighborhoods. In *Proceedings of the First International Conference on Integrated Internet Ad Hoc and Sensor Networks*, 2006.

Luca Mottola and Gian Pietro Picco. Programming wireless sensor networks: Fundamental concepts and state of the art. 43(3), 2011.

Sirajum Munir and John A Stankovic. Depsys: Dependency aware integration of cyber-physical systems for smart homes. 2014.

Ryan Newton, Greg Morrisett, and Matt Welsh. The regiment macroprogramming system. In *Proceedings of the 6th international conference on Information processing in sensor networks*. ACM, 2007.

Luis Oliveira, Luis Almeida, and Daniel Mosse. A clockless synchronisation framework for cooperating mobile robots. In *24TH IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2018)*, pages 305–315. IEEE, April 2018.

Fredrik Osterlind, Adam Dunkels, Joakim Eriksson, Niclas Finne, and Thiemo Voigt. Cross-level sensor network simulation with cooja. In *Proceedings. 2006 31st IEEE Conference on Local Computer Networks*, pages 641–648. IEEE, 2006.

Carlos Pereira, João Mesquita, Diana Guimarães, Frederico Santos, Luis Almeida, and Ana Aguiar. Open iot architecture for continuous patient monitoring in emergency wards. *Electronics*, 8(10):1074, September 2019. doi: 10.3390/electronics8101074.

Joseph Polastre, Jason Hill, and David Culler. Versatile low power media access for wireless sensor networks. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 95–107, 2004.

Nissanka B Priyantha, Anit Chakraborty, and Hari Balakrishnan. The cricket location-support system. pages 32–43, 2000.

V. M. Raee, D. Naboulsi, and R. Glitho. Energy efficient task assignment in virtualized wireless sensor networks. In *2018 IEEE Symposium on Computers and Communications (ISCC)*, pages 00976–00979, 2018. doi: 10.1109/ISCC.2018.8538632.

R. Rajkumar, I. Lee, L. Sha, and J. Stankovic. Cyber-physical systems: The next computing revolution. In *Design Automation Conference*, pages 731–736, 2010.

S. Ramanathan. A unified framework and algorithm for channel assignment in wireless networks. *Wireless Networks, Springer*, 5(2):81–94, September 1999. ISSN 1572-8196. doi: 10.1023/A:1019126406181.

Lenin Ravindranath, Arvind Thiagarajan, Hari Balakrishnan, and Samuel Madden. Code in the air: Simplifying sensing and coordination tasks on smartphones. In *Proceedings of the Twelfth Workshop on Mobile Computing Systems and Applications (HotMobile)*, New York, NY, USA, 2012. ACM.

Anthony Rowe, Rahul Mangharam, and Raj Rajkumar. Rt-link: A time-synchronized link protocol for energy-constrained multi-hop wireless networks. In *2006 3rd Annual IEEE Communications Society on Sensor and Ad Hoc Communications and Networks*, volume 2, pages 402–411. IEEE, 2006.

Anthony Rowe, Karthik Lakshmanan, Haifeng Zhu, and Ragunathan Rajkumar. Rate-harmonized scheduling for saving energy. pages 113–122, 2008.

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*. ACM, 1999.

Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. Context-oriented programming: A software engineering perspective. *Journal of Systems and Software*, 85(8):1801–1817, 2012.

S. Sujin Issac Samuel. A review of connectivity challenges in iot-smart home. In *2016 3rd MEC International Conference on Big Data and Smart City (ICBDSC)*, pages 1–4, 2016. doi: 10.1109/ICBDSC.2016.7460395.

Lambros Sarakis, Theodore Zahariadis, Helen-Catherine Leligou, and Mischa Dohler. A framework for service provisioning in virtual sensor networks. *Journal on Wireless Communications and Networking*, 2012(135), 2012.

Bill Schilit, Norman Adams, and Roy Want. Context-aware computing applications. In *1994 First Workshop on Mobile Computing Systems and Applications*, pages 85–90. IEEE, 1994.

Bill N Schilit and Marvin M Theimer. Disseminating active map information to mobile hosts. *IEEE network*, 8(5):22–32, 1994.

Sanjin Sehic, Fei Li, and Schahram Dustdar. Copal-ml: a macro language for rapid development of context-aware applications in wireless sensor networks. In *Proceedings of the 2nd Workshop on Software Engineering for Sensor Network Applications*, pages 1–6. ACM, 2011.

Zach Shelby and Carsten Bormann. *6LoWPAN: The wireless embedded Internet*, volume 43. John Wiley & Sons, 2011.

Ryo Sugihara and Rajesh K Gupta. Programming models for sensor networks: A survey. *ACM Transactions on Sensor Networks (TOSN)*, 4(2):1–29, 2008.

Ikuta Tanigawa, Kenji Hisazumi, Nobuhiko Ogura, Midori Sugaya, Harumi Watanabe, and Akira Fukuda. Rtcop: Context-oriented programming framework based on c++ for application in embedded software. In *Proceedings of the 2019 2nd International Conference on Information Science and Systems*, ICISS 2019, page 65–72, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450361033. doi: 10.1145/3322645.3322689. URL https://doi.org/10.1145/3322645.3322689.

Punnarumol Temdee and Ramjee Prasad. *Context-Aware Middleware and Applications*, pages 127–148. Springer International Publishing, Cham, 2018. ISBN 978-3-319-59035-6. doi: 10.1007/978-3-319-59035-6_6.

Tijs Van Dam and Koen Langendoen. An adaptive energy-efficient mac protocol for wireless sensor networks. In *Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 171–180, 2003.

Norha M Villegas. *Context Management and Self-Adaptivity for Situation-Aware Smart Software Systems*. PhD thesis, University of Victoria, 2013.

Maciej Wasilak. Txthings, 2015. URL https://github.com/siskin/txThings/.

Matt Welsh and Geoff Mainland. Programming sensor networks using abstract regions. NSDI'04, page 3, USA, 2004. USENIX Association.

Dapeng Wu, Zhenli Liu, Zhigang Yang, Puning Zhang, Ruyan Wang, and Xinqiang Ma. Survivability-enhanced virtual network embedding strategy in virtualized wireless sensor networks. *Sensors*, 21(1), 2021. ISSN 1424-8220. doi: 10.3390/s21010218. URL https://www.mdpi.com/1424-8220/21/1/218.

Chenren Xu, Bernhard Firner, Yanyong Zhang, Richard Howard, Jun Li, and Xiaodong Lin. Improving rf-based device-free passive localization in cluttered indoor environments through probabilistic classification methods. pages 209–220, 2012.

Wei Ye, John Heidemann, and Deborah Estrin. An energy-efficient mac protocol for wireless sensor networks. In *Proceedings. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 3, pages 1567–1576. IEEE, 2002.

Kiran Yedavalli, Bhaskar Krishnamachari, Sharmila Ravula, and Bhaskar Srinivasan. Ecolocation: a sequence based technique for rf localization in wireless sensor networks. pages 285–292, 2005.

Yang Yu, Loren J. Rittle, Vartika Bhandari, and Jason B. LeBrun. Supporting concurrent applications in wireless sensor networks. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, SenSys '06, page 139–152, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933433. doi: 10.1145/1182807.1182822. URL https://doi.org/10.1145/1182807.1182822.

Chi Zhang, Jun Luo, and Jianxin Wu. Mawi: A hybrid magnetic and wi-fi system for scalable indoor localization. pages 275–276, 2014.

Zhiwen Yu, Xingshe Zhou, Zhiyong Yu, Daqing Zhang, and Chung-Yau Chin. An osgi-based infrastructure for context-aware multimedia services. *IEEE Communications Magazine*, 44 (10):136–142, 2006. doi: 10.1109/MCOM.2006.1710425.