Workflow Support for Low-Power Wireless Sensor and Actuator Networks

Workflow-Unterstützung für energiesparende drahtlose Sensor- und Aktor-Netze
Vom Fachbereich Informatik der Technischen Universität Darmstadt genehmigte
Dissertation zur Erlangung des akademischen Grades Doktor-Ingenieur (Dr.-Ing.)
von Systemingenieur Pablo E. Guerrero aus San Nicolás, Argentinien
Oktober 2014 — Darmstadt — D 17

TECHNISCHE
UNIVERSITÄT
DARMSTADT

DVS

Workflow Support for Low-Power Wireless Sensor and Actuator Networks
Workflow-Unterstützung für energiesparende drahtlose Sensor- und Aktor-Netze

Genehmigte Dissertation von Systemingenieur Pablo E. Guerrero aus San Nicolás, Argentinien

1. Gutachten: Prof. Alejandro Buchmann
2. Gutachten: Prof. Dr. Kristof Van Laerhoven
3. Gutachten: Prof. Dr. Pedro José Marrón

Tag der Einreichung: 14.10.2014
Tag der Prüfung: 25.11.2014

Darmstadt — D 17

To Luciana, Zoe and Dana,
from whom I draw inspiration every day.

Abstract

A decade ago, the advances in the miniaturization of electronic components made it possible to integrate three fundamental functions into a tiny, battery-operated device, namely sensing, processing and wireless communication. This gave rise to a new family of computers that, when connected with each other in masses, are known as low-power, wireless sensor and actuator networks (WSANs). These networks are the enabling technology of the Internet of Things, a market that is predicted to encompass $\sim$14 billion devices by 2020 [69]. As the initial challenges of the technology are overcome, such as identifying adequate medium access control protocols, localization techniques, and network standards, the range of possible applications has grown up.

To date, however, there is general consensus that the existing node-level programming languages do not provide adequate abstractions to implement user applications. Indeed, the predominant approach is very close to the hardware and involves the C programming language (or a variant of it). This makes it difficult for domain experts to employ the technology without a sensor network specialist. The research community has proposed a number of middleware approaches to simplify the development effort. However, these macroprogramming languages mainly focus on data extraction, and not on in-network actuation.

In this work we advocate the usage of workflows as a means to define the logic that orchestrates the network activity. With this abstraction, the loop of event-sensing, decision and actuation can be closed, leading to a reduced need for slow and error-prone human intervention in the process. In this way, the whole WSAN loop can be shifted to the network.

Our main contribution consists of the conception of a holistic workflow modeling and execution platform for WSANs, together with the design and implementation of ukuFlow, a workflow platform for low-power nodes, that runs entirely in-network, not requiring an external server infrastructure. Secondly, we present the ongoing work on the development of TUDμNet, a metropolitan-scale federation of sensor network testbeds, with which the empirical evaluation of ukuFlow, among many other research projects, was carried out.

We have identified a set of operators to compose workflows, and provided a lightweight architecture that controls the flow of such processes with an in-network execution algorithm. We present a detailed evaluation of various performance indicators for each major component of the architecture, including the data manager, command execution, and complex event detection modules. The results of the evaluation show the feasibility of the approach, in spite of the adverse resource constraints and the tough network settings employed. We strongly believe that this abstraction is of practical relevance to WSAN practitioners, while still holding promise to an in-network operation.

Zusammenfassung

Durch den Fortschritt bei der Miniaturisierung elektronischer Komponenten in den letzen Jahren wurde die Integration von drei grundlegenden Funktionen in kleine batteriebetriebene Geräte ermöglicht: Datenerfassung, -verarbeitung und drahtlose Kommunikation. Dies ließ eine neue Familie von Computern entstehen, die, wenn sie in großer Anzahl miteinander verbunden werden, als Low-Power-, drahtlose Sensor- und Aktor-Netze (engl. WSAN, wireless sensor and actuator network) bezeichnet werden. Diese Netze sind die Schlüsseltechnologie des "Internet der Dinge", ein Markt, der bis 2020 geschätz etwa 14 Milliarden Geräte umfassen wird [69]. Nachdem die ersten Herausforderungen dieser Technologie überwunden worden sind, wie zum Beispiel die Identifizierung von geeigneten Medium-Access-Control-Protokollen, Techniken zur Knotenlokalisierung, und die Spezifikation von ersten Netzwerkstandards, ist das Spektrum der möglichen Anwendungen stark gewachsen.

Es besteht allgemeiner Konsens darüber, dass die verfügbaren Programmiersprachen, die auf Ebene einzelner Knoten abzielen, keine ausreichenden Abstraktionen zur Implementierung von Benutzeranwendungen bieten. Der vorherrschende Ansatz setzt umfassende Kenntnisse in der hardwarenahen Programmierung bspw. mit C (oder eine Variante davon) voraus. Dies macht es für Domänen-Experten sehr schwierig, diese Technik ohne die Hilfe von Sensornetz-Spezialisten anzuwenden. Die aktuelle Forschung hat bereits eine Reihe von Middleware-Ansätze vorgeschlagen, um den Entwicklungsaufwand zu reduzieren. Diese Ansätze fokussieren sich jedoch hauptsächlich auf die Datenextraktion mittels Makroprogrammiersprachen und nicht auf die Ausführung der Anwendungslogik im Netz (engl. *"in-network actuation"*).

In dieser Arbeit werden Workflows als Mechanismus vorgeschlagen, um den Ablauf der Aktivitäten im Netz zu definieren. Mit dieser Abstraktion kann der Zyklus der Ereignis-Erfassung, der Entscheidung und der Ausführung kombiniert werden. Dies führt zu einem verringerten Bedarf an langsamen und fehleranfälligen menschlichen Eingriffen in den Prozess. Auf diese Weise kann die Ausführung des gesamten Zyklus in das Netz verlagert werden.

Der Hauptbeitrag dieser Arbeit besteht aus zwei Aspekten. Zum einen wurde eine ganzheitliche Workflow-Modellierungs- und Laufzeitplattform für WSANs konzipiert. Zum anderen wurde ukuFlow entworfen und umgesetzt, eine Workflow-Plattform für Low-Power-Knoten, die vollständig im Netz läuft und keine externe Server-Infrastruktur erfordert. Darüber hinaus werden die laufende Aktivitäten an der Entwicklung von TUDμNet vorgestellt, ein auf ein Stadtgebiet angelegter Zusammenschluß von Testumgebungen für Sensornetze. Mit TUDμNet wurde die empirische Auswertung von ukuFlow und einigen anderen Forschungsprojekten durchgeführt.

Um die Komposition von Workflows zu ermöglichen, wurde eine Menge von Operatoren identifiziert. Dazu wurde auch eine leichtgewichtige Architektur vorgeschlagen, die den Kontrollfluss von Workflows mit einem für ein Sensornetz ausgelegten Ausführungsalgorithmus steuert. Es wird eine detaillierte Auswertung der verschiedenen Leistungsindikatoren für jede Hauptkomponente der Architektur präsentiert. Diese beinhalten die Module des Datenmanagers, der Befehlsausführung und der komplexen Ereigniserkennung.

Trotz der Ressourceneinschränkungen, der harten Netzwerkbedingungen und der zusätzlichen Herausforderungen von der Ausführung im Netz, stellen die vorgelegten Ergebnisse die Umsetzbarkeit des Ansatzes dar und zeigen, dass die Abstraktion für WSAN-Anwender eine hohe praktische Relevanz hat.

Acknowledgments

I am immensely grateful to my research advisor, Professor Alejandro Buchmann, for his faith in my ability to succeed in completing my PhD studies. Prof. Buchmann guided me through all aspects of this dissertation, from helping shape the right research topic, through concise technological critiques, to detailed corrections in technical writing. He always supported and encouraged me to pursue my own research and teaching activities. I have also learned much from his person's humility and simplicity.

I feel very privileged for having joined the Databases and Distributed Systems research group. Dr. Mariano Cilia did not only inspire me to pursue this dissertation, he also was an excellent mentor during the undergraduate and first doctoral year, and helped me to get initial research results published. The work with Dr. Christof Bornhövd opened an exciting avenue in the industrial world. I deeply thank Marion Braun, Ursula Paeckel, Gabriele Ploch and Maria Tiedemann for converting all types of logistic hurdles into non-problems.

I am fortunate to have received guidance and feedback from Prof. Dr. Kristof van Laerhoven during my PhD studies. His collaboration in pushing teaching activities around sensor networks, and his advice during the last phase of my research, greatly contributed in expanding my knowledge. I also thank Prof. Dr. Pedro Marrón for taking the time to participate in the dissertation committee. I am thankful for the endorsement from Prof. Dr. Oskar von Stryk, PD Dr. Stefan Schneckenburger, and Prof. Manfred Hegger, with which the work on some of the most interesting testbeds out there was possible.

Special thanks go to Dr. Kai Sachs and Dr. Patric Kabus for their support in so many aspects, during all these years in Darmstadt. The frequent discussions we had provided me with a critical and practical view of research. Unraveling the complexities of sensor networks together with Daniel Jacobi and Arthur Herzog was a lot of fun. Prof. Dr. Ilia Petrov greatly helped me to structure my research activities. It was very pleasant to work with Dr. Christian Seeger and Eugen Berlin. And working with Iliya Gurov in the final years was awesome. It is a pleasure to have become a personal friend of all of you.

At DVS, I got enormous help in intellectual, technical, and personal aspects. These individuals include Dr. Christof Leng, Max Lehn, Robert Rehner and Wesley Terpstra, in topics around peer-to-peer systems; Dr. Stefan Appel, Sebastian Frischbier and Tobias Freudenreich, on event-based systems; and Daniel Bausch and Robert Gottstein regarding flash storage technologies. Teamwork and collaboration was straightforward with you. Working alongside students like Nacho, Guillermo and Hien, was a very enjoyable experience.

The research carried out for this dissertation has been funded by a number of projects. Chronologically, these are the DFG PhD Research Training Program (GRK) Nr. 492, *Enabling Technologies for Electronic Commerce* (GKEC), as well as GRK Nr. 1362, *Cooperative, Adaptive and Responsive Monitoring in Mixed Mode Environments* (GKmM); the hessian LOEWE *Center for Advanced Security Research Darmstadt* (CASED), as well as the LOEWE Research Priority Program on *Cooperative Sensor Communication* (COCOON), and the TU Darmstadt.

Last, but most important, I thank my entire family for being continuously supportive throughout these years. I am eternally grateful to my wife, Luciana, for her love, emotional support, and patience. She constantly motivated me to carry on with the thesis, and guided me when off-track. Our beautiful daughters, Dana and Zoe, are two dreams come true.

Contents

List of Figures

List of Tables

List of Acronyms

AD	Activity Diagram. 19, 22
ADC	analog to digital converter. 8, 56
AML	acute myeloid leukemia. 9
API	application programming interface. 57
ASN.1	Abstract Syntax Notation One. 51
ATM	air traffic management. 9
BPEL	Business Process Execution Language. 19, 22
BPMN	Business Process Model and Notation. 19, 22, 23, 27, 33, 43, 44, 123
CASE	computer-aided software engineering. 21
DAC	digital to analog converter. 30
EPC	Event-driven Process Chain. 22
GUI	graphical user interface. 45
HPPC	hub port power control. 94, 95
HVAC	heating, ventilation, and air conditioning. 77, 79
I^{2}C	Inter-Integrated Circuit. 8
IDE	integrated development environment. 3, 21, 46
ISO	International Organization for Standardization. 51
LQI	link quality indicator. 107
MAC	medium access control. 12, 58
MCU	micro-controller unit. 77, 99–101
MtR	Members-to-Root. 57–59, 113, 114
NOTAM	notices to airmen. 10
OASIS	Organization for the Advancement of Structured Information Standards. 19
OMG	Object Management Group. 19, 47
OS	operating system. 10, 11, 87, 102
PDA	personal digital assistant. 7
PoE	Power-over-Ethernet. 81, 82
PRR	packet reception ratio. 107
RCBF	reprogramming cycles between failures. 89, 90, 95
RFID	radio-frequency identification. 3, 7
RSSI	received signal strength indication. 107
RtM	Root-to-Members. 57–59, 113, 114, 116
SPI	Serial Peripheral Interface bus. 8, 61
UART	universal asynchronous receiver/transmitter. 8, 84

ULD unit load device. 26
UML Unified Modeling Language. 19
USB Universal Serial Bus. xv, 84–86, 88, 94
uWDL ukuFlow Workflow Definition Language. 22, 123

WDL Workflow Definition Language. 22
WfMC Workflow Management Coalition. 45, 46, 72
WfMS workflow management system. 3, 19, 21, 45, 122
WSAN wireless sensor and actuator network. iii, v, 2–5, 7–12, 17, 19–23, 26–30, 33–35, 37, 39, 43–47, 56, 57, 60, 63, 76, 99, 123–125
WSN wireless sensor network. 1, 75, 80, 84

YAWL Yet Another Workflow Language. 22

1 Introduction

If we knew what it was we were doing, it would not be called research, would it?

Albert Einstein (1879-1955)

Bell's Law states that a new class of computing devices is formed roughly every decade [8]. The timeline of the first computer classes was composed of mainframes, supercomputers, personal workstations, and notebooks. Mark Weiser's work on *ubiquitous computing* in the late 1980's triggered a new class, which encompassed devices that "*weave themselves into the fabric of everyday life until they are indistinguishable from it*" [138]. When, in the early 2000's, researchers began pursuing the idea of integrating three fundamental functions into a tiny device, namely *sensing*, *processing* and *wireless communication* [110], a new family of computers was born. These computers were collectively called wireless sensor networks (WSNs). The first sensor node prototypes were built with small 8-bit microcontrollers, a few kilobytes of RAM, a simple wireless radio (around 10kbps) and integrated sensors. Despite their simplicity, these served to kick off the technology and develop novel applications that captured information from the physical world at an unprecedented high resolution.

One of the earliest sensor network deployments was carried out at Great Duck Island [91], where researchers in the *life sciences* aimed at unobtrusively learning about seabirds and their environment by installing sensor nodes in and around burrows during nesting periods. Through sensor nodes placed in glaciers [93], *geologists* were able to monitor and better understand dynamics of glacier behavior. In dutch potato fields, sensor networks aided precision agriculture in protecting crops against fungal diseases, strongly associated to the climatological conditions [79]. While the development and deployment of applications in this first group encountered problems such as power conservation for network longevity and administration of the remote site, the high-level logic of these applications was simple: push the (mostly) raw observed data out of the sensor network. The parameters of this logic, e.g., the sampling rates, were specified by the domain experts.

A second group of applications goes beyond the continuous observation of the environment: an event of interest must be detected, afterwards followed by an observation of the phenomena. One such deployment was carried out with *volcanologists* at Ecuador's Volcán Reventador [141]. Nodes waited for their seismometer readings to exceed a certain threshold, and then notified a base station, which triggered a data collection phase. Structural health monitoring systems like *Wisden* [145] or the one deployed on the Golden Gate Bridge [75] have been built together with *civil engineers*. The detection of vibration events was followed by local data storage and posterior progressive coding (compression) for efficient data transmission. The parameters of the application logic, i.e., the sampling rates and the thresholds, were also given by the experts in the domain, who may vary them to adjust or refine the experiment.

A third type of applications enables a more complicated interaction. The habitat monitoring system deployed at the Coastal Redwood Forests of California [89], for example, allowed relational queries to be injected into the network, whose results were aggregated as they were streamed back to a base station. In the industrial scenario of the EU CoBIs project [127], nodes attached to chemical drums cooperated with each other, without using an external infrastructure, to check for hazardous situations and violations of safety regulations. Again, the logic behind these applications was precisely stated by domain experts, now in the shape of SQL-like queries or inference rules, respectively.

A common denominator observed across all these systems are the difficulties in developing application software. This is evidenced, for instance, by the interest from the research community to report at scientific venues about deployment experiences (and mentioned, e.g., in [125]). Application developers must typically counter the processing, storage and energy constraints of the sensor nodes, while coping with their massive network scale and the complexities of the wireless environment on which they are deployed. To this purpose, many *middleware* approaches have been proposed and discussed in the literature [62, 114, 98]. Reportedly, however, very few applications have been developed exploiting high-level programming constructs [100]. The question of which programming abstractions are best suited for sensor network applications indeed remains open.

The described evolution of applications, in addition, shows that in the last years the research focus has been placed on extracting data from the network, reaching an extremely low-power operation lasting months or years. An aspect that, to a certain extent, has not received sufficient attention is that sensor network nodes can be easily enhanced to perform *actuation*, forming a wireless sensor and actuator network (WSAN) [2]. In essence, actuators open or close a switch, or drive more complex actuators via scalar values [73]. Individual actuators can be as diverse as sensors. A WSAN can thus not only monitor but also affect the physical environment, closing the loop between sensing and actuation. Applications exploiting actuators include first attempts at controlling irrigation and applying pesticides at vineyards [16], and adaptive light control in tunnels [19].

While a considerable amount of logic has been effectively moved into the networked nodes' software, the decision on how, when and where to perform certain actuation is only taken off the network, either by a *human*, or with the help of a decision support system. Such an approach corresponds to an *interactive computing* model (as described by Tennenhouse [131]), where humans are placed *in the loop*. This is a logical first stage, since only after data had been consolidated in a central database and understood, was it possible to reason about it and decide what to do next.

Much of the application logic carried out by humans, however, can also be pushed into the WSAN, potentially reducing the need for unnecessary, slow and error-prone human intervention in the process. Intuitively, this approach presents a number of benefits, namely:

- faster reaction to phenomena in the environment, since the decision is taken closer to the point of interest,
- enhanced reliability, due to the lower chance of losing messages, and
- energy savings (i.e., extended network lifetime) derived from the reduced number of messages exchanged between event sources, sinks and actuation nodes.

Application developers thus need a system that enables *in-network operation*, shifting the whole WSAN loop to the network. Such systems correspond to the *proactive computing* model, where humans are placed *above* the loop.

Apart from exploiting actuation capabilities, we observe that many sensor networks are highly optimized but typically limited to a *single* application. From a user perspective, however, it is highly conceivable that the deployed sensor/actuator infrastructure is exploited for *multiple*, concurrent applications [126, 137, 101]. In this work we adhere to the premise of multi-purpose sensor networks and consider the aforementioned aspects as a whole.

1.1 Problem Statement

As noted from the previously cited application groups, it is the domain experts who have the knowledge of the behavior of the target environment and what is needed to do with it as a response. They have the detailed information of what is expected to happen, either from what they have observed, learned or suspect and want to corroborate. However, the existing tools to develop and deploy applications are too low-level for most domain experts. A central goal of our work is finding out how to simplify the definition of the application logic that orchestrates the WSAN activity by identifying adequate high-level programming abstractions.

Currently, applications are mostly written using a combination of an imperative approach, like the C language, and a number of (low-level) interrupt-driven state machines that react to incoming messages or sensed data, like nesC [46]. This is employed in all mainstream sensor network operating systems like Contiki, SOS and TinyOS [33, 59, 83]. The application logic of such programs is specified at a node-level, i.e., it describes what an individual node should do. As the applications get more complex and non-functional requirements such as reliability (uninterrupted data delivery) or system longevity (power management) come into play, these **state machines become too complex to develop** [35], and extending them becomes cumbersome and error-prone.

In this dissertation we argue that high-level application behavior can be naturally expressed by domain experts through *workflows* with relative ease. In their more general sense, workflows are *collections of interrelated work tasks, initiated in response to an event, achieving a result for the stakeholder* [121]. As such, workflows are an essential concept in many scientific and engineering disciplines, and a common tool to experts in most domains. The overall goal of this work is thus the investigation of the feasibility of a holistic workflow approach that enables domain experts to develop applications and their deployment on these WSAN devices. Given the multitude of contexts in which the concept of workflows is employed, a first challenge consists in **identifying the right workflow modeling abstractions and their applicability to WSANs**.

Unfortunately, current workflow management systems (WfMSs) were not designed to target WSAN applications. Typical strategies simply attempt to extend workflow engines –originally built to operate on electronic documents– to interact with pervasive devices, retrieving information from networked sensors or radio-frequency identification (RFID) tags, and instructing actuators to perform an action. In contrast, we advocate pushing the workflow engine down into the sensor network for an autonomous operation. Existing workflow engines, however, do not have a footprint that is small enough to fit on state-of-the-art sensor nodes. While a current workflow engine (e.g., IBM's WebSphere MQ Workflow) minimally requires a 300MHz processor with at least 64 MB of RAM and 1.5 GB of hard drive space, a state-of-the-art sensor node typically offers a 10MHz microcontroller with less than 128KB of RAM. A second (and resulting) challenge tackled by this thesis is thus the **design and development of a runtime system that supports the identified workflow programming abstractions** while simultaneously considering resource constraints, energy efficiency, communication unreliability, and self-organization.

1.2 Proposed Approach and Contributions of this Thesis

Much work has already focused on simplifying the development of applications for WSANs (e.g., [46, 89, 6, 40]). However, less attention has been paid to offering approaches accessible to domain experts not necessarily familiar with programming languages and concepts common for computer scientists and traditional software developers. The approach proposed in this work, depicted in Fig. 1.1, consists in employing *workflows* as first-class citizens in the development of applications. Our approach is divided in 4 steps. In the *modeling* step, domain experts can define the application logic by means of an integrated development environment (IDE) especially designed to support workflows. In a second step, this workflow model is *verified* against a number of conditions for its correctness, and then *converted* into an intermediate representation that the WSAN system can handle. Once ready, the third step consists in *registering* the workflow with the network. The WSAN performs a number of checks, (e.g., whether resources are available) and provides the user with feedback about the completion of the registration. From that point on, the sensor network takes care of the *instantiation and execution* of the workflow.

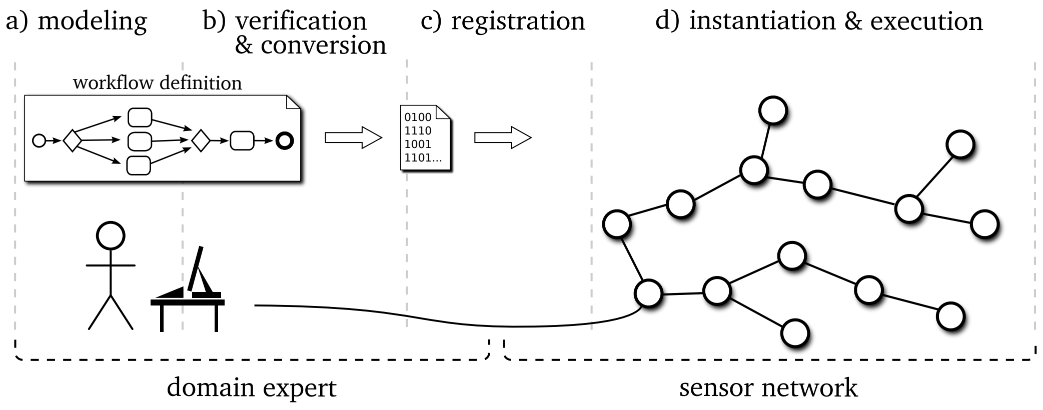


Figure 1.1.: Approach

The contributions of this work are:

- The conception and definition of a holistic workflow modeling and execution platform for sensor networks that enables domain experts to develop and deploy sensor network applications, abstracting from low-level issues such as medium access control, routing, task synchronization and node group construction, among others.
- The design and implementation of **ukuFlow**, a workflow platform for resource-constrained, low-power wireless devices, which runs entirely in-network, not requiring an external server infrastructure.
- The development of **TUDμNet**, a metropolitan-scale federation of sensor network testbeds that enables experimentation and systematic evaluation of sensor network systems, in general, and is also used to evaluate networking aspects of the ukuFlow platform, in particular.

We present ukuFlow's macroprogramming model, which enables domain experts to describe high-level application logic, as well as its runtime workflow engine, which interprets and executes workflows within the WSAN, reducing latency and energy consumption.

1.3 Dissertation Roadmap

This dissertation is structured as follows.

In Chapter 2, we present the background for this work, and discuss the related work. We provide an overview of the targeted hardware platforms and their components, describe ubiquitous applications and elaborate a case study that is used in many parts of the thesis to exemplify several of the macroprogramming constructs. We discuss well-known approaches suggested in the literature, and provide arguments that support the usage of workflows for the development of WSAN applications.

Chapter 3 goes straight into the mixed graphical/declarative/imperative models used in this work: the workflow model, the data model, the scoping model, the actuation model, and the event model. We exemplify each of these constructs, and also present a comprehensive workflow based on an airport scenario, which includes the majority of the model elements.

We present the design and implementation of ukuFlow in Chapter 4. We elicit the requirements and derive the high-level architecture. We describe an Eclipse plug-in that enables the modeling of workflow and event diagrams, as well as the mechanism to upload workflows to a connected node. We then provide details of the runtime architecture of ukuFlow, including extensions to the networking mechanisms and the event generation and composition algorithms.

In order to carry out a realistic evaluation of the approach, in Chapter 5 we digress from the main topic and discuss means to evaluate empirically sensor network systems. We describe the ongoing efforts in building a series of testbeds that enable a repeatable experimental evaluation of WSAN software. Then, in Chapter 6 we give a detailed evaluation of ukuFlow, which we split into a micro evaluation, dedicated to single-node aspects, and a macro evaluation, targeted at the more complex aspects of this work that involve network interactions. We conclude and provide pointers to future work in Chapter 7.

2 Background and Related Work

If you steal from one author, it's plagiarism; if you steal from many, it's research.

Wilson Mizner (1876-1933)

In this chapter we elaborate on the context of this thesis, and describe applications targeted by our approach. We provide a concrete example revolving around an airport scenario. The application is modeled by means of an abstract workflow, which motivates the use of this programming method. Then we present the necessary building blocks leading to the approach that tackles the challenges of modeling application workflows and designing a workflow execution environment for low-power WSANs. Finally, we discuss the related approaches existing in the literature.

2.1 Low-Power Wireless Sensing Systems

Ever since their initial conception, a sensor node has been defined as a unit with a microcontroller, memory, a radio communication unit, a number of sensors (and/or actuators), and a (usually limited) power supply. Both informally, as well as in the literature, the term *sensor network* has been overly abused, spanning different classes of devices such as:

- Simple RFID tags that are able to store a few bytes of data and reply with short messages. Being powered by microwaves sent by an RFID reader, these have a potentially unlimited lifetime.
- *Mote-class* devices that exhibit potentially years-long lifetime in typical *sense-store-send-sleep* applications, when adequately operated, thanks to their 8/16-bit processors with low active power draws and deep sleep modes. Example nodes are the Mica2 [97], the TelosB [109] and the Z1. More recently, 32-bit nodes have appeared such as the Econotag, jNode [120], Opal and Egs [76], that are much faster and promise to only incur a small overhead in the power budget.
- High-performance nodes, targeted at high-resolution signal processing applications, that employ ARM-based 32-bit processors. Platforms such as the SunSPOT [122] and the Imote2 [103] offer a higher flexibility at the cost of an increased energy consumption that leads to shorter network lifetimes.
- High-precision, custom-made sensing equipment that also provides wireless connectivity. These are used in specialized scientific equipment, or are embedded into large trucks or aircraft machinery (e.g., [148]).
- Smartphones, personal digital assistants (PDAs), tablets and the like, used as sensing devices and other sensor-enriched, Ethernet-based devices, commonly acting as sensor network gateways, such as the Stargate [108].

We summarize these devices in Table 2.1. While, as described by Hill et al. [64], a heavily heterogeneous system could encompass devices from all these classes interacting with each other in a hierarchy, in many applications, such as those described in Section 2.3, a simpler organization composed of sensing devices together with a limited number of gateways suffices. In this work, we use the acronym WSAN to refer

Table 2.1.: Device classes and relevant properties

Device Class	Processing (GHz)	Memory RAM	ROM	Examples	Power active (mA)	sleep (μA)	Battery Lifetime
- tags	n/a		< 100 B	RFID, NFC	n/a		unlimited
- mote-class nodes	< 0.03	< 32 KB	< 256 KB	TelosB, Econotag	2	1	months to years
- high perf. nodes	< 0.5	256 KB	32 MB	SunSPOT, Imote2	7	500	weeks to months
- custom hardware	~0.1 to ~1	KB to MB		U-Mote	70	84	weeks to months
- smartphones	~0.5 to ~1.5 (multicore)	2 GB	64 GB	Nexus 4, iPhone 5	137	5400	days to weeks

strictly to **mote-class** devices. While this enables a longer system lifetime, it simultaneously sets an upper bound on the **footprint** of the software stack, requiring a careful design to ensure low-power operation.

2.2 Sensors and Actuators

Hardware miniaturization techniques, together with reduced manufacturing costs, have made a large variety of sensors and actuators available that, depending on the target application at hand, can be used to sense and actuate upon environmental phenomena. Many of these sensors offer a low power consumption and require no additional calibration.

Most WSAN nodes are built with integrated, standard sensors (e.g., to measure temperature, humidity, sound or light) connected to analog to digital converter (ADC) ports; other sensors or actuators (such as LEDs, servos, buzzers or relays), can be added with relative ease.

To date, much more versatile sensors and actuators are available than there were in the first years of sensor network research. Using more elaborate bus interfaces such as universal asynchronous receiver/-transmitter (UART), Inter-Integrated Circuit (I^2C) and Serial Peripheral Interface bus (SPI), sensors and actuators such as the well-known *Phidgets*[1] [49] can be simply plugged into, e.g., the Z1 nodes (cf. Fig. 2.1a). Besides providing raw sensor data, these can deliver very valuable *high-level* information. This is the case, for example, of the ADXL345, a 3-axis accelerometer built into the Z1 that can detect high-level events such as tapping or double-tapping it. This is typically achieved in a more energy-efficient manner than it would through a software driver.

2.3 Pervasive Applications and Workflows

There are innumerable scenarios where pervasive, WSAN technologies come into play. Examples include urban waste management, emergency/response scenarios, supply chain management, smart metering and grid, home automation, and health/elderly care, to name a few (the reader is referred to [94] and to [112] for an illustrated selection of applications).

As presented in the introduction, pervasive applications are of varied nature, which we illustrate in Fig. 2.2. The most commonly envisioned type are known as *sense-collect* applications: nodes sample their

[1] Phidgets, Inc., http://www.phidgets.com/, last visited October 2014.

sensors and send data to a sink through multi-hop routes. An example application is the environmental monitoring system described in [91]. More elaborated versions of this type of application require detecting events of interest by applying filter and aggregation operations on the sampled raw data. At the other extreme of the spectrum there are *inform-actuate* applications, where a central entity commands nodes to perform a certain action. One example of this application type is a traffic light control system [140]. There exist specialized, efficient protocols to support these types of applications (e.g., CTP [48] for collection applications, Glossy [37] for data dissemination).

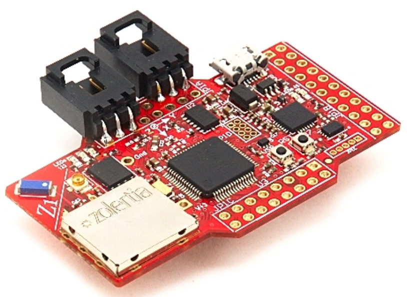
WSAN applications, in general, combine requirements from the previous two, and are known as *sense-and-respond* applications [24]. Many of the sense-and-respond applications can be modeled using workflows. Figure 2.3 presents two examples. Figure 2.3a presents a supply chain management scenario, adapted from [54], which models business logic to be executed at a warehouse when a truck arrives. Fig. 2.3b illustrates the guidelines of a healthcare scenario to manage acute myeloid leukemia (AML) in children, adapted from [29].

While some processes have not been originally modeled with pervasive technologies –sensors and actuators– in mind, they can easily be enhanced with these, as is the case in the AML treatment, e.g., by networking medical laboratory devices to automate data processing. Other processes, as in the supply chain management scenario, have been built with WSAN technologies from the ground up, e.g., to monitor the cold chain during a shipment's distribution activities.

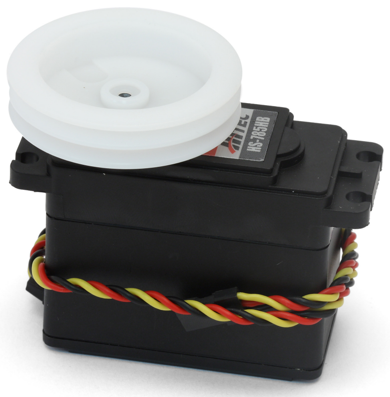
2.3.1 Enhanced Airport Management Case Study

We now turn to a scenario concerned with airport operations. Motivated by the ever increasing cargo and passenger air traffic, the optimization of air traffic management (ATM) procedures has become imperative. One way to alleviate this situation is by reducing the inter-aircraft (temporal) separation, i.e., fitting more planes in the arrival-departure sequence, as well as by minimizing the time aircraft spend on ground (known as turn-round time).

ATM organizations have well-defined, strict processes for their various traffic control activities. To model these operations, workflow technologies have been used for years [86], and controllers in the airport tower employ various technical facilities to monitor and guide approaching, on-ground and airborne aircraft, using workflow engines to keep track of (aircraft) state changes [67].



(a) Zolertia Z1 node with two Phidget ports (reproduced with permission from Advancare, SL.)



(b) Example winch servo Phidget (reproduced with permission from Phidgets Inc.)

Figure 2.1.: State-of-the-art WSAN hardware

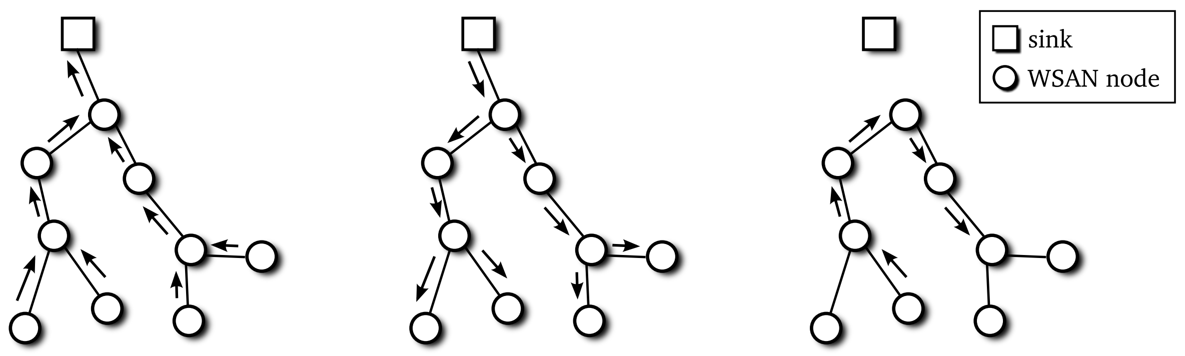


Figure 2.2.: Application types. Left: *sense-collect*, middle: *inform-actuate*, right: *sense-and-respond*

Figure 2.4 illustrates a process in an airport setting, concretely the tasks an aircraft goes through for departure in the case of positive progress (exceptions are not shown), adapted from [134]. The process includes a number of activities (actual operations or commands to be performed, such as *De-ice* or *Line-up*), as well as updates to aircraft state (such as the *clearances*). State changes, or transitions, are performed by the pilot through an automated flight service station (shortened AFSS), which provides pilot briefings, receives and processes flight plans, originates notices to airmen (NOTAM), and broadcasts aviation weather briefings. In the rest of this work we will return to this scenario (and subparts of it) to discuss several aspects of the approach proposed in this thesis.

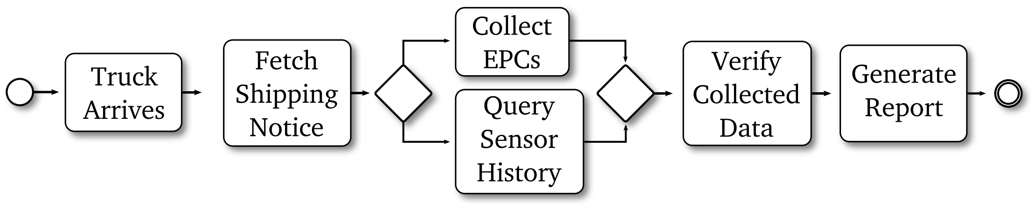
2.4 Application Development with Operating Systems - and WSAN Libraries

WSAN applications are mostly developed and written using a programming language such as C (or a dialect thereof, like nesC [46]). While not as abstruse as binary and assembly languages, WSAN languages are still very low-level. With them, developers resort to collections of libraries that provide essential operating system (OS) functionality such as CPU scheduling and hardware access (e.g., radio and sensors). The research community has developed a number of OSs specific to resource-constrained networked sensors (e.g., TinyOS [65], Contiki [33], SOS [59] or Mantis [9]). To date, sensor network research, as well as first WSAN prototypes and products, are based mostly on TinyOS and Contiki. Despite the popularity reached by TinyOS as OS for WSANs in the first years, the fact that Contiki is written entirely in C, among other features, has made its adoption much simpler for WSAN researchers and system developers [80].

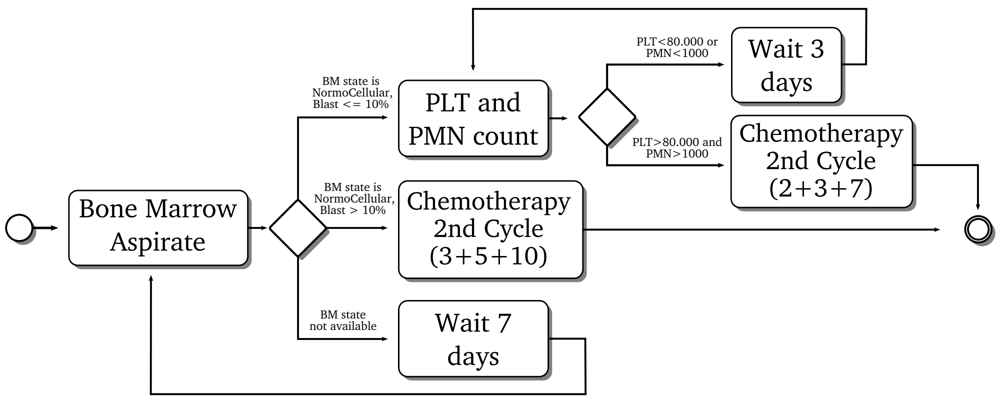
Contiki is an open source OS, ported to a large number of 8/16 bit platforms such as the TelosB, Z1 and MicaZ, and some 32 bit nodes such as the Econotag, to name a few. At runtime, Contiki is a multi-tasking system composed of a core event-driven kernel and a set of modules. A Contiki application can make use of traditional threads, or resort to light-weight threads called *protothreads* [35].

In Contiki, two communication stacks are available: *uIP* and *Rime*. uIP [32] is a small-footprint TCP-UDP/IP stack (typical configurations require ∼5KB of code and ∼2KB of RAM). This makes it possible to communicate, through the Internet, with Contiki nodes. Rime [34] is a set of communication protocols ranging from *local broadcast* to *reliable network flooding*, organized as a strict stack of thin layers that applications can connect to. Contiki applications can use either stack, or these two stacks may be connected (transmitting uIP data over Rime or vice versa).

The resource constraints imposed by the hardware platforms have limited considerably the services offered by a WSAN OS, leaving out advanced aspects such as memory protection, persistent data storage or network transport protocols. Similarly to other WSAN OS for mote-class nodes, Contiki applications are cross-compiled to generate a program image with which nodes are later reprogrammed (i.e., flashed).



(a) Workflow for handling arrival of a truck with a shipment, adapted from [54]


Bone Marrow Aspirate
BM state is NormoCellular, Blast <= 10%
BM state is NormoCellular, Blast > 10%
BM state not available
PLT and PMN count
Chemotherapy 2nd Cycle (3+5+10)
Wait 7 days
PLT<80.000 or PMN<1000
PLT>80.000 and PMN>1000
Wait 3 days
Chemotherapy 2nd Cycle (2+3+7)

(b) Workflow for handling acute myeloid leukemia in children, adapted from [29]

Figure 2.3.: Applications in different domains modeled with workflows

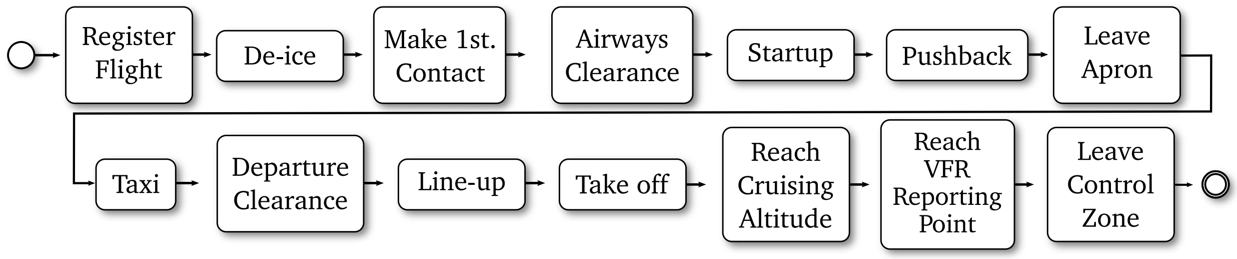
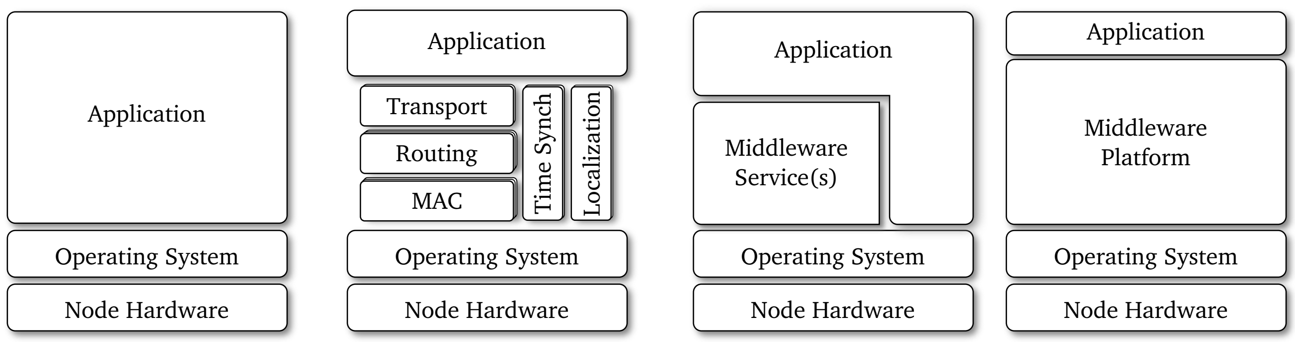


Figure 2.4.: *Surface management operations* workflow for handling an aircraft departure at an airport

While Contiki was the first OS that explored the idea of compilation of modules that can be dynamically loaded and unloaded at runtime [33], and after several research attempts to increase program modularity, state-of-the-art OSs still fail to offer a reliable mechanism to support module dissemination and dynamic linking, a feature that has a major impact in the design of WSAN systems.

2.5 Programming Abstractions, System Services and WSAN Middleware

Developing a meaningful WSAN application becomes a challenging endeavour as soon as it has to fulfill certain requirements such as scaling up beyond a dozen nodes, being fault-tolerant with respect to environmental conditions, or supporting network dynamics (i.e., churn). Figure 2.5 compares several approaches to tackle these problems. The traditional approach of writing applications (cf. Fig. 2.5a) using the bare operating system functions –the simple CPU scheduling and sensor drivers– has the benefit that it allows a high degree of customizability, but has been shown to be very complex and time consuming. Components developed within the *Application* block will most likely not be reused in other similar applications due to their level of specialization and intertwining.



(a) Entire app. built on bare operating system **(b)** App. built on extended set of services **(c)** App. using a middleware approach. Left: with a high-level service, right: holistic solution

Figure 2.5.: Approaches in developing WSAN applications

Following a *divide and conquer* approach, and reusing existing blocks such as medium access control (MAC) and routing protocols, time synchronization or localization services, it is possible to simplify the application development (cf. Fig. 2.5b, arrangement of components is rather schematic). Prior work on these individual research areas has enabled understanding their essential tradeoffs and led to relatively stable libraries, some of which are integrated into OS libraries. While reusing these building blocks indeed might reduce the development effort, the decision of *which* combination of the available building blocks would satisfy the application requirements, at best, remains complex. This is exemplified already through MAC protocols, for which a database has been collected[2] (parodically called the *MAC Alphabet Soup*, due to the names that authors give to their protocols, like B-MAC, S-MAC, T-MAC, etc.). A catalog of routing protocols presents the same situation [1]. Together with the problems of resolving interface incompatibilities between these blocks, it becomes a barrier for WSAN non-experts to follow this second approach.

A third alternative, which has been previously suggested [62, 114, 98] and to which we adhere in this work, is to resort to *middleware*, i.e., "software that provides a programming model above the basic building blocks of processes and message passing" [26]. Middleware can be seen as a block that sits atop the OS and leverages from its simpler functions to provide developers with higher-level, distributed programming abstractions.

Abstractly speaking, the central idea behind middleware is to use a generic block that implements and offers more elaborated services, thus simplifying the development of applications. While the final solution would not be optimal *for all* conceivable applications, it is targeted at serving well *many* scenarios. The research community has proposed a number of middleware approaches, each of which inevitably fits better a different subset of applications or goals. There exist various perspectives from which these approaches can be classified; a detailed evaluation is presented in [128, 100].

We distinguish, primarily, between *high-level services* and *holistic solutions* (cf. Fig. 2.5c). A high-level service is one that needs to be used as extension to (or must be invoked from) the programming language, and is conceived to simplify *one* aspect of the application development. High-level services have been proposed to deal with different aspects of WSAN application development. Examples of such services include *routing*, *security*, *node localization*, *sensor calibration* and *object tracking*. A holistic solution, in contrast, offers self-contained means to define the *entire* application logic, typically with less intention to interact with other approaches. In the next subsections we first describe one broadly used service, namely node grouping, and later provide an overview of relevant holistic solutions.

[2] The MAC Alphabet Soup website, www.st.ewi.tudelft.nl/ koen/MACsoup/, last visited October 2014.

In many systems, the network nodes are not expected to all perform the same task. While some nodes are busy executing one function, others might be idle (and saving energy) or performing another task in parallel. Returning to our airport scenario workflow (cf. Fig. 2.4), for example, we can associate tasks to geographical areas in the airport's space. In Fig. 2.6 we extend the workflow with labels and colored backgrounds of the various areas an aircraft moves through, such as the apron, taxiway, or runway. Each of these areas could be utilized for different tasks simultaneously, and hint at the need to define *spatial* aspects of the execution of the tasks. Besides a node's physical position (or its distance to a certain coordinate), there are other criteria that can be used to specify a group, such as:

- nodes within a maximum number of hops from another node
- proximity to an event-detecting node
- nodes with certain type of sensors or actuators (e.g., in networks with heterogenous nodes)
- nodes reading particular sensor values (e.g., between certain thresholds), and logical combinations among these
- nodes with certain characteristics (e.g., CPU speed, communication interfaces, or battery level).

One approach to realize this functionality is pursued by *Generic Role Assignment* (GRA) [115, 42]. In GRA, the user submits a *role specification*, that is, a list of the possible roles and the rules for the nodes to decide which roles to adopt. For instance, when building a topology for a clustering routing algorithm, nodes could choose between being '*cluster heads*', '*gateway*' or '*slaves*'. The role specification for this example is presented in Listing 2.1. At runtime, the role specification is distributed throughout the network, and each node picks one or more roles out of the specification depending on conditions such as its internal state. Further revisions of the initial concept have extended the capabilities, e.g., by adding security considerations [4].

Another system offering a similar functionality is *Abstract Regions* [139], which extends the concepts in an earlier platform (Hood [143]) by considering multi-hop neighbors. An abstract region establishes a relationship between a particular node and other nodes in the network. There are four operations to be carried out on an abstract region. The first is *neighbor discovery*, which identifies the set of candidate neighbor nodes of an abstract region by exchanging messages via broadcast. *Enumeration* returns the set of nodes that are members of the abstract region. *Data sharing* is employed to share variables in the form of name-value pairs within a region through the functions `get(v,n)` and `put(v,n)`, where `v` is the value and `n` is the id of the node in the abstract region. The last operation is *data reduction*, which, given

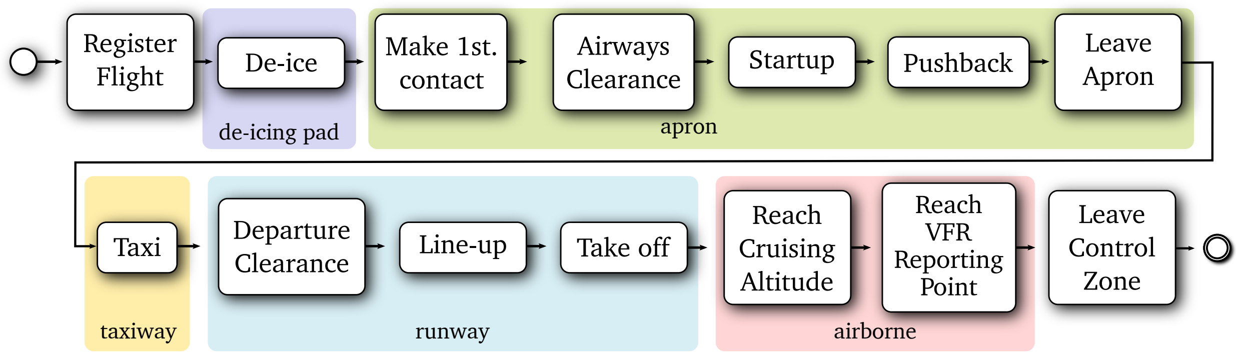


Figure 2.6.: Geographical extension of tasks in surface management operations workflow

```
CH :: {
  count(1 hop) {
    role == CH
  } == 0
}
GW :: {
  clusterheads == retrieve(1 hop, 2) {
    role == CH
  } &&
  count(2 hops) {
    role == GW &&
    clusterheads == super.clusterheads
  } == 0
}
SLAVE :: else
```

Listing 2.1: A role specification in the Generic Role Assignment approach

an associative operator, like *sum* or *min*, reduces a variable in the member nodes in the region and stores it in a shared variable. Abstract Regions was implemented and evaluated on TinyOS.

Based on previous work [38] in the area of publish/subscribe systems, in the context of this thesis we have developed a related grouping abstraction called *Scopes* [71, 72]. A scope is a high-level abstraction, provided to applications, to organize nodes into groups. A scope is defined through a logical expression, which must be satisfied by a node in order for it to become member. Scopes can be nested, and thus are organized in a hierarchy. We provide two example scope definitions: Listing 2.2 presents the specification of scope `HVAC_Outlier`, where nodes must fulfill the conditions of being attached to HVAC (humidity, ventilation and air conditioning) equipment, having temperature sensors connected to them, and reading temperature sensor values less than or equal to 20 °C or greater than or equal to 25 °C. The property `ATTACHED_TO` is application-specific; corresponding nodes set it to `hvac`, e.g., at deployment time. Listing 2.3 exemplifies the specialization, or refinement, of a scope called `Room22` by selecting only those that have an accelerometer sensor.

```
CREATE SCOPE HVAC_Outlier AS (
  ATTACHED_TO = 'hvac' AND
  EXISTS SENSOR_TEMPERATURE AND
   (SENSOR_TEMPERATURE_CELSIUS <= 20 OR
    SENSOR_TEMPERATURE_CELSIUS >= 25)
);
```

Listing 2.2: Definition of an HVAC scope

```
CREATE SCOPE Room22MotionNodes (
    EXISTS SENSOR_ACCELEROMETER
) AS SUBSCOPE OF Room22;
```

Listing 2.3: Definition of a subscope through *specialization*

Once a scope is created, it continues to exist until it is explicitly removed. The framework takes care of reliably maintaining the scope membership, even as nodes fail, temporarily leave and rejoin again, or as nodes' properties vary. A scope is created at a node, called *scope root* node, that takes the administrative role of maintaining the scope. Once a scope has been created, the framework enables a bidirectional communication channel between the scope root and the scope's member nodes. Different routing algorithms have been used to implement the Scope functionality; these will be described closer in Section 4.5. In this work we resort to the usefulness of node grouping abstractions by exploiting the Scopes framework.

2.5.2 Holistic Solutions

The previously described systems are conceived to be used, or invoked from, applications written in another lower-level programming language such as C. We now turn to holistic solutions, which, as we will see, offer an integrated environment to define and execute the application logic. We classify these approaches from a programmer's perspective, i.e., from the set of programming abstractions and run-time mechanisms offered. Note that this is a non-exclusive classification, since one approach might share some of its concepts with others.

Virtual Machines

Following the observation that a wide range of WSAN applications is composed of a small number of high-level primitives, one approach is to identify these primitives and offer a concise way to represent these programs. Virtual machines, in the form of *bytecode interpreters*, are a natural fit for this task: given a bytecode *instruction set*, a system interprets at runtime the programs by translating and executing the necessary low-level instructions.

Maté [81] is one such virtual machine, implemented on top of TinyOS. In Maté, programs are composed of platform-independent, 1-byte wide instruction bytecodes. Programs are furthermore broken into *capsules* with a maximum of 24 instructions, a limit chosen so that a capsule fits in a TinyOS packet when using RFM's TR1000 radios. The Maté virtual machine makes use of two stacks, one for operands and one for return addresses; most operations operate on elements of the operand stack, which makes operations short. The bytecode instruction set contains instructions of different granularity (i.e., different number of clock cycles). Listing 2.4 presents an example capsule that replicates the TinyOS program `cnt_to_leds`, and contains simple instructions like `copy` or `putled`, which take a couple hundred clock cycles. Other instructions are of a much higher granularity, like `send`, which invokes an ad-hoc routing algorithm and takes thousands of clock cycles.

```
pushc 1     # Push one onto operand stack
add         # Add the one to the stored counter
copy        # Copy the new counter value
pushc 7
and         # Take bottom three bits of copy
putled      # Set the LEDs to these three bits
halt
```

Listing 2.4: A Maté capsule to show the bottom 3 bits of a counter on a node's LED (adapted from [81]).

The interface exposed by Maté is of higher-level than that of the system it is built on, TinyOS, in that it hides asynchrony: capsules are suspended when an instruction is invoked and until its execution is finished. For example, when the `sense` instruction is invoked to retrieve data from a sensor, the capsule is blocked until sensor data is ready for its reading. This makes capsules simpler than TinyOS programs because the developer does not need to deal with asynchronous event notifications (e.g., implementing a handler for the reception of sensor data).

In order to execute code in remote nodes, bytecode capsules need to be sent through the network. For this purpose, an application must invoke the `forw` instruction, which transmits the capsule to a node's neighbors. This brings extra complexity which needs to be managed in the capsule's code.

Mobile Agents

The forwarding of Maté capsules is known as *weak* code mobility: a capsule's code is executed from a fixed start address once it is installed on a neighbor node. A more sophisticated approach to code mobility is achieved by *strong* code mobility, which enables code to clone itself –or migrate– to other nodes, thereby resuming execution on the instruction where it was stopped. This is achieved by not only forwarding the code but also any necessary process data and execution state (such as registers). This is the approach followed by mobile agents.

The Agilla system [40, 39, 41], which is based on Maté and was developed with TinyOS, implements the mobile agent functionality for sensor networks. An Agilla application is conformed by a number of mobile agents distributed throughout the network. Multiple agents can exist, in parallel, on a single node; these resort to a shared memory model (in the form of a tuple-space) to communicate with each other. In addition, each Agilla node maintains a list of all its one-hop neighbors (and their positions). This allows an agent to decide where to migrate or clone to. Agents can be much larger than capsules: these can occupy up to 440 bytes (20 TinyOS packets). Since most Agilla instructions are one byte wide, agents can have up to 440 instructions.

While the idea of employing a bytecode interpreter indeed is an effective way that leads to more compact programs, the most important factor that restricts a broader adoption of these systems is that *domain experts* are typically not familiar with their assembly-like, bytecode language. Other systems have been proposed that integrate some of the concepts of the virtual machine approach, such as SwissQM [102] and MoteRunner [18], and which provide users with a toolchain to convert code written in a higher-level language, like SQL and a Java dialect, respectively, into an intermediate bytecode that the VM can interpret. We discuss these approaches in the next subsections.

Databases

Researchers from the database community have explored the idea of using *relational* abstractions as interface to sensor networks, defining application logic in terms of SQL-like statements in the `SELECT-FROM-WHERE` form [47, 7].

Two examples in this category are Cougar [12, 147] and TinyDB [88, 89]. In these systems, the main difference to the traditional relational model is that queries are *continuous*, i.e., are periodically executed against a data stream originated at the sensor nodes. In TinyDB, for instance, the data model is conformed by a virtual table called `sensors` that contains one column for each sensor type connected to the nodes, and one row for each node in the network. The periodicity of the evaluations is specified by the `SAMPLE PERIOD` clause (cf. Listing 2.5).

```
SELECT nodeid, temperature, humidity
  FROM sensors
  WHERE floor = 4
  SAMPLE PERIOD 5 SEC
```

Listing 2.5: A TinyDB query to obtain temperature and humidity readings from nodes on a 4th floor (adapted from [47])

Queries are submitted to a central server that is connected to a base station node. The server-side software *parses* the queries and performs a number of *query optimizations* before sending it through the WSAN. For instance, conditions in the `WHERE` clause can be rearranged so that sensors with lower sampling costs are evaluated first (e.g., a light sensor vs. a CO_2 sensor). The server is also in charge of collecting the query results as they are received. Once a query has been disseminated, nodes in the sensor network begin with the *query processing* by acquiring data from (i.e., sampling) attached sensors as specified in the attributes of the SELECT clause.

A technique suggested for saving power that has been studied in the context of sensor databases is *filtering* and *aggregation* of data along its multi-hop path to the sink node [146, 87]. Given that, in terms of power consumption, computation is much cheaper than communication in a sensor network, considerable gains can be achieved. With this, it is possible to support operations like AVG, MIN or COUNT, providing a more complete SELECT-FROM-WHERE-GROUPBY-HAVING SQL system, as Listing 2.6 depicts.

```
SELECT room, AVG(temperature)
  FROM sensors
  WHERE floor = 4
  GROUP BY room
  HAVING AVG(temperature) > threshold
  SAMPLE PERIOD 20 SEC
```

Listing 2.6: TinyDB query to find out rooms on the 4th floor with average temperature exceeding a certain threshold.

Recently, the idea of sensor databases has been resumed from a different perspective: that of having *each node* as a database [133]. This work partially reflects the advances in non-volatile storage: increased size (on-board, megabyte chips and external SD cards of various gigabytes), ever decreasing cost (a 16GB SD-card can be obtained as of 2014 for €4), and shorter access times. By contrast, communication bandwidth has remained low when retaining the same power budget. The proposed system, Antelope, provides a SQL variant called AQL and implements efficient algorithms for realizing fundamental database operations: creation and removal of relations and indices; insertion, removal and selection of tuples; and joining relations.

While, in general, the database approach is very practical for managing, accessing and retrieving sensor data, its drawback is that it was not conceived to define actuation logic. This approach could be expanded by using triggers, and TinyDB indeed does offer a limited type of actuation, but this usage does not result naturally suitable for WSAN applications.

Macroprogramming

There exist other approaches that attempt to offer a centralized mechanism to define the logic of the WSAN as a whole. In *Kairos* [55], the goal is to extend a *procedural* language like C with certain programming abstractions so that the developer can specify the global behavior of the WSAN through a centralized program. Kairos offers 3 programming constructs: addressing arbitrary nodes, reading and writing variables at nodes, and iterating through the one-hop neighbors of a node. Listing 2.7 presents a centralized Kairos C program to calculate the average temperature of the nodes in a given room, which exemplifies the usage of some of its constructs: in line 3, the list of nodes in the entire network is requested, while in line 8 the `temperature` variable at a named node is read.

Kairos programs are taken by a precompiler that identifies references to the additional constructs and translates them into node-level code that can be compiled by the standard platform compiler to generate node binaries. At runtime, Kairos consists of a run-time system that supports the execution of these special constructs. By means of message-passing, node lists are built and updated, and variable state is updated (which resembles the shared-memory model of Agilla). With these constructs it is simpler to implement a “textbook” algorithm; the authors report a double performance overhead in convergence time and network traffic (and thus power consumption) compared to explicitly distributed versions of the evaluated applications.

```
int getAvgTemperature (int floor_nr ) {
  int sum=0, num_nodes=0;
  node_list network_nodes = get_available_nodes();
  for (node node_ptr = get_first(network_nodes);
    node_ptr != NULL;
    node_ptr = get_next(network_nodes)) {
    if (floor_nr@node_ptr == floor_nr) {
      sum += temperature@node_ptr;
      num_nodes++;
    }
  }
  return sum / num_nodes;
}
```

Listing 2.7: A Kairos C program (adapted from [55]).

In addition to being much more readable than their assembler counterparts, procedural programming languages like C offer a level of expressiveness that is necessary to precisely specify application behavior. Its usage as main paradigm to organize application logic, however, can still be very complex for domain experts. We advocate that a more abstract representation of this logic is necessary.

We next discuss two such approaches centered around *data* flow and *control* flow, respectively. While these are complementary to each other, as we will see, the latter has reached a level of industrial maturity which indicates its acceptance as programming paradigm.

Dataflow

The dataflow architecture provides a programming paradigm that can be very practical in many applications. The central idea behind this architecture is to organize the global application logic by means of a graph that coordinates the flow of data between processing elements.

COSMOS [6] is a framework composed of a macroprogramming language (mPL) and an operating system (mOS) that follows this philosophy. In COSMOS, the processing elements are called *functional components* -collections of computing primitives written in a subset of C- connected with each other through *interaction assignments* -a directed graph that specifies the distributed dataflow between components-. The functional components have typed data inputs and outputs, hence the graph can be statically type-checked. Similarly, in the *Abstract Task Graph* approach [111], programs consist of two parts. The first, *declarative* part specifies the application's tasks and constraints on their distribution (i.e., assignment to nodes) and communication. The second, *imperative* part contains the node-level implementation of the task in a traditional computer language.

The *functional* programming approach also matches the dataflow category. Flask [90] is a system that allows embedding nesC code fragments into Haskell, a language where the primary control abstraction are functions. Flask's runtime environment offers a data collection operation, `fold`, which operates on data flowing from a node's own sensors as well as its child nodes'. The concepts in Flask indeed allow the implementation of a SQL-like functionality, realized in an example application called FlaskDB.

While we consider that analyzing an application from a *data flow* perspective is necessary (and we indeed use one form of it for a particular component, namely that of the event models, as will be presented in Section 3.5), in this work we advocate that the *control flow* view of an application better helps to understand and manage its overall structure.

Workflow

The complementary view to a dataflow model is that of control flow. Here, applications are also defined as a graph with processing elements, but their interconnections represent flow of control. A workflow is defined by the Workflow Management Coalition as "the computerized facilitation or automation of a business process, in whole or part" [68]. [3]

The idea of utilizing workflows to describe application logic is omnipresent in the software development community, which is evidenced by the usefulness of standards such as Activity Diagrams (ADs), from the Unified Modeling Language (UML); the Business Process Model and Notation (BPMN), from the Object Management Group (OMG); or the Business Process Execution Language (BPEL) from the Organization for the Advancement of Structured Information Standards (OASIS). Applications modeled and implemented using these technologies, however, are typically executed using a workflow management system, which were not designed to run within the resource constraints of mote-class wireless sensors.

In [54], we began exploring the idea of moving selected application logic, i.e., parts of a workflow, from its typical execution environment (mainframe and backend servers) towards the network periphery (the gateways or even the sensors and actuator embedded devices themselves). We showed how the normal execution flow of a ubiquitous supply-chain management application can be successfully handled by these devices, thereby offloading the servers from this duty and thus enhancing the system scalability.

To the best of the author's knowledge, the idea of utilizing workflows to define WSAN application's logic was first introduced in a position paper [53]. In that earlier work, we formalized an abstract workflow model and presented a workflow execution algorithm that runs entirely on WSAN hardware. Through an evaluation of existing workflow models available in the industry (which is later discussed in Section 3.1), the simple workflow model proposed in that earlier work was superseded by another workflow model and language, BPMN. The latter has evolved as a de-facto standard, which business analysts (i.e., domain experts) are becoming familiar with.

This line of work has seen interest from other researchers in recent years (e.g., [123, 43, 17, 132]). In [123], Spiess et al. describe a mechanism to identify fragments of a workflow that need to be deployed and executed using WSANs. They propose using a cost function that considers communication and processing cost, among other parameters, to decide which deployment scheme to adopt. These ideas are orthogonal to the approach presented in this thesis, and can thus be used to complement it, e.g. by using ukuFlow's purely in-network execution engine as a deployment strategy. More recently, the EU project *makesense* resumed this idea and began implementing a system that indeed executes application logic in a sensor network [132]. Similarly to our approach, they make a number of extensions to the BPMN specification to accommodate WSAN logic into a BPMN workflow. Their approach, however, still relies on a backend server to control and execute workflows, seeing WSAN nodes as devices where only the so-called *local actions* can be executed. In our work, in contrast, we demonstrate the possibility of WSANs to act as workflow engines, delegating entirely a workflow (or sub-workflow) to the runtime system for its execution.

In [17], Caracaş and Bernauer describe a system with which simplified BPMN workflows can be converted into executable binaries of sufficiently small footprint to work on mote-class devices. These generated binaries are used to reprogram nodes, which either requires physical access to the nodes, or makes use of a dynamic, over-the-air reprogramming mechanism, which in our targeted platforms has severe deficiencies (e.g., unreliable wireless module dissemination and restrictions on module size). This is a major argument in favor of a runtime platform supporting dynamic workflow deployment.

[3] Depending on the domain, workflows are also commonly known as processes, business processes or flowcharts, among others. In this work, we refrain from using these terms exchangeably, adopting simply the term *workflow* instead. This also avoids the confusion with the notion of process established in operating systems' terminology.

2.6 Summary

In this chapter we have presented building blocks that are necessary to realize the vision of pervasive computing. While applications can be realized with different types of embedded devices, in this work we restrict ourselves to low-power, mote-class devices and matching sensors and actuators. In order for WSANs to become mainstream tools, application development must be simplified, which can be achieved through programming abstractions and respective runtime systems. This chapter discusses existing middleware approaches targeting this goal. In general, it is observed that the required computer programming abilities remain quite high: Maté and Agilla offer an assembler bytecode instruction set, Flask requires Haskell. A number of platforms like TinyDB require SQL knowledge; in our experience, and despite the proliferation of web-based applications based on LAMP stacks (Linux, Apache, MySQL and PHP), few programmers posses SQL skills. In addition, many of the presented approaches are conceived as sense only.

In the next chapter we introduce the macroprogramming model of ukuFlow, which adopts many existing ideas for simplifying programming abstractions: it offers a centralized perspective (cf. network- vs. node-level logic) to defining application logic by means of workflow abstractions, employing a node grouping mechanism to define aggregate sensing and actuation behavior.

3 Macroprogramming Workflows with ukuFlow

Programs must be written for people to read, and only incidentally for machines to execute.

H. Abelson and G. Sussman

ukuFlow[1] is a holistic workflow management system (WfMS) for low-power wireless sensor and actuator networks aimed at domain experts. As a WfMS, ukuFlow includes models, tools and mechanisms to support the end-to-end workflow development and execution lifecycle:

- Workflow *modeling* is performed directly by domain experts through a computer-aided software engineering (CASE) tool [63] included as a plug-in into a popular IDE, Eclipse.
- Once completely specified, workflow models need to be *verified* before they can be *converted* into an intermediate representation that is suitable for WSANs. The verification includes checks for, e.g., syntactical correctness and well-formedness. The conversion is a type of compilation that transforms a workflow into a much more compact representation.
- A workflow that has been converted successfully is ready for its deployment. A runtime system offers an interface with which workflows can be *registered* for their execution, available workflows and instances can be *enumerated*, and eventually *unregistered*.
- The ukuFlow runtime system takes care of the workflow *execution*. Immediately after a successful registration, a workflow is scheduled for execution, following the instantiation semantics provided by its developer.

This work seeks to offer a steep learning curve (i.e., be learnable in a short time by the domain expert). For this purpose, ukuFlow features a **mixed graphical/declarative/imperative** macroprogramming model, which is the topic of this chapter. The next section describes the *graphical* approach to modeling workflows, and provides examples of their composition semantics. Section 3.2 describes the data model used in these workflows. In Section 3.3 we explain the *declarative* approach to define node groups. Section 3.4 discusses the actuation/command model. Finally, Section 3.5 presents the details on the generation and filtering of events.

Throughout this chapter, we present examples related to airport operations that elaborate on the case study introduced in subsection 2.3.1. These applications were designed in cooperation with researchers from the Institute of Flight Systems and Automatic Control of the Technische Universität Darmstadt, who are specialized on cockpit systems and air traffic management, and illustrate the potential applications that can be developed using workflow abstractions.

[1] The word 'uku stands for *tiny* in Hawaiian

3.1 Workflow Model

In ukuFlow, applications are defined in terms of simple, arrow and boxes *workflows* that follow a graphical Workflow Definition Language (WDL). The goal of this macroprogramming language is to abstract away internal complexities of WSANs such as routing, grouping, data collection, event detection and action execution, providing a representation of the modeled application that can be understood quickly.

There are two mainstream graphical workflow models that can be used to describe an application: UML's Activity Diagrams (ADs) and OMG's Business Process Model and Notation (BPMN). The virtues of a given language can be evaluated by checking its support for *workflow patterns* [136], a comprehensive catalog of constructs generally accepted in the community as measuring bar. Both ADs and BPMN have their strengths, as well as inabilities to support advanced workflow constructs (discussed, e.g., by Russell et al. in [118] and Wohed et al. in [144], respectively). In principle, however, both notations could be chosen for WSAN workflow modeling. Also, other graphical *notations* exist, such as flow charts, state diagrams and Event-driven Process Chains (EPCs); there exist workflow *languages* for which a graphical notation can be inferred, like the Business Process Execution Language (BPEL) and Yet Another Workflow Language (YAWL); workflow *formalisms* like process algebras and Petri nets; and *domain-specific* workflow models (like the one initially proposed by us in [53]). In many cases, transformations can be found that convert a workflow expressed using one model into another model.

For ukuFlow, we have chosen to adopt BPMN 2.0 [104] as graphical notation. The BPMN specification, however, is very rich and detailed; in particular, workflow –i.e. process– diagrams (one of the three types of diagrams) count with more than 100 modeling constructs (63 types of events, 7 types of gateways, 6 types of tasks, and a multitude of markers customizing these). For WSANs, this requires identifying a minimal *subset* of constructs that suffices to model the targeted applications: this is the goal of the ukuFlow Workflow Definition Language (uWDL) described in this chapter. As such, uWDL does not offer the full *process conformance* level but is rather merely "based on" the BPMN specification.

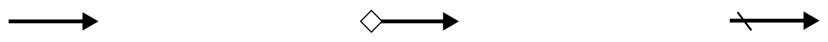
3.1.1 Control Structures

In BPMN, and consequently in uWDL, a workflow is defined in terms of *events*, *activities* and *gateways*, connected with each other by means of *sequence flows* (cf. Fig. 3.1). These elements, to which we generically refer as *workflow elements*, precisely specify how the control must proceed along such workflow. The BPMN specification includes an extensive set of event, activity and gateway types, many of which do not apply to WSAN workflows. For instance, BPMN activities can be of one of 7 types, such as a *user task* (expected to be performed by a human) or a *manual task* (expected to be executed outside of the workflow engine). In uWDL we adopt a subset of BPMN constructs that still allows accommodating traditional applications; later we describe a number of constructs in BPMN that can be emulated by the simpler ones adopted.

(a) events **(b)** activities **(c)** gateways **(d)** sequence flows

Figure 3.1.: Abstract BPMN elements

Similarly to the BPMN specification [104], to explain the semantics of the workflow elements and their interconnections we resort to the notion of a *token* that traverses these elements. (In Chapter 4, we elaborate on this idea to describe the actual operation of the workflow engine.) Workflow elements are connected with each other through *sequence flows* (cf. Fig. 3.2). In its basic, *unconditional* form (3.2a), a sequence flow is used to indicate the direction in which a token needs to be routed through the workflow diagram. In conjunction with some of the gateways, as we will see later, it is possible to use *conditional* (i.e., associated to a boolean condition, 3.2b) or *default* sequence flows (3.2c).



(a) unconditional **(b)** conditional **(c)** default

Figure 3.2.: BPMN sequence flows in uWDL

BPMN Events

In every uWDL workflow there are, at least implicitly, two mandatory workflow elements: the *start* and *end* events (cf. Fig. 3.3). These workflow elements indicate where a particular workflow will begin and conclude, and can be seen as the elements at which tokens are initially created and eventually removed. It is recommended that workflows have one *start event* and only one *end event*. In practice, however, it is possible to have a workflow where the *start event* is implicit, and also allow there to be multiple *end events*. A simple transformation can identify the starting workflow element and add a *start event* in front of it. If there are multiple *end events*, these can be replaced by a single *end event* to which all others converge.

(a) start event **(b)** end event

Figure 3.3.: BPMN events in uWDL

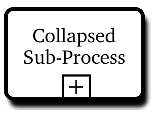
The *start* and *end events* in uWDL unconditionally generate and consume tokens, respectively (i.e., correspond to the BPMN *none* start event and *none* end event, respectively). This means that as soon as a workflow is deployed, the workflow might begin with its execution (this aspect is further elaborated later in Section 4.3.1).

Activities

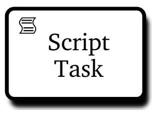
In a BPMN workflow, actual work is performed in activities, which are represented with a rounded box (cf. Fig. 3.1b). Activities can be *atomic* (i.e., can not be further refined) or *non-atomic* (i.e., represent an entire workflow). With this modeling abstraction it is possible to describe workflows at several levels of detail, thus enabling focusing attention to individual parts of an application. A compound activity is represented in a collapsed form with the + sign (cf. Fig. 3.4a). In uWDL, the only atomic activity adopted is BPMN's *script task*, which contains a script that can be directly executed by the WSAN (cf. Fig. 3.4b). (Section 3.4 describes how scripts can be defined.) With the aforementioned elements it is possible to model a *linear* process. This is exemplified in the workflows of Fig. 3.5. Figure 3.5a describes the steps to manage the overall cargo (un)loading workflow; Fig. 3.5b is concerned with aircraft refueling.

Gateways

In most applications, *alternative* execution paths are necessary. This is the objective of BPMN's gateways, which are represented with a diamond (cf. Fig. 3.1c). With these constructs it is possible to model splits, choice, merges and synchronization.

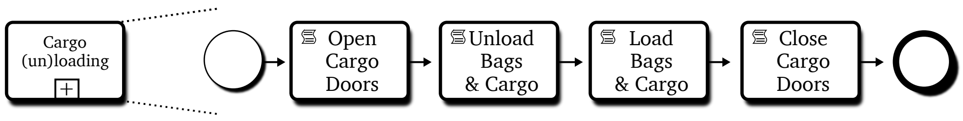


(a) collapsed sub-process activity

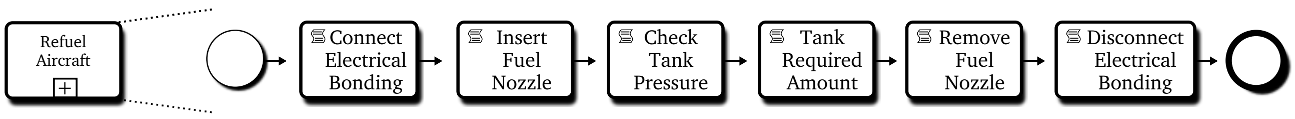


(b) Script task activity

Figure 3.4.: BPMN activities in uWDL



(a) sequential activities for aircraft cargo unloading/loading



(b) sequential activities for aircraft refueling.

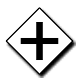
Figure 3.5.: Collapsed sub-processes and expanded representations of linear workflows. Tokens are created in the *start event*, traverse linearly the script tasks, and are removed at the end events.

Parallel Gateway

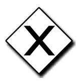
The *parallel gateway* is used to model the *split*, or divergence, of a branch into two or more parallel branches. When a token enters this gateway, all of its outgoing branches are activated (i.e., a token will be generated for them). Parallel gateways are also used to model *synchronization* between several activated branches, typically created previously with a parallel gateway as well. These gateways are represented with the plus symbol inside a diamond (cf. Fig. 3.6a). Figure 3.7 exemplifies these constructs: as soon as a token enters the split parallel gateway, two tokens are created and passed through the outgoing branches *Refuel Aircraft* and *Service Cabin*. Tokens arriving at a synchronization parallel gateway are held until all previously activated tokens have arrived; only then they are consumed and the outgoing branch is activated.

Exclusive Decision Gateway

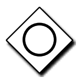
The *exclusive decision gateway*, represented with the x symbol inside the diamond (cf. Fig. 3.6b), is used to model the *exclusive choice* of *one* out of a set of outgoing branches, based on the evaluation of a logical, boolean expression (details about such expressions are given afterwards together with the data model, in subsection 3.2). Each outgoing branch is evaluated at once following a pre-specified order. In addition, one of the outgoing branches can be indicated as *default* path (with the sequence flow as depicted in Fig. 3.2c), which is activated in case none of the other branches' expressions evaluate to true. Figure 3.8 exemplifies these constructs. Since only one outgoing branch is activated (Fig. 3.8a),



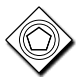
(a) parallel gateway



(b) exclusive gateway



(c) inclusive gateway



(d) event-based gw.

Figure 3.6.: BPMN gateways in uWDL

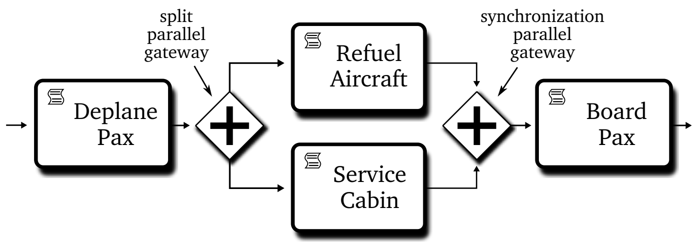
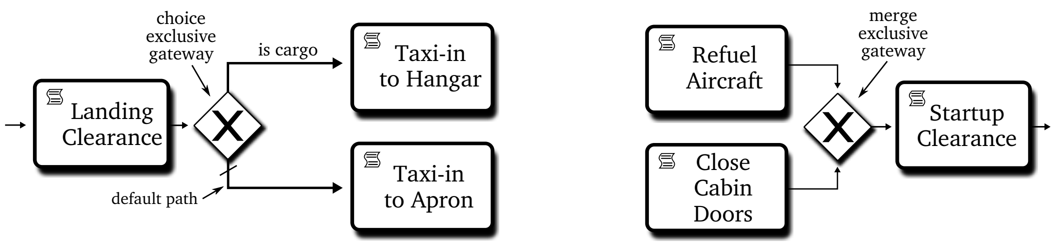


Figure 3.7.: Split and synchronization parallel gateways to parallelize operations performed on an aircraft after deplaning (the procedure to have passengers disembark from the aircraft)



(a) exclusive decision choice

(b) simple merge

Figure 3.8.: Choice and merge exclusive decision gateways to opt for *one* out of a set of options: a) to perform after landing, b) to perform before getting a startup clearance

i.e., receives a token, when a token arrives at a merge exclusive gateway (Fig. 3.8b), it can immediately be routed through the outgoing branch.

Inclusive Decision Gateway

The *inclusive decision gateway*, represented with a circle symbol inside the diamond (cf. Fig. 3.6c), combines the previous types of gateways and is used to model the situation were potentially *more than one* out of a set of outgoing branches needs to be chosen based on the evaluation of several logical expressions. Similarly to the exclusive gateway, one of the branches can be designated as default. Figure 3.9 exemplifies its usage: in a), control is passed to several tasks, depending on the type of aircraft being tracked, in b), control is passed to the *pushback clearance* task only after the tokens (dynamically created at runtime) from the tasks *airways clearance* and *close cargo doors* arrive at the simple synchronization inclusive gateway.

Event-based Exclusive Decision Gateway

Finally, the *event-based exclusive decision gateway*, represented with a pentagon surrounded by a double circle (cf. Fig. 3.6d), is used to model *exclusive choice*. In contrast to the exclusive decision gateway, whose outgoing branches are associated to logical expressions that refer to workflow data available at the time of evaluation, the decision is made according to *events* that happen after a token arrives at the gateway (this is why they are also called *deferred choice* gateways).

Outgoing branches do not have any expressions on the sequence flows. Instead, these gateways are always used together with a number of other workflow elements that concretely specify which events are of interest and must occur in order to activate a certain outgoing branch. In particular, these elements are either *receive tasks* or *timer events*. (Section 3.5 describes in detail the event model used to specify the events of interest and provides examples for these expressions.) The *timer event* is a simple timeout

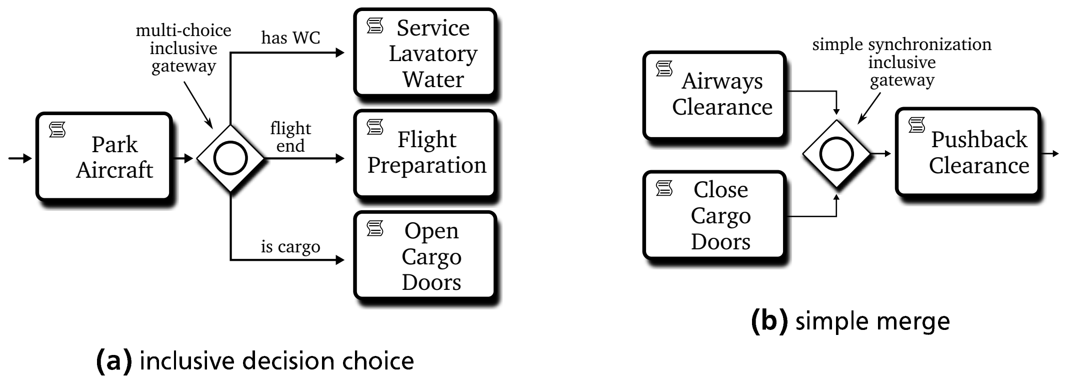


Figure 3.9.: Multi-choice and merge inclusive decision gateways to opt for *some* out of a set of options to execute after an aircraft was parked: in a), depending on the state of the aircraft, some of the tasks (or all) will be executed; in b), parallel tokens created previously will synchronize before continuing to the pushback clearance task

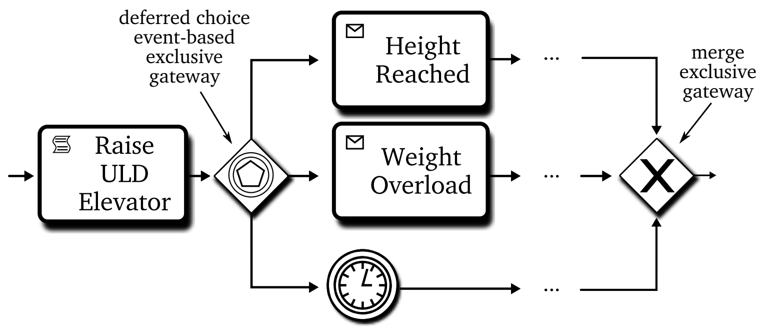


Figure 3.10.: Deferred choice, event-based exclusive decision gateway to opt for *one* out of a set of options based on occurring events, and posterior simple merge

which is activated when none of the events in the other *receive tasks* have occurred, and is analogous to a default sequence flow. Similarly to the exclusive decision gateway, only one of the outgoing branches are activated. Figure 3.10 illustrates its usage: the *raise ULD elevator* task requests the lifting of the unit load device (ULD) container. After this request, the elevator might correctly reach the desired height (and continue with the desired behavior), or might detect a weight overload (and trigger a failure message). Other problems might occur that prevent a correct operation of the elevator, for which a timeout is foreseen. These branches ultimately merge with a simple merge exclusive gateway.

3.1.2 Workflow Concurrency and Looping

An ukuFlow workflow can be seen as a normal computer program which, instead of being executed on a single server, runs on an entire WSAN. By design, an ukuFlow WSAN can host multiple workflows deployed *simultaneously*, each serving an individual purpose.

In many applications, a workflow is modeled as a singleton, i.e., real-world logic that globally relates to a single main entity. In such a system the focus lies on a central object, for instance, a facility management application for a building. In other applications, in contrast, workflows describe logic associated to real-world activities that occur in parallel. There, the workflow focus lies on individual objects, for example in object tracking applications such as aircraft monitoring in an airport management system. In such cases, multiple instances of a workflow need to be executed concurrently.

Very much like in an operating system, and depending on the application, such workflows might have one or more workflow instances at runtime, all running in parallel within the WSAN, or in pseudo-parallel if these workflows (or parts of them) are co-located on individual sensor nodes.

Each of these instances, in turn, will have at least one token, created when the *start event* of a workflow instance is processed. At any point in time during its execution, a workflow instance can have a varying number of tokens being processed. Tokens are eventually removed when they reach the *end event*. A workflow instance continues to exist until it has no more tokens, when it is terminated. A cardinality diagram summarizing these aspects is presented in Fig. 3.11.

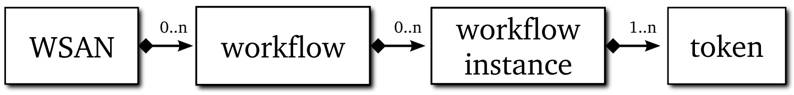
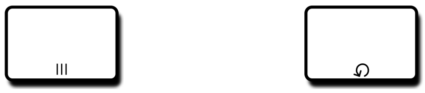


Figure 3.11.: Cardinality of concurrent workflow entities

In addition, workflow instances might have to be executed only *once*, or iterate *multiple* times (i.e., a fixed or an unlimited number of times). BPMN accounts for the concurrency and looping aspects of a workflow through special markers used in conjunction with activities: Fig. 3.12a illustrates the multi-instance marker; Fig. 3.12b the looping marker. The specification of these properties is supported by uWDL as well.



(a) parallelism marker **(b)** looping marker

Figure 3.12.: BPMN concurrency and looping characteristics

3.2 Data Model

While in a workflow the emphasis is placed on the flow of control, data is an important perspective to consider in a WSAN application. The BPMN standard, however, does not provide a concrete model for specifying or accessing data. Instead, BPMN makes arrangements for accommodating different data definition, query and access models. In ukuFlow, this is resolved by a simple, WSAN-conscious data model based on a *name-value pairs* data structure, and a logical language to access its contents. In this model, each workflow instance has its own transient data space where it can store and access data. The data space is *created* when the workflow instance is initialized and allocated resources; remains *accessible* during the instance's execution; and is *disposed* when the instance is terminated. A data space is *shared* across all of a workflow instance's tokens, which can refer to this data at all times. While this feature simplifies the storage needs between tokens, it requires careful use since race conditions could occur.

Data spaces contain application-specific data belonging to a particular workflow instance that resemble standard numeric variables. These data spaces can be accessed and manipulated in the conditional *sequence flows*, as well as in workflow elements such as *script tasks* and *receive tasks*. For this purpose, logical and arithmetical expressions can be used that evaluate to boolean or integer values. For example, an instance for the workflow part presented in Fig. 3.9a could contain data pairs for the names `has_wc`, `current_segment`, `num_segments`, and `airplane_type`. With these, it is possible to realize the informal expressions of the example, as specified in Table 3.1.

Value pairs do not need to be defined nor declared. Instead, the first appearance at run time will trigger its initialization and allocation in the data space. Through additional pre-deployment verifications, it is further possible to calculate the maximum size that data spaces can occupy at runtime.

Table 3.1.: Example data expressions

informal sequence flow label	actual data expression
has WC	`has_wc == true`
flight end	`current_segment == num_segments`
is cargo	`is_cargo == true`

3.3 Scoping Model

While the macroprogramming approach through workflows follows a centralized view, WSANs are inherently distributed systems. Pervasive scenarios are characterized by employing large, and variable, number of nodes during the application lifetime. Many of these nodes are deployed at special positions to carry out particular functions, thus they vary in type and devices (sensors and actuators) attached to them. In many cases, applications need to refer to *collections* of nodes, for instance, to instruct them with a command or to retrieve data from them. The unique identifier typically available in WSAN nodes (e.g., IEEE EUI-64 or IPv6 addresses), therefore, falls short for addressing them.

ukuFlow employs the Scopes network structuring mechanism, previously introduced in Section 2.5.1, which allows defining groups of nodes based on their properties and give them a name -an abstraction we call a *scope*. There are several properties with which a scope can be defined besides the node ID, which might include:

- static properties:
  - node characteristics (e.g., CPU type and speed)
  - populated sensors and actuators
  - position (if node is stationary or position is pre-programmed)
- dynamic properties:
  - sensor and actuator's current data
  - battery level
  - position (if node is mobile or position is inferred)

Scopes can be specified through a *declarative* language, which facilitates their usage for non-experts[2]. In the same way as data expressions, a scope specification can refer to node properties and contain predicates about them. As an example, we can create a scope that groups nodes with magnetometers on an airport's apron by requiring that the node's geographical position lie inside a polygon defined by (x_i,y_i) coordinates and that the node is equipped with a magnetometer, whether on-board or external (cf. Fig. 3.13). For this we would use the expression in Listing 3.1.

The next two sections present how declarative scope expressions are integrated and exploited in the ukuFlow macroprogramming model.

[2] The Scopes specification syntax can be found at `www.dvs.tu-darmstadt.de/research/scopes/syntax.html`.

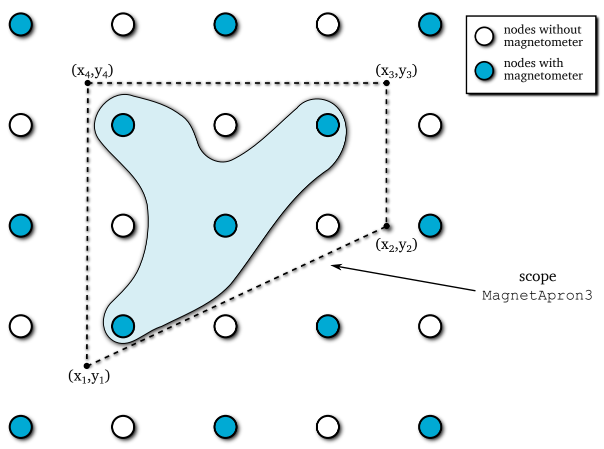


Figure 3.13.: Geographical scope

```
CREATE SCOPE MagnetApron3 AS (
  EXISTS SENSOR_MAGNETOMETER AND
   IN_POLYGON((x1,y1), (x2,y2),
              (x3,y3), (x4,y4)
   )
)
```

Listing 3.1: Example scope specification

3.4 Actuation Model

In ukuFlow's WDL, actuation is performed in script tasks through a series of *statements*. Figure 3.14 summarizes the types of statements mentioned.

In some script tasks, calculations need to be performed, e.g., in order to first find out the exact parameters to other statements. For this purpose, ukuFlow includes the *computation statement*, which allows mathematical expressions to be performed on workflow data. These computation statements read and write data pairs from/into the workflow instance's data space.

There is a rich variety of actuation types that a WSAN can invoke to control and influence its environment, including:

- turning LEDs on/off,
- opening/closing a gate or switching a voltage through a relay,
- setting a servo to a particular position,
- opening/closing a solenoid valve to a given degree,

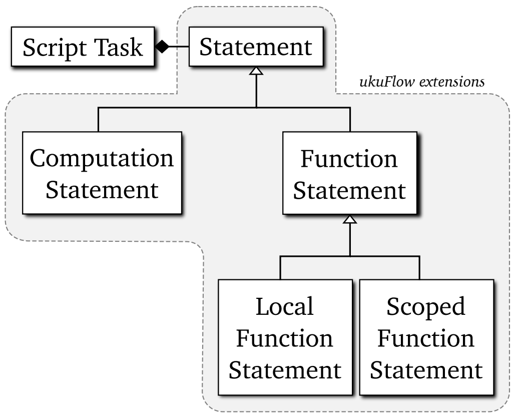


Figure 3.14.: ukuFlow extensions to BPMN for script tasks

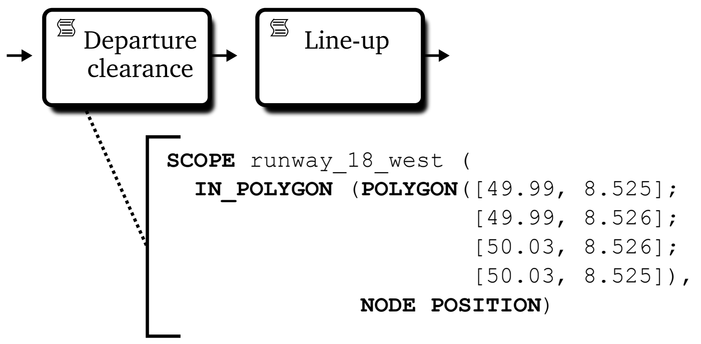
- driving a motor with a certain rotational speed, and
- controlling a sounder/buzzer, among others.

A WSAN node can instruct the aforementioned actuators with an analog signal, obtained with a digital to analog converter (DAC), or through a digital interface, like I^{2}C, UART or SPI. Normally, the functionality of these actuators is implemented in specialized operating system drivers, which are written by the device developers.

Together with actions affecting the real world, others exist that mainly consist in *communicating* information (notifications and alerts). So are first devices that can directly send short emails, tweets, and IRC messages, issue Telnet, FTP, and other UNIX-like commands, which are usually forwarded through a serial port to a computer connected to the Internet.

All of these actions are exposed by operating system libraries and can be invoked through a *name*. Typically, the actions need to be parameterized, passing constants or variables that refine their behavior. Within ukuFlow, these types of actions are called *function statements*.

Depending on the application, the actuation can be requested to be executed at an arbitrary location, or alternatively a group of nodes can be specified by means of a *scope* name. In the former case, the WSAN node running the workflow will execute the action, which is called a *local function statement*. Such statements are prefixed by the `local` keyword. In the latter case, only WSAN nodes that belong to the specified scope will execute the action, which is called a *scoped function statement*. Such statements are prefixed by the `@<scope_name>` identifier. The actual specification of the scope used is indicated in a BPMN element called *text annotation*. These text annotations can be associated to any element in the uWDL workflow, but should ideally be connected to the script task where the scoped function statement takes place. The workflow should contain only one specification for every scope identifier. An example of this annotation is presented in Fig. 3.15. There, the *Departure Clearance* script task contains a scoped function statement that refers to the scope `runway_18_west`, in this case specified as all nodes contained in the given geometrical area presented in Fig. 3.16.



```
SCOPE runway_18_west (
  IN_POLYGON (POLYGON([49.99, 8.525];
                      [49.99, 8.526];
                      [50.03, 8.526];
                      [50.03, 8.525]),
             NODE POSITION)
```

Figure 3.15.: Text annotation with scope specification

The syntax for function and computation statements is presented in Listing 3.2 and 3.3, respectively. An example of their usage is provided in Listing 3.4, an application to control an airport's runway lighting. First, a node's humidity sensor is read; and its value (in percentage) is reduced to an integer between 0 and 10. Later, the result is used to calculate the light flashing rate. Finally, those nodes in the `runway_18_west` scope are instructed to execute the `flash_lighting` function statement with the specified parameter.

```
local [variable=] <function_name>  <parameters>;
@<scope_name>     <function_name>  <parameters>;
```

Listing 3.2: uWDL function statement syntax

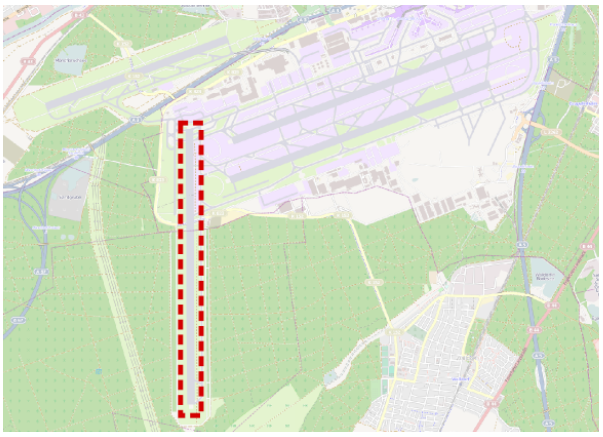


Figure 3.16.: Geographical scope selecting nodes in Frankfurt airport's west runway (nr. 18)

```
variable = <expression>;
```

Listing 3.3: uWDL computation statement syntax

```
num_blinks = SENSOR_HUMIDITY_PERCENT;
num_blinks = $num_blinks / 10;

flash_rate = 60 * $num_blinks;
@runway_18_west  flash_lighting  $flash_rate;
```

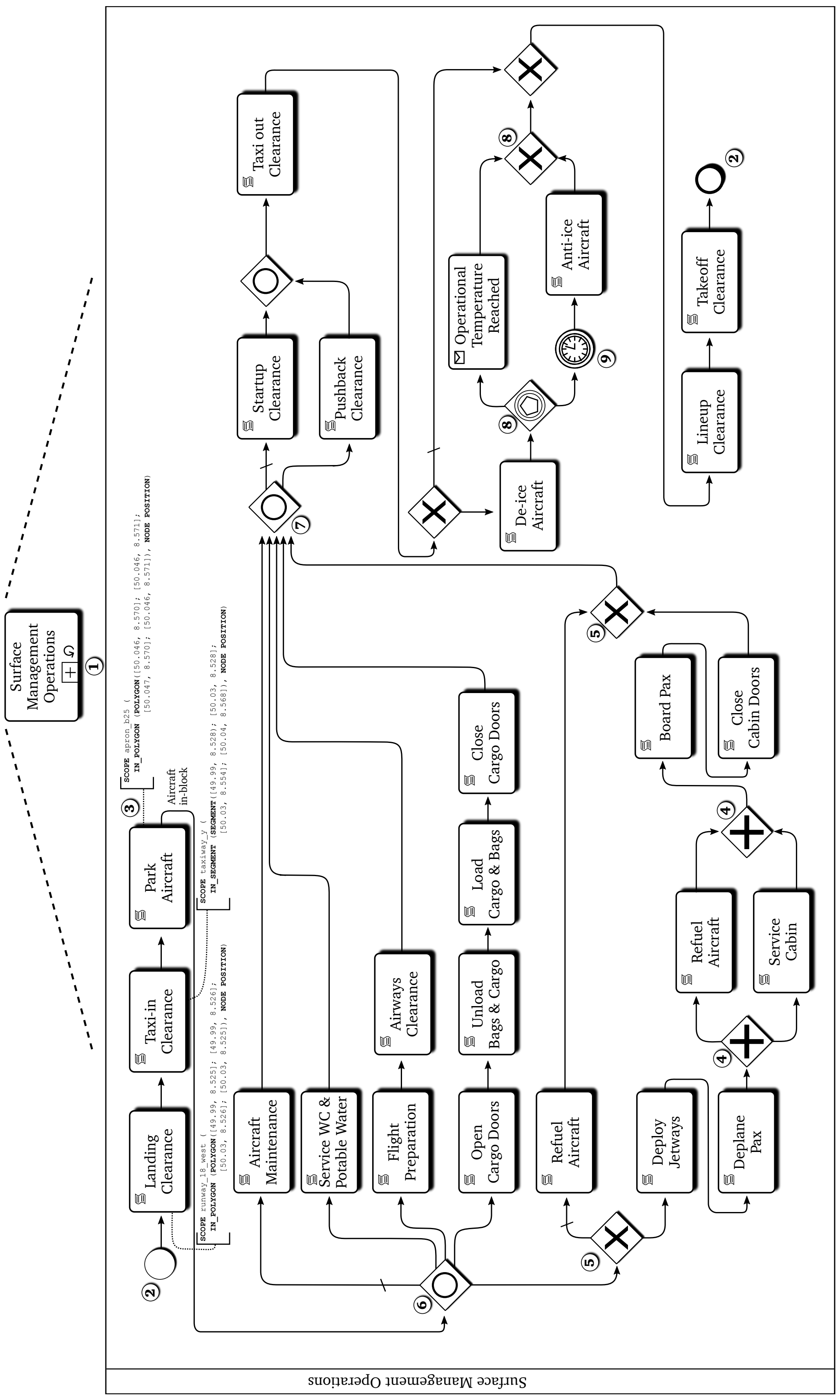
Listing 3.4: uWDL function statement example

To summarize, a script task is a particular type of BPMN activity that includes a script composed by statements which are executed in an explicit sequential order. The invocation of a script task is simple, since no parameters are required. Instead, tasks are customized through the workflow instance data space. A task's statement can read or write into this space, and thus affect the actual execution of following workflow elements.

3.4.1 Putting it all Together

We now present the design of a wider application for *surface management operations* in an airport setting. The workflow, presented in Fig. 3.17, models the complete set of states an aircraft goes through when stopping at an intermediate airport.

First, the entire process (①) is represented as a single, collapsed task (+ marker), expected to run repeatedly (through the looping marker, ↻). The expanded view of the process reveals 25 activities (script tasks), connected through 12 gateways and using multiple scopes. As usual, the process has one *start* and one *end event* (②). Some statements within activities refer to individual scopes. For instance, the task *Park Aircraft* is annotated with the specification for scope `apron_b25` (③). (For the sake of clarity, not all scope specifications are included in the diagram.) The process includes the various gateways available in uWDL. Both a split parallel gateways and its counterpart for synchronization (④) model the parallelism between refueling an aircraft while the cabin is being serviced. A choice exclusive gateway and its counterpart for merging (⑤) are used to choose between a cargo aircraft (default path) or a passenger aircraft, a decision made through static process data. A multi-choice inclusive gateway (⑥) is used to parallelize the execution into multiple paths depending on the aircraft/flight type (e.g., some aircraft might need servicing the WC while others not, some might need to load/unload cargo, etc.). Furthermore, notice how the counterpart for synchronization (⑦) is also used for creating multiple choices



Surface Management Operations
Surface Management Operations
Landing Clearance
Taxi-in Clearance
Park Aircraft
Aircraft in-block
SCOPE runway_18_west (IN_POLYGON (POLYGON([49.99, 8.525]; [49.99, 8.526]; [50.03, 8.526]; [50.03, 8.525]), NODE POSITION)
SCOPE taxiway_y (IN_SEGMENT (SEGMENT([49.99, 8.528]; [50.03, 8.528]; [50.03, 8.554]; [50.04, 8.568]), NODE POSITION)
SCOPE apron_b25 (IN_POLYGON (POLYGON([50.046, 8.570]; [50.046, 8.571]; [50.047, 8.570]; [50.046, 8.571]), NODE POSITION)
Aircraft Maintenance
Service WC & Potable Water
Flight Preparation
Airways Clearance
Open Cargo Doors
Unload Bags & Cargo
Load Cargo & Bags
Close Cargo Doors
Refuel Aircraft
Deploy Jetways
Deplane Pax
Refuel Aircraft
Service Cabin
Board Pax
Close Cabin Doors
Startup Clearance
Pushback Clearance
Taxi out Clearance
De-ice Aircraft
Operational Temperature Reached
Anti-ice Aircraft
Lineup Clearance
Takeoff Clearance
1
2
2
3
4
4
5
5
6
7
8
8
9

Figure 3.17.: Complete *surface management operations* workflow for handling aircraft stops

(i.e., it has multiple incoming and outgoing branches). We will come back to the topic of model compilation to deal with this case. Finally, the workflow has an event-based gateway (⑧). This is used to detect the aircraft temperature after execution of the de-icing procedure. In case the operational temperature (where it is safe to fly the aircraft) is not reached after a certain time (modeled with a *timer event*, ⑨), a second anti-icing procedure is executed. These two alternative branches join with a normal merge exclusive gateway.

3.5 Event Model

As mentioned when describing BPMN gateways, during the execution of a workflow not all decisions are made based on previously existing data. In many situations it is more adequate to pass control to a branch depending on the detection of a particular *event*. In this section we describe the concept of event used in this work, and explain the event model that is employed to specify expressions used in uWDL's *receive tasks*.

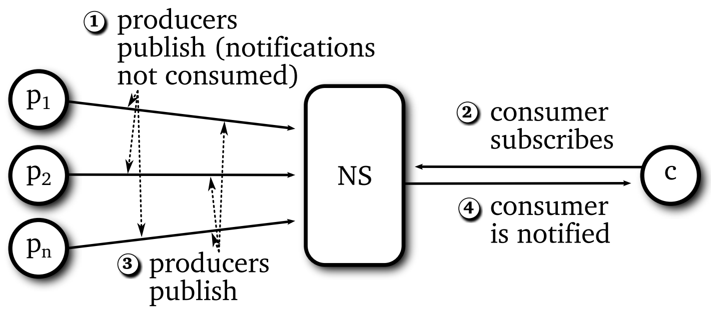
3.5.1 Event Concepts

As presented in [15], there are several definitions of an event. Chakravarthy et al. [21, 22] define it as an instantaneous, atomic (happens completely or not at all) occurrence of interest at a point in time. These events are known as *status events*. Chakravarthy et al. identify several phases when operating with events, starting with the *event occurrence* in the physical world, and later *event detection* and *signaling* phases (in the digital world). In contrast to Chakravarthy et al., M. Chandy emphasizes that events are *significant changes* in the state of the universe [23]. This implicitly relates an observation with an expectation or another, previous, observation. We refer to these as *change events*.

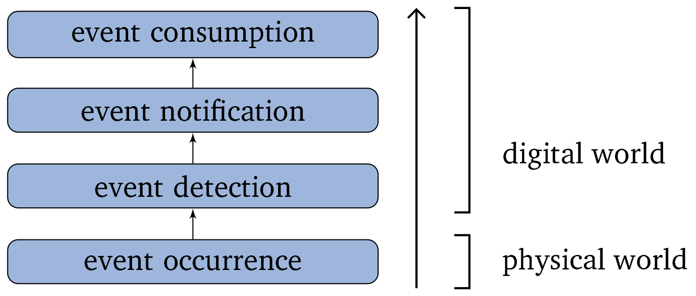
The importance of distinguishing between an *event* and an *interest* on an event follows from these two first definitions. Assuming there is an entity (which could be a software component or a person) interested in an event, the latter is packaged into a message, in which case we speak of an event *notification*. In traditional publish/subscribe systems, event producers are connected with event consumers by means of a *notification service*. Figure 3.18a shows this traditional interaction, where, on one side, data producers (p_i) publish events to the notification service (NS, ①), while on the other side event consumers (c) *subscribe* to, i.e., express their interest in, events (②). The detected events (③) are converted into meaningul, higher level events, and are finally used to *notify* (④) the interested parties. Figure 3.18b presents these traditional phases.

The decoupling between producers and consumers offered by the notification service, however, is costly in WSANs, since published events that go non-consumed require sampling sensors and potentially routing data through the network, the two most energy-expensive operations. The definition of an event from Hinze et al. [66], however, nicely complements the previous two, noting that by considering *time* as an integral part of the state of the universe, two observations that yielded the same values at different times can also be considered events. This enables a uniform modeling, where subscribers indicate a series of parameters about how the data for the events should be acquired (e.g., sampling frequency and patterns). Following this concept, we simplify the publish/subscribe mechanism (cf. Fig. 3.19a): it is only after consumers have subscribed to an event (①) that data producers acquire the data (②) and publish the events (③), with which consumers are later notified (④). We extend the event processing phases with a *data acquisition* step, where corresponding sensor nodes are tasked to sample the necessary raw data as specified by the currently interested consumers (depicted in Fig. 3.19b).

The raw data stemming from the WSAN nodes and their sensors can be used to form an event that is typically denominated a *simple*, or *primitive* event. These are events that typically include the time



(a) publish/subscribe mechanism



(b) event processing phases

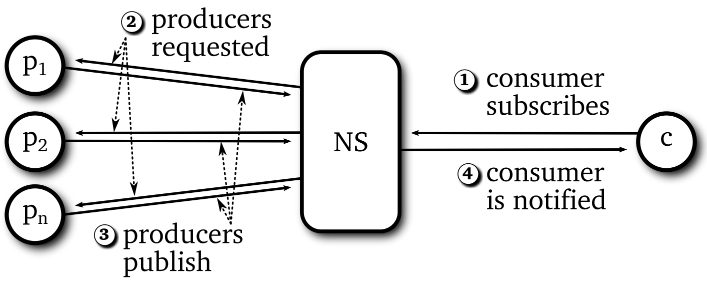
Figure 3.18.: Traditional publish/subscribe approach to dealing with events

of creation and the actual magnitude measured by the sensor, among others. Events can be further processed to determine whether they fulfill certain properties, an operation known as event *filtering*. When the processing is performed on multiple events, we speak of a *composite* event. Event generators, filters and composers are generalized as *event operators*; the different ways in which these operators are connected with each other to produce and detect more elaborated events is normally specified in an *event algebra*. For uWDL, we have adopted a simple and powerful algebra for operating with events in WSANs that, as we will see, enables minimizing the power consumption of the network nodes.

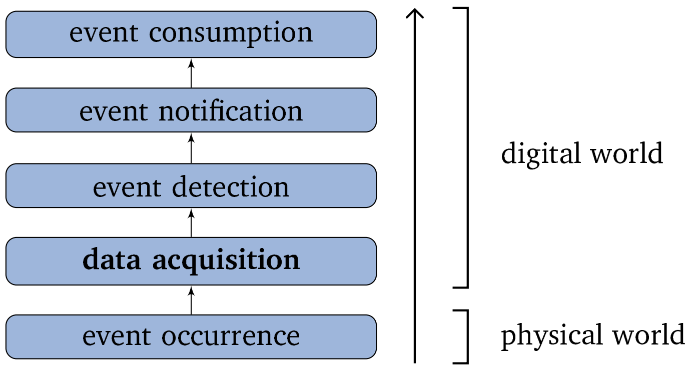
Events in ukuFlow are understood as occurring at their time of detection, an interpretation known as *point semantics* [22]. This is opposed to the *interval semantics* [44], which consider the occurrence interval of the event. Although that interval semantics model enables more sophisticated semantics for composition operators (e.g., sequences), it comes at the price of an additional system complexity, and is subject of investigation in future work. As a result, each event in uWDL is associated with a timestamp. The distributed nature of WSANs invalidates the assumption of the availability of a perfectly synchronized global clock at each node. However, time synchronization protocols exist that enable an average accuracy of around $\delta = 20\mu s$ [45], which is sufficient for many sensor network applications. In practice the developers must consider, thus, that a timestamped event can be offset by $\pm\delta$.

3.5.2 Event Generation

In order for events to be detected, the nodes acting as event sources must be tasked to begin with the necessary generation (i.e., sampling) or computation of data. This data forms the bottom-most elements of a (more elaborate) event. We model this task through *event generators*.



(a) acquisitional mechanism



(b) extended event processing phases

Figure 3.19.: WSAN-oriented approach to events

Event Instances

Event generators produce simple event instances composed of a fixed number of fields: timestamp, ID of the event operator that generated it, data source, the actual magnitude measured or obtained from that source, source node ID, and a scope identifier to which the node belongs and through which the event was requested (cf. Table 3.2). The data source can refer to a node's sensors, the node's hardware state (including its clock or battery), or other node-computable information. As we will see in the next Chapter, limiting an event's contents to these bare essential fields allows it to typically fit in a single radio message, which simplifies the routing protocols considerably and therefore also the power consumption of the nodes.

Table 3.2.: Example simple temperature event with payload 28 degrees Celsius

field	**value**
EVENT_TYPE	simple
EVENT_OPERATOR_ID	3
SOURCE	SENSOR_TEMPERATURE_CELSIUS
MAGNITUDE	28
TIMESTAMP	18-03-2013 10:24:40
ORIGIN_NODE	28
ORIGIN_SCOPE	heaters

Temporal Properties

In essence, there are two types of event generators: *non-recurring* and *recurring*. Non-recurring generators are used when data is requested only once (i.e., to create a single instance of an event). An event can be produced immediately after the subscription to the event is issued (called *immediate* event generator); at a specified, fixed time (*absolute*); after a certain time offset elapses (*offset*); or in conjunction with the occurrence of another event (*relative*).

More typically, however, an event generator needs to produce data in recurring fashion (i.e., to create many instances of an event). This recurrence can take place a certain number of times, or repeat forever until explicitly stopped.

For some special modalities and physical phenomena, it is important to be able to sample sensors following particular patterns. Such patterns can be as simple as sampling once regularly at a certain interval (called *periodical*), or more elaborated, following a certain *pattern* or obeying a given *function* that can be discretized.

Patterns can be specified in uWDL as a time interval that is split into a sequence of slots; the slot sequence is specified with a bit string where a 1 denotes a slot in which an event must be generated and `0` the opposite.

Functional generators are specified with the function name, its specific parameters, and the overall evaluation period and interval needed to discretize it. The gaussian (normal), chi-squared and Pareto distribution functions are included in uWDL. The chi-squared distribution function is useful, for instance, when using an ultrasound pulse transmitter and microphone to measure distance, since the expected bounce times are known in advance. The gaussian function is useful in general situations where real-world phenomena are bursty and a higher precision is required around the event occurrence; an example of its usage is presented in Fig. 3.20 with parameters $\mu = 10s$, $\sigma^2 = 1$ and $a = 5$.

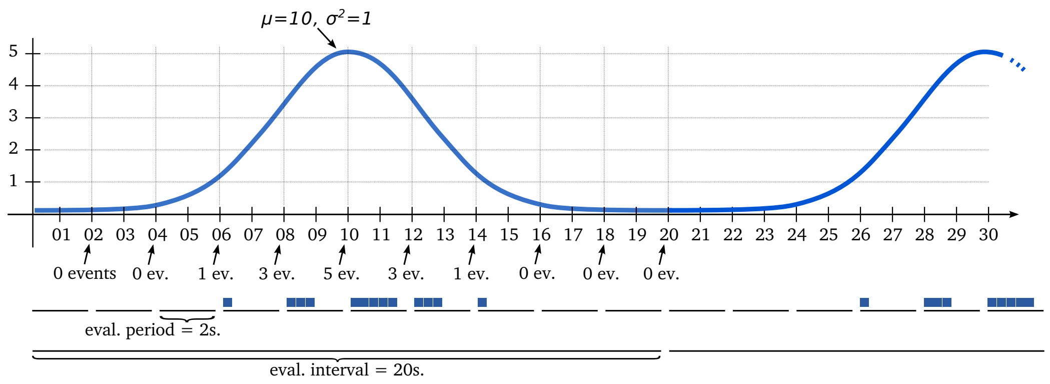


Figure 3.20.: Example of a *gaussian* functional event generator

Spatial Properties

So far, we have discussed the *what* and *when* to generate events. The other important aspect about event generation is *where*, that is, which nodes should produce the data. In ukuFlow, this is tackled also through the Scopes approach. There are three alternatives when specifying the spatial properties. First, events might be generated by a single, master node, such as the scope's root node. This is useful, for example, for *timer events*, where a special node in the network acts as time synchronizer. Second, events might be generated at a subset of the nodes in the network by associating the event generator to a scope identifier. For this purpose, membership expressions, as illustrated in Fig. 3.15, can refer to node properties and their status. Finally, the events might be generated at all nodes (through the *world* scope). This is the most expensive case, but useful and necessary under certain scenarios.

Example Usage

A well-defined specification of event generators is provided through a declarative script language, which we exemplify next:

- Generate temperature data once now from all nodes in the WSAN (non-recurring, immediate temperature generator):

```
IMMEDIATE_EG SENSOR_TEMPERATURE_CELSIUS
```

- Generate a light event once, on Christmas eve, from nodes in the west runway scope (non-recurring, absolute generator):

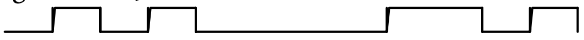
```
ABSOLUTE_EG SENSOR_LIGHT_RAW 2011-12-24 18:00:00 @s runway_18_west
```

- Generate temperature once, after 2 hours from all nodes (non-recurring, offset generator):

```
OFFSET_EG SENSOR_TEMPERATURE_RAW 2:00:00
```

- Generate humidity every 20 seconds starting now and repeating only 10 times (recurring, periodic generator):

```
PERIODIC_EG SENSOR_HUMIDITY_PERCENT ^0:20 x10
```

- Generate temperature events using the provided pattern with the overall sequence length of 30 seconds, and repeating an infinite number of times from nodes in the `runway_south` scope (recurring, patterned generator):
  Pattern: 

```
PATTERN_EG SENSOR_TEMPERATURE_FAHRENHEIT p010100001101 ^0:30 @s runway_south
```

- Generate light following a gaussian distribution with parameters mean $\mu = 0{:}10$ seconds, variance $\sigma^2 = 1s^2$ and $a = 5$ max events, evaluate every 0:02 s., and use an interval of 0:20 s (recurring, temperature generator following a normal distribution):

```
FUNCTIONAL_EG SENSOR_LIGHT_RAW GAUSSIAN_DISTRIBUTION m0:10 v0:01 a5 ^0:02 i0:20
```

The classes of generators and their relationships are depicted in Fig. 3.21. Note that it is possible to specify similar event generation behavior in different ways, for example, a non-recurring event generator produces the same data as a periodic one that repeats only once; a periodic event generator with period, e.g., 10 seconds, is similar to a patterned generator with the pattern `00001` and a slot length of 2 seconds, and so on. Which specification is used depends on user preference.

The event generators we discussed produce individual, low-level events. These are usually referred to as simple events. They can be combined by applying other event operators as described in the next section.

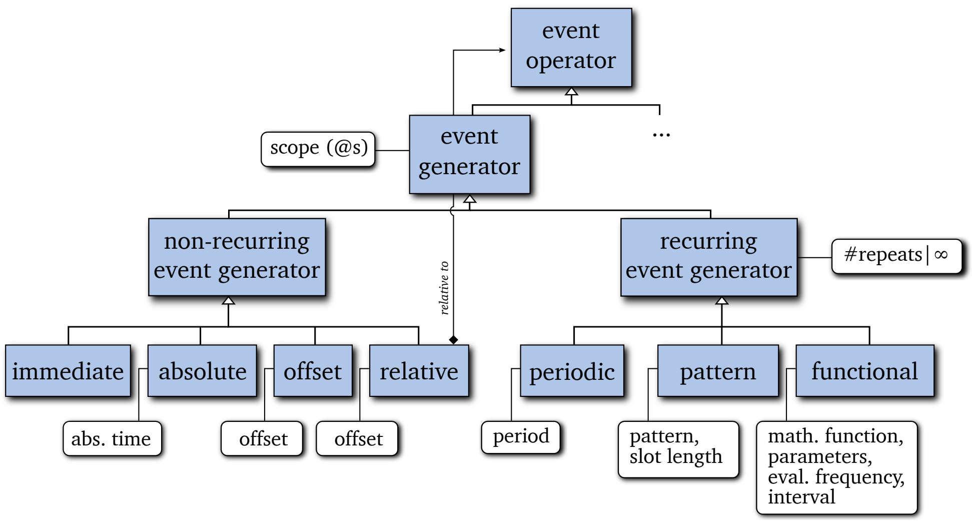


Figure 3.21.: The ukuFlow event operator hierarchy, section about event generators. The leaf elements are the concrete event generators; rounded boxes depict their attributes, the black diamond depicts a *reference* relation.

3.5.3 Event Filtering and Composition

In most situations, the raw events resulting from the generators are too low-level for a decision to be made by an event-based gateway. Commonly, a more elaborated event detection is necessary, for instance to confirm peaks or smooth out outliers. This is achieved by means of event *filters* and *composers*.

In contrast to the event generators, which (among other things) contain a scope identifier to specify the exact subset of nodes that should produce events, these more complex event operators do not require such a parameter. Instead, these might perform their operations at other nodes of the network, depending on its type and nested expressions. In general, it is best to perform these operations in the network as close to the producer nodes as possible, in order to minimize the overall energy consumption. For the time, assume that all event filters and composers are deployed in a centralized fashion; in the next chapter we will describe a mechanism to control the event operator placement.

Event Filter

An event filter is conceptually similar to a function f that takes events as input and, assuming a positive evaluation, outputs another event, where $f : \mathbb{E}^n \rightarrow \mathbb{E}$ and $\mathbb{E}$ is the set of all events. An input to a filter, thus, can be events coming directly from a generator, or other filtered or composed events.

Simple event filters are stateless functions that include a set of *constraints* in the form of logical expressions which apply to the contents of the input event, such as its magnitude or source node ID. In this way we can specify that, e.g., temperature events interesting to the decision are those with values lower than 5° or greater than 45°. An example expression takes the following syntax:

```
SIMPLE_EF  [ SOURCE == SENSOR_TEMPERATURE_CELSIUS &&
            ( (MAGNITUDE < 5) || (MAGNITUDE > 45) ) ] (<eo1>)
```

The last element of the simple filter, <eo1>, refers to the event operator from which this filter consumes its input. This could be instantiated with an actual expression such as a periodic event generator that outputs temperature events. Simple filters are thus useful to detect status events and act upon them.

Note that the output of this filter is another event with exactly the same content as the original event (including the original timestamp), and solely the event operator ID is modified to reflect the ID of the simple filter.

Event Composers

In contrast to simple filters, event composers accept, or might require, multiple inputs in order to generate their output. The literature presents numerous approaches and models to compose events into more complex ones (e.g., [28, 95, 22, 21, 36]), each motivated by a variety of applications such as healthcare, logistics, supply chain management, smart cities, and social networks. The types of operations also vary widely, from aggregation, correlation, logical operations, and sequencing, among others. However, the targeted platforms for which these operators were conceived (from cloud environment, through high performance servers to stand-alone, embedded servers) are typically orders of magnitude more powerful than WSAN environments, in particular with regards to the available memory and processing capabilities.

We have investigated the feasibility of a range of composite event operators on low-power sensor nodes, and selected a subset that enable the realization of useful expressions for sensor network applications for their integration into uWDL. We present these operators in Fig. 3.22.

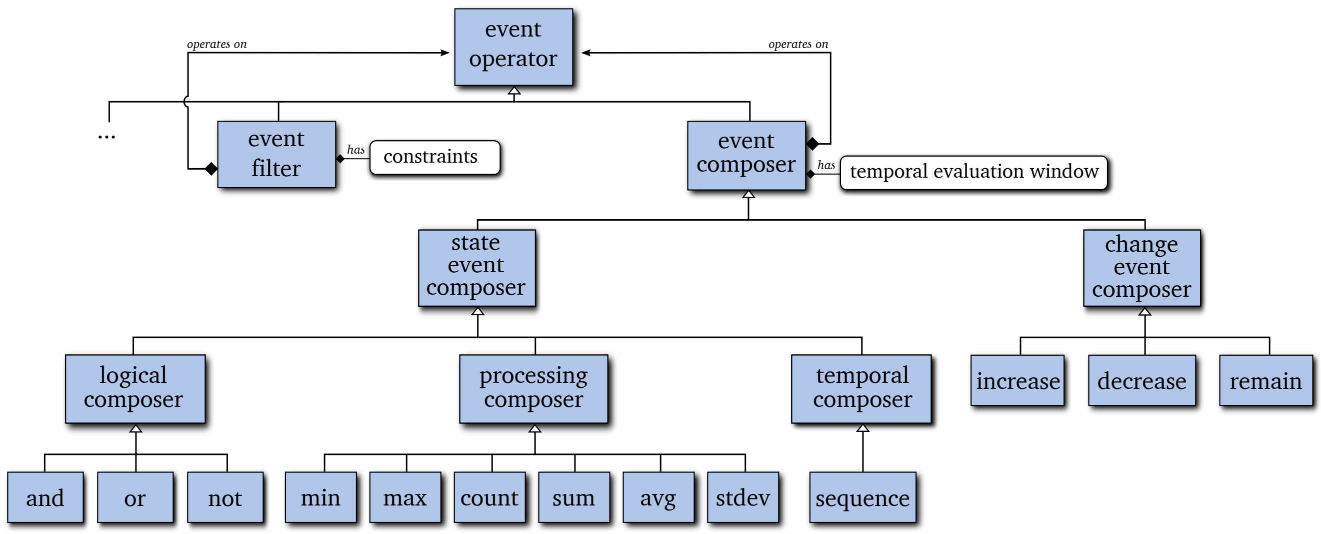


Figure 3.22.: The ukuFlow event operator hierarchy, section on event filters and composers. Both types are applied to an input set of nested event operators.

Event composers are split into two groups: *state* event composers and *change* event composers. While both operate on a user-specified *temporal evaluation window*, the former are used to report observations about the current or aggregated state of the environment, and answer questions of the form "*what is the average temperature in the rooms at the north side today?*", while the latter refer to explicit changes in the environment, addressing questions similar to "*did the temperature increase by more than 10 °C in the last hour?*".

State event composers are, in turn, subdivided into logical compositions, processing compositions and temporal compositions. A logical composition is associated to a logical expression such as `and`, `or` or `not`, producing an output when such expressions evaluate to true. Processing compositions typically operate over a series of individual, simple events, to identify outliers (`max` or `min`), or calculate aggregations (`count`, `sum`, `avg` or `st.dev.`). Temporal compositions make emphasis on the timestamps of the input events, as is the case of the `seq` operator which produces an output event if two input events occur in a given sequence.

Finally, change event composers look at a sequence of events over a period of time, but additionally analyse the observed magnitudes. Figure 3.23 presents two sequences of events within a time interval. For these operators, the event detection criteria to produce an output event depends on the type of change composer. For the increase event composer it is determinant whether the sequence is monotonically increasing and if the magnitude span, i.e., the difference between the maximum and minimum values in the interval, exceeds a user-specified threshold (if the magnitude span is below this threshold, it is considered uninteresting for the user). Analogously, the decrease event composer checks if the sequence is monotonically decreasing and the difference exceeds the threshold. For the remain composer, in contrast, it is not relevant if the values are monotonic; instead it evaluates if all the observed magnitudes are within the user-specified delta.

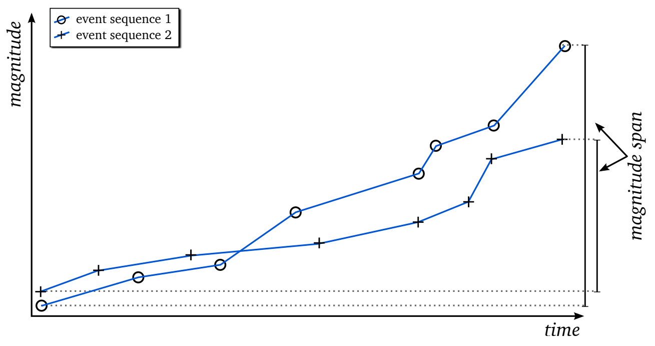


Figure 3.23.: Two examples of monotonically increasing event magnitudes and their magnitude span

When the conditions are met, or the computation of aggregations is ready, event composers generate events that reflect or summarize the situation under which the composite event took place. Following Madden et al.'s notation [87], we distinguish between *exemplary* and *summary* filters. For instance, the *exemplary* nature of the `max` operator enables including the magnitude and source node address (if available) and source scope of the individual input event that led to the resulting composite event; the *summary* nature of the `avg` operator creates a new magnitude value resulting from the aggregation or processing of multiple individual values.

In contrast to simple filters, the timestamp field is updated to signal the time of the composite detection, and will typically be greater or equal to the timestamp of the last contributing input event (recall that we resort to event point semantics). Similarly to the behavior of the simple filter, however, the generated composite events contain the event operator ID of the composite filter. Note that the model does not provide explicit support for *data provenance*, i.e., help to determine the origins and causes of an event, since attaching every source event to the output event would make their size prohibitively large. While compression techniques can be applied for this purpose, this topic requires further investigation.

Example Usage

In the following, we demonstrate the expressiveness of uWDL's composite event operators:

- Detect the occurrence of two events in a time window of 30 seconds, regardless of their temporal order (logical conjunction event composition):

```
AND_EC ^0:30 (eo$_1$, eo$_2$)
```

- Detect the occurrence of either one of two events in a time window of 60 seconds (logical disjunction event composition):

```
OR_EC ^1:00 (eo$_1$, eo$_2$)
```

- Identify the non-occurrence of an input event in a time window of 60 seconds (logical negation event composition):

```
NOT_EC ^1:00 (eo$_1$)
```

- Aggregate events within a time window of 60 seconds (count processing composition):

```
COUNT_EC ^1:00 (eo$_1$)
```

- Detect occurrence of two events in sequential order in a time window of 2 minutes (sequence temporal composition):

```
SEQUENCE_EC ^2:00 (eo$_1$, eo$_2$)
```

- Detect an increase in the magnitude of 200 units in a stream of events, in a window of 30 seconds (increase change event composer):

```
INCREASE_EC ^0:30 r200 (eo$_1$)
```

Event Operator Classification

In Table 3.3 we provide a classification of the event operators included in ukuFlow. We consider the following criteria:

- The *arity* of an event operator refers to the number of input event types it accepts.
- The *processing cardinality* denotes both the number of events (of the types specified as allowed) taken as input and generated as output. For those event operators that function in time windows (i.e., recurrent event generators and composers), the processing cardinality refers to the event exchange relationship within each time window.
- The exemplary/summary property, as discussed in the previous section, defines whether the event operator returns a representative of the set of input events, or is computed over that set.
- Finally, for the point semantics used in the event algebra we specify the moment when the output event is timestamped, distinguishing between event creation time (i.e., when the event is generated) and when a composite event is detected.

Table 3.3.: Classification of ukuFlow's event operators

	Event Operator															
	Event Generator	Event Filter	Event Composer													
			Logical			Processing Function						Temporal	Change		
			`and`	`or`	`not`	`min`	`max`	`count`	`sum`	`avg`	`stdev`	`seq`	`inc`	`dec`	`rem`
Arity	0	1	2	2	1	1	1	1	1	1	1	2	1	1	1
Processing cardinality	0:(1..n)	1:1	n:1	n:1	0:1	n:1	n:1	n:1	n:1	n:1	n:1	2:1	n:1	n:1	n:1
Exemplary/ Summary	n/a	E	S	S	S	E	E	S	S	S	S	S	S	S	S
Creation/ Detection timestamp	C	C	D	D	D	D	D	D	D	D	D	D	D	D	D

3.5.4 Combining Event Operators

The previous subsections introduced the individual event operators modeled in ukuFlow. As indicated, these event operators can be combined (i.e., nested) with each other recursively, building a more sophisticated overall expression that refines the final event expected. The obtained combination is known in the literature as an *event detection graph* (EDG) [22, 13] or event processing chain [36].

Event expressions can be written in a compact format in a single line. Furthermore, for clarity, it is also possible to associate identifiers with sub-expressions that can be referred to from other, higher-level expressions. A reserved identifier, `TOP`, is used for the top-level expression. We illustrate combinations of several event expressions using the event operators described previously.

- Count the number of nodes in the scope `windows`:

```
COUNT_EC ^0:30 ( IMMEDIATE_EG NODE_ID @windows )
```

- Detect room with highest CO_2 concentration (above 600ppm) in first floor:

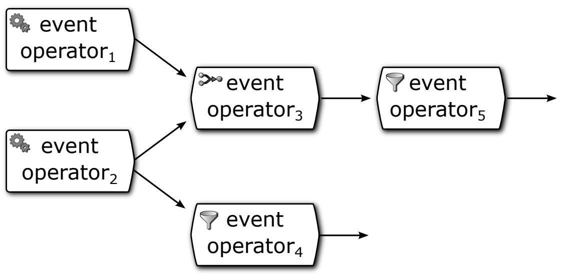
```
TOP = SIMPLE_EF [ MAGNITUDE > 600 ] ( maxF )
maxF = MAX_EC ^0:20 ( co2egen )
co2egen = PERIODIC_EG SENSOR_CO2_PPM ^0:20 @floor1
```

- Detect fire hazard situations in room 23, where average temperature exceeds 45°, while the average relative humidity is inferior to 70%:

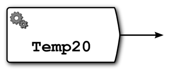
```
TOP = AND_EC ^1:00 ( tempf, humidf )
tempf = SIMPLE_EF [ MAGNITUDE > 45 ] ( tempAvg )
humidf = SIMPLE_EF [ MAGNITUDE < 70 ] ( humidAvg )
tempAvg = AVERAGE_EC ^1:00 ( temp )
humidAvg = AVERAGE_EC ^1:00 ( humid )
temp = PERIODIC_EG SENSOR_TEMPERATURE_CELSIUS ^0:20 @windows
egen = PERIODIC_EG SENSOR_HUMIDITY_PERCENT ^0:20 @lights
```

In the last example, the temperature and humidity readings will be buffered and every three events will be averaged before comparing to the thresholds. These relationships can be adjusted by specifying the event generation (i.e., sampling) frequency and the width of the averaging window. The conjunction is satisfied only when both conditions are met (high temperature and low humidity).

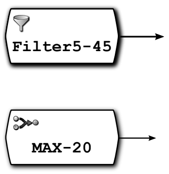
In addition to a *declarative* language to specify event expressions, uWDL offers a *graphical* notation to represent them, which we call *event-based diagrams*. In essence, ukuFlow's event based diagrams resemble the main BPMN notation as well as that of event processing chains [36]. In this notation, each event operator is represented by a box, and has at least one outgoing arrow, naturally reading best from left to right. Figure 3.24a exemplifies two such expressions with event operators ($\mathtt{eo}_x$). These expressions are constructed in nested fashion as $\mathtt{eo}_5(\mathtt{eo}_3(\mathtt{eo}_1,\mathtt{eo}_2))$ and $\mathtt{eo}_4(\mathtt{eo}_2)$.



(a) Generic event diagram



(b) event generator



(c) event filter (top) and composer

Figure 3.24.: Event-based diagrams

Event generators are represented through boxes pointing towards the right, adorned with a set of gears, and typically located leftmost of an event-based diagram. This can be seen in the event operators 1 and 2 of Fig. 3.24a, and with `Temp20` in Fig. 3.24b. Event filters and composers are hexagons, as shown in event operators 3, 4 and 5 of Fig. 3.24a, and with `Filter5-45` in Fig. 3.24c. Filters are adorned with a funnel, composers with a merging icon.

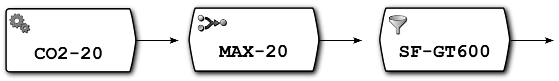


Figure 3.25.: Event-based diagram for event expression to detect CO_2 concentrations exceeding a threshold of 600 ppm

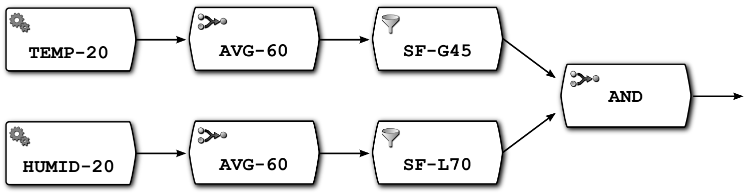


Figure 3.26.: Event-based diagram for event expression to detect fire hazards by checking that the average temperature is greater than 45° and that the average humidity is lower than 70%

In the Fig. 3.25 and 3.26 we present the graphical view of the two last examples of the previous subsection. These views highlight the usefulness of the event-based diagrams, and are an important contribution to the simplification of the design of WSAN applications for domain experts.

Note that this is a centralized view of the overall event expression. As we will see in the next chapter, at runtime, each node fulfills a particular role with regard to this expression in order to carry out this collective event detection.

3.6 Summary

This chapter described ukuFlow's Workflow Definition Language, uWDL. The model was designed carefully by adopting a subset of BPMN 2.0's notation and extending it, as shown in Fig. 3.27, to suit WSANs. Under the Workflow Patterns terminology, ukuFlow includes the workflow operators known as *basic control-flow patterns*. Despite this simplicity, it is possible to describe a wide range of applications, as shown with the airport scenario of Section 2.3.1.

Elaborating on the idea of scopes to engineer event-based systems, uWDL resorts to the declarative nature of the Scopes framework to address network nodes according to spatial properties.

In the literature, approaches can be found that either a) offer a much higher flexibility (i.e., support more workflow operators or parameters for these), but are not targeted to the resource-constrained sensor nodes, or b) carry out individual tasks in highly efficient ways on the targeted hardware (e.g., power-efficient detection of specific events), but are not intuitively programmed by average domain-experts. The mixed graphical/declarative/imperative model proposed in uWDL addresses both aspects simultaneously, and remains extensible with further workflow and event operators as required by the application domains.

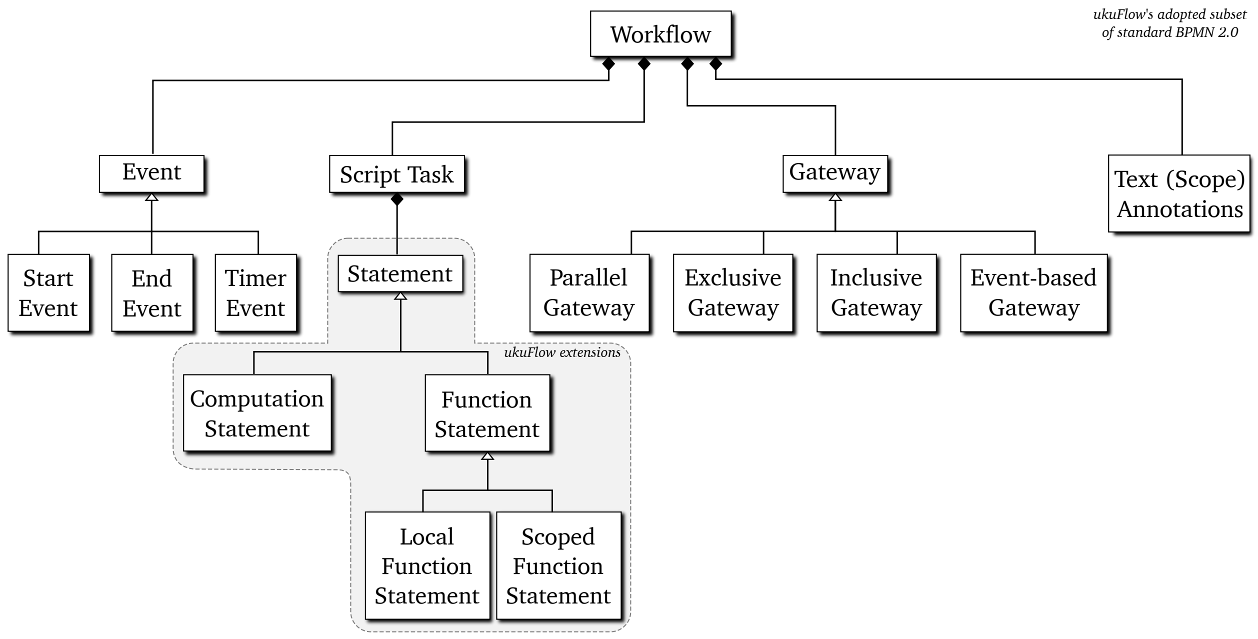


Figure 3.27.: Summary of uWDL

4 ukuFlow Design and Implementation

Any problem in computer science can be solved with another level of indirection...

David Wheeler (1927–2004)

...except for the problem of too many layers of indirection.

Kevlin Henney

In the decade of the 90's, the interest of the research and industrial communities on workflow systems led to the conformation of the Workflow Management Coalition (WfMC), a consortium that aims to create and contribute to workflow and business process management. One of their most relevant contributions, the Reference Architecture [68], is the architectural precursor to most current workflow systems: products such as IBM FlowMark[85], JBoss jBPM[77] and Software AG webMethods, etc., have adhered to this architecture. These systems, however, are targeted at large platforms such as servers and mainframes. Variations to the reference model have also been proposed, e.g., when the system needs to consider mobile [58] and disconnected participants [3], but still remain too large to be able to run in the resource-constrained devices aimed at in this thesis. In this chapter we present the design of a novel system that reflects main concepts of WfMC's reference architecture and that still enables its usage on low-power sensor nodes, and discuss a number of concepts, techniques and implementation details that deal with WSAN's complexities.

4.1 System Requirements and High Level Architecture

A macroprogramming approach to use workflows for WSANs poses a number of requirements to its architecture:

- *easy definition of application logic* (i.e., workflows) by means of a graphical user interface (GUI)
- *reprogrammability*, in terms of not needing to reprogram nodes (i.e., physically connect them to a programmer board) to change their logic every time a new workflow is deployed or undeployed
- *system adaptability* to network topology changes (by adding or removing nodes or varying link qualities), as well as *extensibility* with regards to incorporating new sensors or actuators attached to the nodes
- *in-network execution* of workflows, independently from external servers
- *multi-user* support, enabling multiple workflows to run in the network in parallel and/or pseudo-parallel fashion

As described by the WfMC in the Reference Architecture [68], the high level view of any WfMS is composed of three areas:

- *build-time* functions, which are related to the design and modeling of workflows;

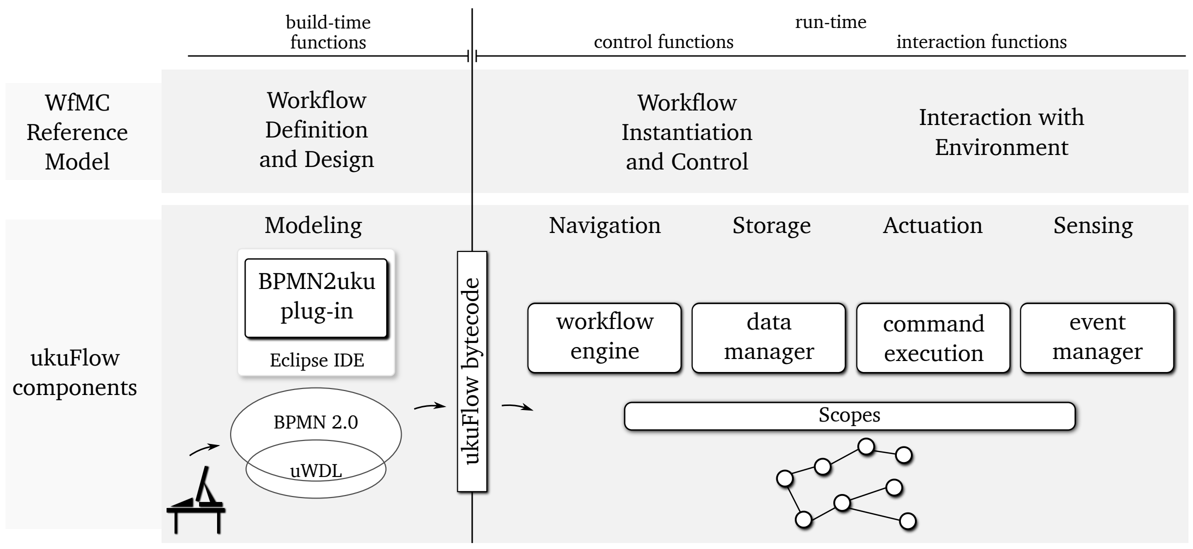


Figure 4.1.: High level architecture of ukuFlow

- *run-time control* functions, which concern to the core workflow engine functionality that make up the workflow enactment service; and
- *run-time interaction* functions, which, in the case of WSANs, relate to sensor and actuator nodes (or groups thereof).

Figure 4.1 presents the high level architecture of ukuFlow, its components, and their relation to the concepts of the WfMC's reference model. One important design decision is to make the ukuFlow engine an interpreted system. This enables a highly dynamic operation at very low node reprogramming cost: when deploying or undeploying a workflow, instead of distributing large binaries (i.e., data objects in the orders of several KBs) wirelessly throughout the affected nodes, only the workflow *specification* (which, as we will see, occupies some hundred bytes) needs to be transmitted.

Workflow specifications follow the *ukuFlow bytecode*, an instruction set designed to represent uWDL workflows in a compact fashion. The bytecode connects the build-time and run-time aspects with each other. For the build-time functionality, ukuFlow employs a workflow editor, *BPMN2uku*, which is integrated into the popular IDE Eclipse. Next, in Section 4.2, we further describe the plug-in extension. Run-time control can be split into two aspects: *navigation* and *data management*. The *workflow engine* in ukuFlow is in charge of controlling the process instantiation in the network, as well as the navigation (i.e. execution) through the corresponding workflow operators, and is described in Section 4.3.1. In turn, the *data manager* is responsible for storing workflow instance's data, which can be arbitrary user variables or might precede from node sensors (further described in Section 4.4). Section 4.5 discusses the application of Scopes as the network layer required in order to implement the interaction functions such as the distributed execution of commands (Section 4.6) and event detection (Section 4.7).

4.2 The BPMN2uku Editor

To facilitate the design, composition and editing of workflows, literally hundreds of tools exist offering a graphical interface. Among these, BPMN-based workflow editors can be found, both in open-source form, such as the Eclipse BPMN2 Modeler or the Oryx Editor, as well as commercial variants, such as Activiti Eclipse Designer, Signavio Process Editor or ARIS Express. In this work we adopt the Eclipse BPMN2 Modeler [113] as base system for the uWDL extensions contained in BPMN2uku due to its openness and extensibility. The BPMN2 Modeler is an Eclipse plug-in in incubation phase built using a number of open-source frameworks such as the BPMN 2.0 Eclipse Modeling Framework meta-model (an open-source reference implementation of the BPMN 2.0 specification from the OMG) and the Graphiti graphics framework. Through the integration to Eclipse, services such as project management and source code version control, among others, are immediately available to the developer.

The most important functions of BPMN2uku are a) a graphical interface for modeling workflows with uWDL as described in the previous chapter, which is structured into the main workflow diagram editor and an event script diagram editor; and b) the validation and conversion of such models into the ukuFlow bytecode for their deployment into a WSAN running the ukuFlow middleware.

4.2.1 Workflow Editor

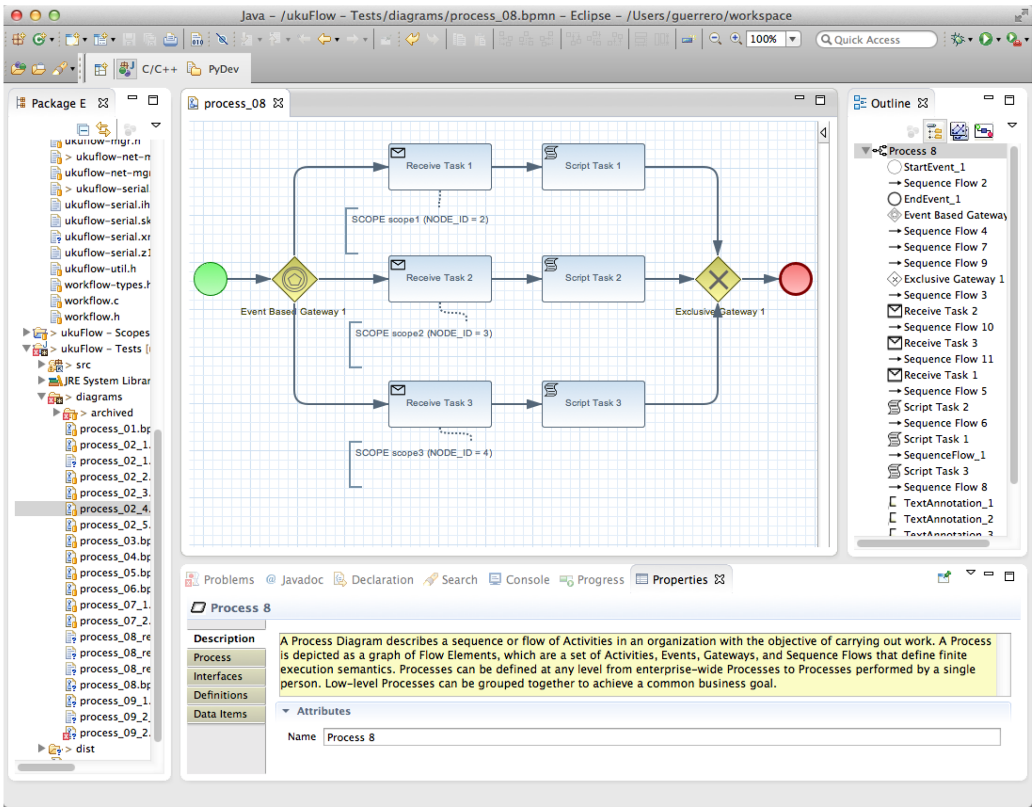
BPMN2uku's workflow editor is the main interface to model WSAN application logic. When a new workflow model is created, a wizard is presented which requires the workflow's name and file location, and the user is taken to the plug-in's main interface. This interface consists of a *canvas*, a *palette*, a *properties pane*, and the *package* (i.e., project) *explorer*, as shown in a screenshot in Fig. 4.2a. Then, the canvas is populated with a simple workflow consisting of a start and an end event connected by sequence flow that serves as a starting point.

The canvas is a freehand-drawing area into which the developer can drag and drop elements from the palette (or copy from other models), and then connect these in a quick and intuitive way. Once on the canvas, the developer can link workflow elements with each other through the corresponding connectors (i.e., sequence flows or associations). Workflow elements (tasks, gateways, etc.) available are presented in the palette. The complete palette available in uWDL, grouped by type, is shown expanded in Fig. 4.2b. In addition, the palette contains a number of workflow patterns [117] that further facilitate the composition of workflows.

The properties pane (at the bottom) is the component through which element properties can be visualized or edited, such as their description or name. Furthermore, this pane is context-sensitive, thus its contents depend on the currently selected element in the canvas. For example, when a *script task* is selected, a text field is shown where the user can enter the statements that should be executed; if a *receive task* is selected, the user can enter the event expression to be detected in that branch of an *event-based exclusive decision gateway*. When no element is selected, the properties shown in the pane refer to the entire workflow, and thus offer the user the possibility to change the workflow name, the number of instances and the looping properties. Alternatively, by double-clicking on a workflow element, a pop-up window is presented through which these properties can be modified.

The package explorer is the area where a project's multiple files are listed, providing an overview of the types of files within a project. Each file's current status is represented by means of a small icon next to them, signaling that they are correct models, or that they contain warnings or errors or that they are out of sync with the version control system.

Note that workflows created with the BPMN2 Modeler, and hence in BPMN2uku, are stored in an XML format for which a Document Type Definition (DTD) is available. Proficient users can thus streamline the creation of workflow models by generating these XML files themselves, if desired. For this purpose it is also possible to resort to an XML editor integrated into the plug-in.



Process 8

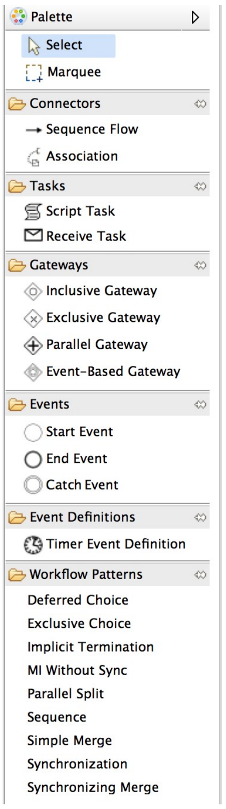
Description | Process | Interfaces | Definitions | Data Items

A Process Diagram describes a sequence or flow of Activities in an organization with the objective of carrying out work. A Process is depicted as a graph of Flow Elements, which are a set of Activities, Events, Gateways, and Sequence Flows that define finite execution semantics. Processes can be defined at any level from enterprise-wide Processes to Processes performed by a single person. Low-level Processes can be grouped together to achieve a common business goal.

Attributes

Name: Process 8

(a) Main canvas



(b) Expanded palette

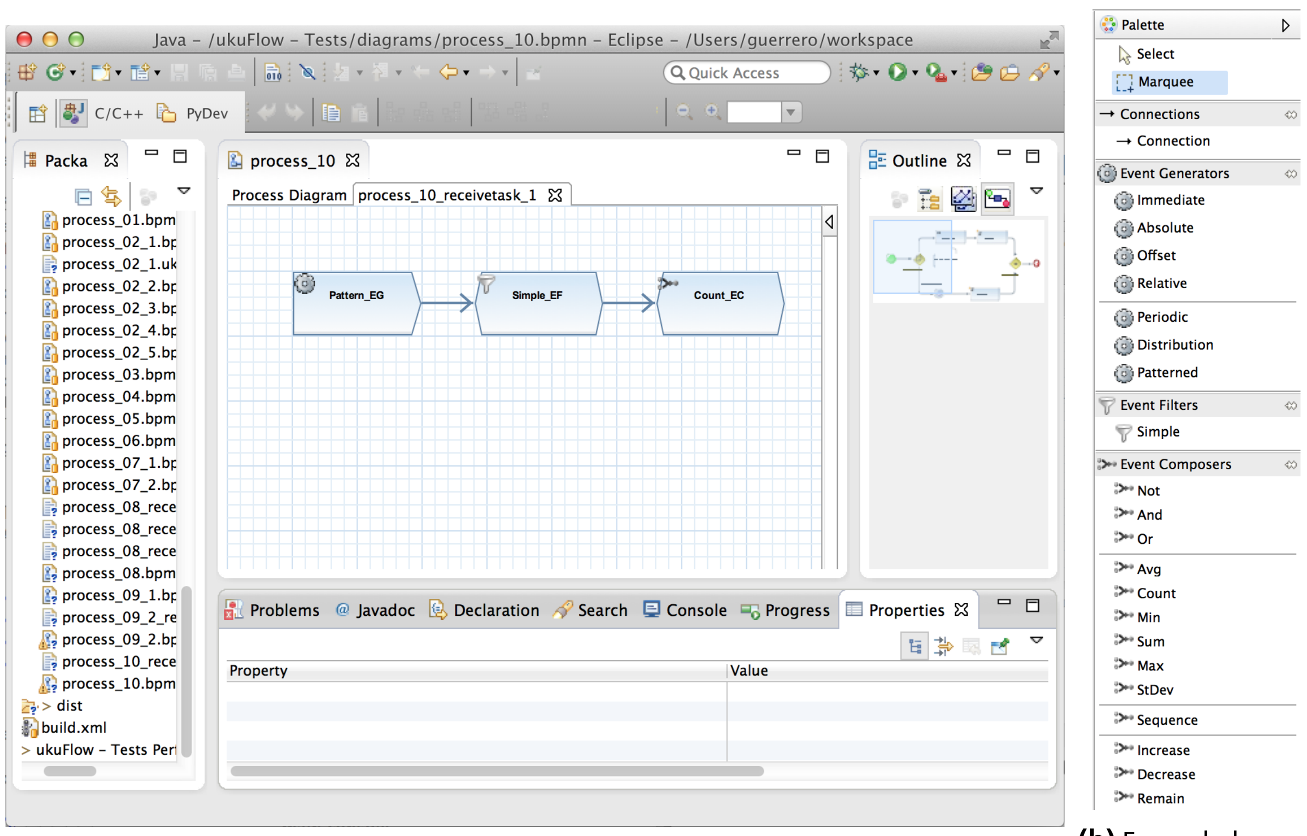
Figure 4.2.: BPMN2uku workflow editor

4.2.2 Event Script Diagram Editor

While the definition of actions in script tasks presents a convenient interface to enter statements, writing and understanding event scripts manually can quickly become a complex duty. For this purpose, BPMN2uku complements the workflow diagram editor with an *event script* diagram editor.

Similarly to the workflow editor, the event script diagram editor offers a canvas for freehand positioning of elements from the palette (Figs. 4.3a and 4.3b). The editor is opened either through a contextual menu item, or through an icon that appears when hovering over a *receive task*. This opens a new tab within the workflow editor, and creates the necessary graphical representations of event operators and connections in it, in case that event expressions were available (i.e., entered in the properties pane). The palette is also grouped by event operators types (event generators, filters and composers).

The user can graphically edit the properties of each event operator by double-clicking on them, which opens a pop-up window whose contents depend on the type of operator. The event operators can be connected, following their cardinality rules. Saving the contents of the event script's diagram will store the information on a separate XML file that contains information about the operators as well as the graphical layout. The core information is parsed and the corresponding event script is generated and used to update the script found in the corresponding *receive task* of the workflow diagram.



(a) Main canvas

(b) Expanded palette

Figure 4.3.: BPMN2uku event script editor

4.2.3 Workflow Validation

In order to simplify the design of the workflow engine that runs on the resource-constrained sensor network, it is crucial to reduce the chances of deploying incorrect workflows as much as possible. The BMPN2uku plug-in incorporates several types of validators that further facilitate the development of correct workflows.

The majority of the validations are performed *on demand*, either when the user explicitly requires it, or indirectly when a workflow registration is requested. These validations ensure that the workflow is *well-formed*. A workflow is said to be well-formed if it adheres to the following rules:

1. there is only one *start event*, and it must have no incoming and one outgoing sequence flow,
2. there is only one *end event*, and it must have no outgoing and one incoming sequence flow,
3. its activities (*script task*, *receive task*, *timer catch event*, etc.) have one incoming and one outgoing sequence flow, and
4. each *diverging* gateway matches with one *converging* gateway (and vice-versa).

Through these rules, the workflow engine has considerably fewer checks to perform at run time. This simplifies its code base, and reduces the chances of irrecoverable faults, where the autonomous workflow execution would need to be halted. In particular, by requiring that workflows have a single *start event* it is simpler for the engine to identify the starting point for the execution of a workflow instance. Similarly, one single *end event* leads to simpler workflows whose instances must be terminated (and its resources cleaned up) at a single place. Finally, by requiring balanced diverging and converging gateways (of all types), the possibility of unbounded forking of tokens is avoided, which would lead to uncontrolled memory utilization.

In order to perform these validations, a parser was generated using JavaCC and a grammar that describes uWDL. This parser reads the XML representation of the workflow, and tries to generate a tree object model from it. Once a tree is successfully constructed, further parsing is performed on workflow operators such as *script task*'s statements and *receive task*'s event scripts, and *text annotation*'s scope definitions. While these scripts share many elements with each other, separate parsers were built for each of these, which facilitates the code comprehension and maintenance.

In addition, a number of checks occur early, at *design time*. For instance, the event script editor does not allow connections to end at event generators, since these only allow outgoing connections. Another example in the event script diagram is the avoidance of loops between event operators. For this purpose, a set of validators running in the background are invoked when the user connects workflow (or event) operators with each other. If such connection leads to a loop, the connection is forbidden.

Note that it is possible to build workflow transformations that convert a ill-formed workflow into a well-formed one (e.g., as proposed in [31]). Shifting the validations and transformations to the design time tools, however, heavily offloads the sensor network from unnecessary functionality, and contributes to the requirement of in-network workflow executing. A detailed description of the well-formedness checks is provided in [63].

4.2.4 The ukuFlow Bytecode

After successfully parsing the entire workflow, its representation is ready to be deployed into the network. Using the native BPMN-XML representation of the EMF underneath the workflow models, however, is impractical on sensor nodes. To begin, while these XML files *could* typically be stored in a sensor node's internal memory (i.e., RAM or flash), they are considerably large. This is depicted in the upper plot of Fig. 4.4. This plot presents the size of the pure BPMN-XML files for 20 test workflows of varying complexity, defined using the BPMN2uku plug-in. Workflows represented with this format are large due to their verbosity and because they include information of the graphical layout (position on the canvas of each element), namespace definitions (containing a series of long URLs), element names and descriptions, and encoding information, all of which are not fundamentally required for the execution of the workflow. As presented in the top figure, reducing the unnecessary information from the BPMN-XML files still yields workflows that require multiple KBytes (cf. the BPMN-Contents bars). Furthermore, this would require an XML parser running on the sensor network, a functionality that commonly requires dozens of KB. For instance, Mini-XML [129] can be made to compile to 36 KB, not leaving much out of the 48 KB available in TelosB or other mote-class devices for other functions.

For this reason, we have designed the *ukuFlow bytecode*, a compact format that includes the minimal information required for the workflow execution. Figure 4.5 presents the general structure of a workflow expressed in the ukuFlow bytecode. The structure is divided into the workflow's metadata, followed by a series of workflow operators and finally the scope specifications. A detailed description of ukuFlow's bytecode is presented in Appendix A. Since most of the platforms employed have 16-bit microcontrollers, we visualize the ukuFlow bytecode aligned to 16 bits. Note however that this is not a 16-bit instruction set. Instead, the operators are of variable width and occupy anything from 2 to multiple bytes.

The bottom plot of Fig. 4.4 illustrates the efficacy of this representation, showing the size of uWDL workflows compiled into the bytecode (note the program sizes specified in bytes). We find this savings to average 2.84% of the BPMN-Contents format. At a gross of around 17 bytes per workflow element (i.e., including workflow metadata and corresponding scope definitions), it is possible to deploy multiple

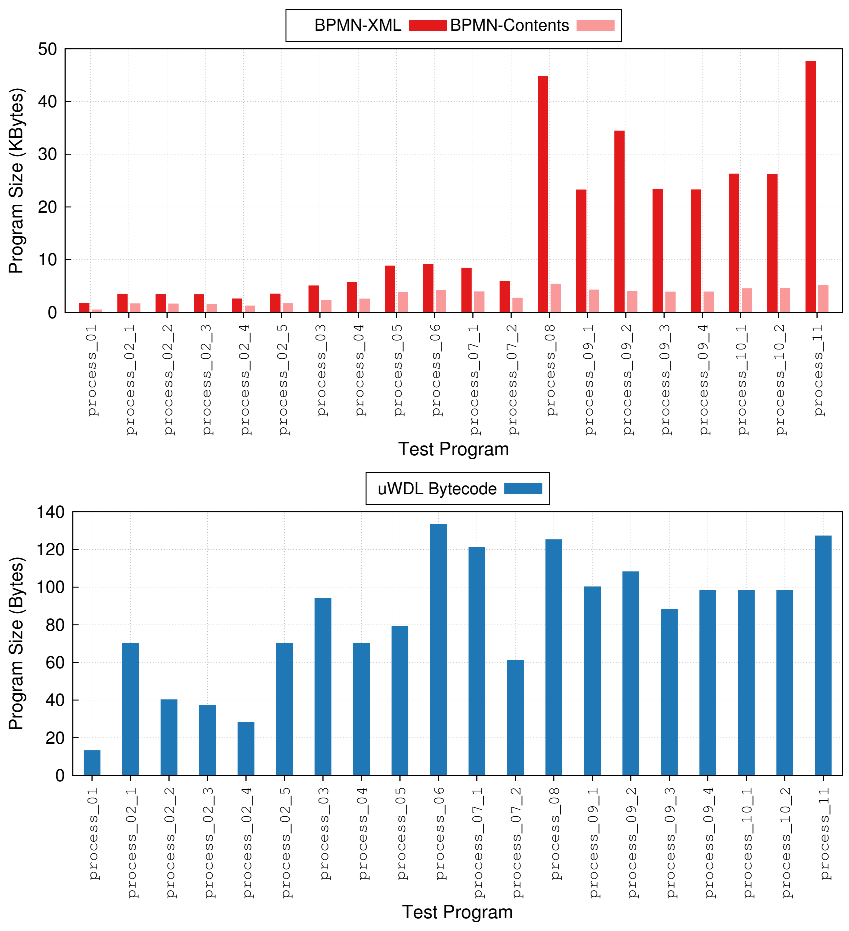


Figure 4.4.: Compression properties of the uWDL bytecode representation

workflows, each with 10's or even 100's of workflow elements at a single node, and still leave sufficient RAM present in mote-class devices such as the TelosB (~10 KBytes).

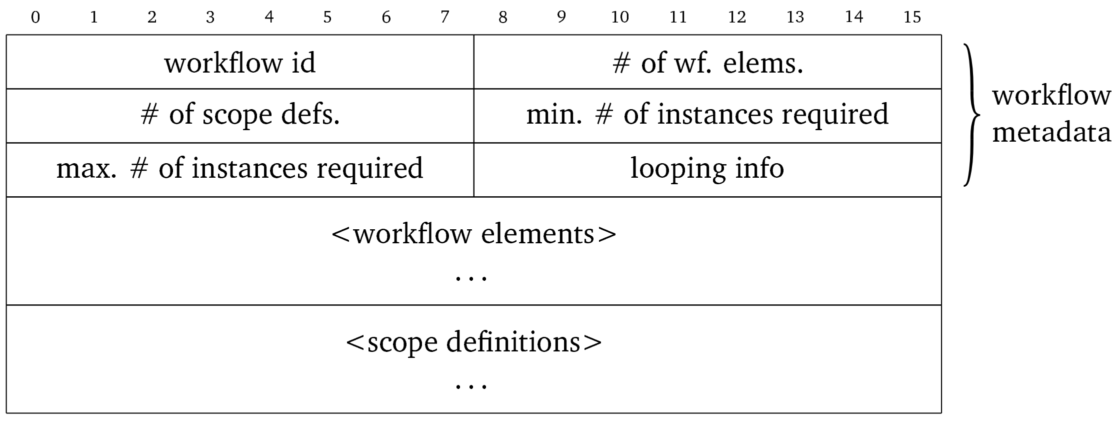


Figure 4.5.: ukuFlow bytecode structure

The size of a workflow representation can be further reduced. One approach is to resort to the Abstract Syntax Notation One (ASN.1), a standard from the International Organization for Standardization (ISO) for the representation and (de)coding of data. When using the (unaligned) Packet Encoding Rules, it is possible to make each field occupy only as many bits as needed by the available values. Decoding ASN data on a sensor node, however, requires a specialized library that –again- occupies dozens of KB. An

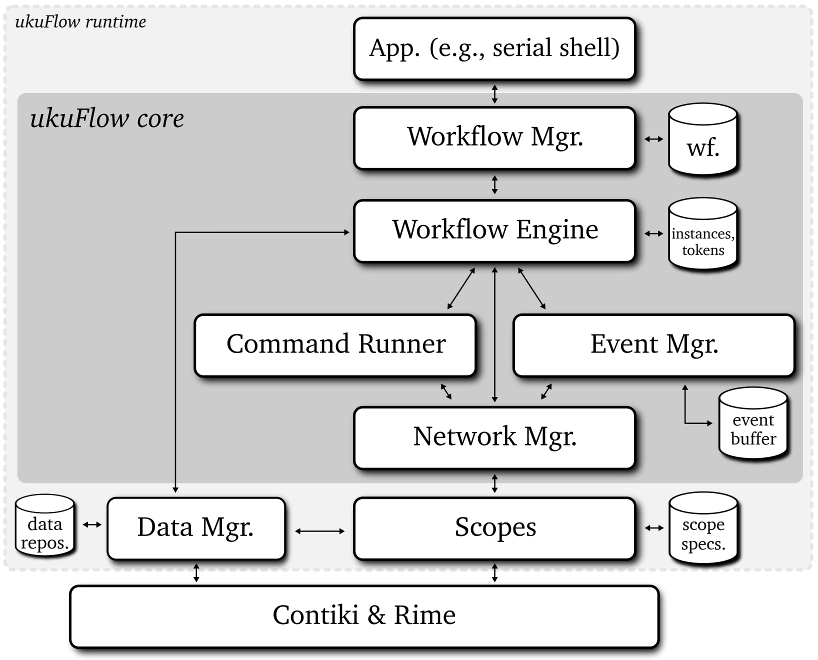


Figure 4.6.: ukuFlow runtime architecture

alternative approach might be offered by specialized, lightweight compression mechanisms such as LZO or S-LZW [119], but its application goes beyond this work.

4.3 The ukuFlow Runtime Architecture

Once a workflow is deployed, it is the duty of the system runtime to deal with its efficient execution. In this section we describe the central aspects related to workflow management and execution. The ukuFlow runtime has been designed to operate entirely in-network on low power, 8/16-bit embedded microcontrollers. Figure 4.6 presents the main modules of the ukuFlow runtime for nodes in the sensor network.

The overall ukuFlow runtime can be broadly described as a multilayered –stacked– architecture, with non-strict interactions between layers. At the lowest layer there is the operating system (i.e., Contiki), offering essential functionality such as process and memory management, as well as primitive communication mechanisms (the Rime networking stack). Sitting immediately on top of the OS there is the Scopes framework (further discussed in Section 4.5) and the data management module (Section 4.4). The ukuFlow core is composed of the workflow manager and engine modules (described next), the command runner logic (Section 4.6) and the event manager (Section 4.7).

While the previously described modules strive to realize distributed functionality (group creation, workflow execution, etc.), at the topmost layer, node-level modules can be employed that, e.g., interface with the serial port in order to accept workflow (de)registration requests.

4.3.1 Workflow Management and the ukuFlow Engine

The key service offered by ukuFlow is the in-network execution of workflows. An important architectural decision is to make ukuFlow an interpreted system that understands and executes workflow bytecode.

This relieves the system from having to completely reprogram nodes dynamically. As a result, ukuFlow's design reflects a *virtual machine* architecture that can be compiled for various sensor platforms, thus enabling interaction in a heterogeneous network (assuming a common communication layer such as IEEE 802.15.4). In order to minimize energy consumption, the entire design of ukuFlow is event-driven – exploiting the *protothread* abstraction provided by the operating system.

The workflow manager is the outermost module that offers the functionality of (de)registering workflows, and listing running ones. On a registration request, the workflow manager checks whether there is sufficient memory available to store the workflow specification (i.e., the bytecode), and if so it makes a copy of the specification locally in RAM and serves it to the engine. At the cost of a slightly higher code complexity, most operations in ukuFlow are then made *in-place*, i.e., they operate on this single workflow bytecode array without requiring having to copy or split it into other objects. This also implies that memory is accessed in an unaligned fashion in 16-bit microcontrollers (such as the MSP430), to avoid using 16 bits for data items that require only 8 bits.

Workflow Instantiation and Token Management

After successful registration, a workflow is ready to be instantiated and tokens for it allocated. As introduced in Section 3.1.2, a workflow can be defined as a singleton or require multiple (concurrent) instances, and be specified to execute once or a fixed or unlimited number of times. This information is provided in the workflow's metadata, Table 4.1 provides a summary together with smallest, largest and default values (note that $n \leq m$). While the indication of the maximum number of instances and the looping property are *strict* values, the minimum number of parallel instances is a *desired* value. This means that ukuFlow attempts to have *at least* as many instances in parallel as shown, but the system might not be able to accommodate as many due to engine configuration parameters (presented later) or to a lack of resources (e.g. RAM).

Table 4.1.: Specification of workflow instances

	smallest value	largest value	default value
min. # of instances, n	0	255	1
max. # of instances, m	1	255	1
looping	1	255 or ∞ (0)	1

Similarly to modern operating systems, the ukuFlow engine deals with workflow instantiation by means of a two-level scheduler: a *long-term* scheduler that decides which and when registered workflows are allowed to be instantiated, and a *short-term* scheduler that decides which token of a ready workflow instance receives CPU cycles.

In addition to the workflow specification bytecode (stored by the workflow manager), at runtime other data structures are employed to track the workflow execution, which collectively resemble an operating system's process control block (PCB). First, *workflow nodes* contain a pointer to the respective workflow specification, a count of the workflow's instances, and the workflow's state. A *workflow instance*, in turn, contains a pointer to its workflow node, a pointer to a data repository (where workflow data variables are stored), and a count of the instance's tokens. Finally, *workflow tokens* can be understood as threads, as they share a common data repository and can operate in parallel on it. Through this layered process structure, the memory usage requirements are optimized, since common data is not replicated unnecessarily. Objects of these structures are created and stored dynamically, in the sensor node's heap. For each workflow operator being visited by a token, its content is adjusted to store (link to) operator-dependent state information. In addition, tokens keep a reference to their *parent* token.

The initial token created for a workflow instance has no parent, and is similar to process `0` in Unix operating systems. Tokens created as a result of visiting *parallel* or *inclusive gateways* reference their parent token, which is put in the blocked queue until the merging gateway is visited. This information is used, e.g., to synchronize multiple child tokens of a single workflow instance upon their arrival at a merging gateway.

The logic of the long-term scheduler to decide when to create new workflow instances is realized with a finite-state machine, presented in Fig. 4.7. A workflow node can be in one of the states `new`, `spawn`, `running`, `blocked` or `terminated`. When a new workflow is registered, a workflow node is created and assigned the state *spawn*. Transitions between states are given by factors such as the workflow's specification of minimum and maximum number of instances, its looping properties, and the system load. Most transitions occur either when the system is ready to create a new instance, or when an instance terminates. When a workflow is requested to be unregistered, its running instances must be stopped. Instead of immediately interrupting their execution, these are allowed to gracefully continue their current activity and only then they are stopped. Eventually, all of the workflow instances finish, which changes the workflow's state to `terminated`, releasing the associated resources.

Figure 4.8 presents the interaction between the two schedulers. Once a workflow is registered (①), a corresponding workflow node is created and put into the *ready workflows* queue, indicating that workflow instances can be created for it. If resources are available, a workflow instance is spawned (②) and linked to the workflow node. If the workflow node still requires further instances, it is enqueued again in the *ready workflows* queue, otherwise it is moved to the *running workflows* queue. Whenever a workflow instance is created, the short-term scheduler is notified so that it can create the instance's initial token, which is associated to the workflow's (unique) start event (③). This token is inserted in the *ready tokens* queue, where it waits until it is *dispatched* (④) and receives CPU cycles. If the token's task is completed quickly, it is put back in the ready tokens queue, otherwise (e.g., it is a long operation involving networking aspects), it is put in the *blocked tokens* queue (⑤). Once such a long operation concludes, the token is put again in the ready queue (⑥). When a token visits a forking gateway (such as the *parallel gateway* or the *inclusive decision gateway*), further tokens are created, linked to the corresponding (parent) workflow instance, and pushed to the ready tokens queue. Similarly, when a token visits a merging gateway, tokens are deleted (⑦). Eventually, the last token of the last workflow instance is finished, which causes the associated running workflow to be put in the ready queue (⑧) and later be unregistered (⑨).

The workflow engine can be configured with a number of parameters that help to control the execution and memory utilization. These parameters can be tuned at compile-time depending on the processing

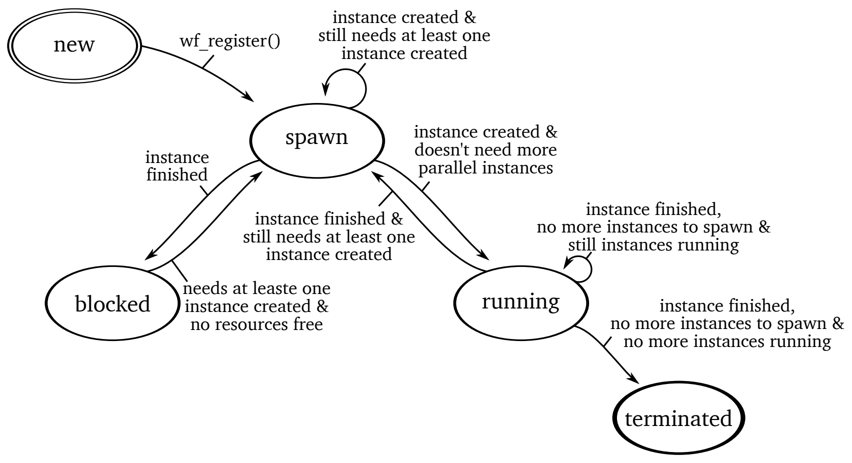


Figure 4.7.: ukuFlow long-term workflow scheduler's finite-state machine

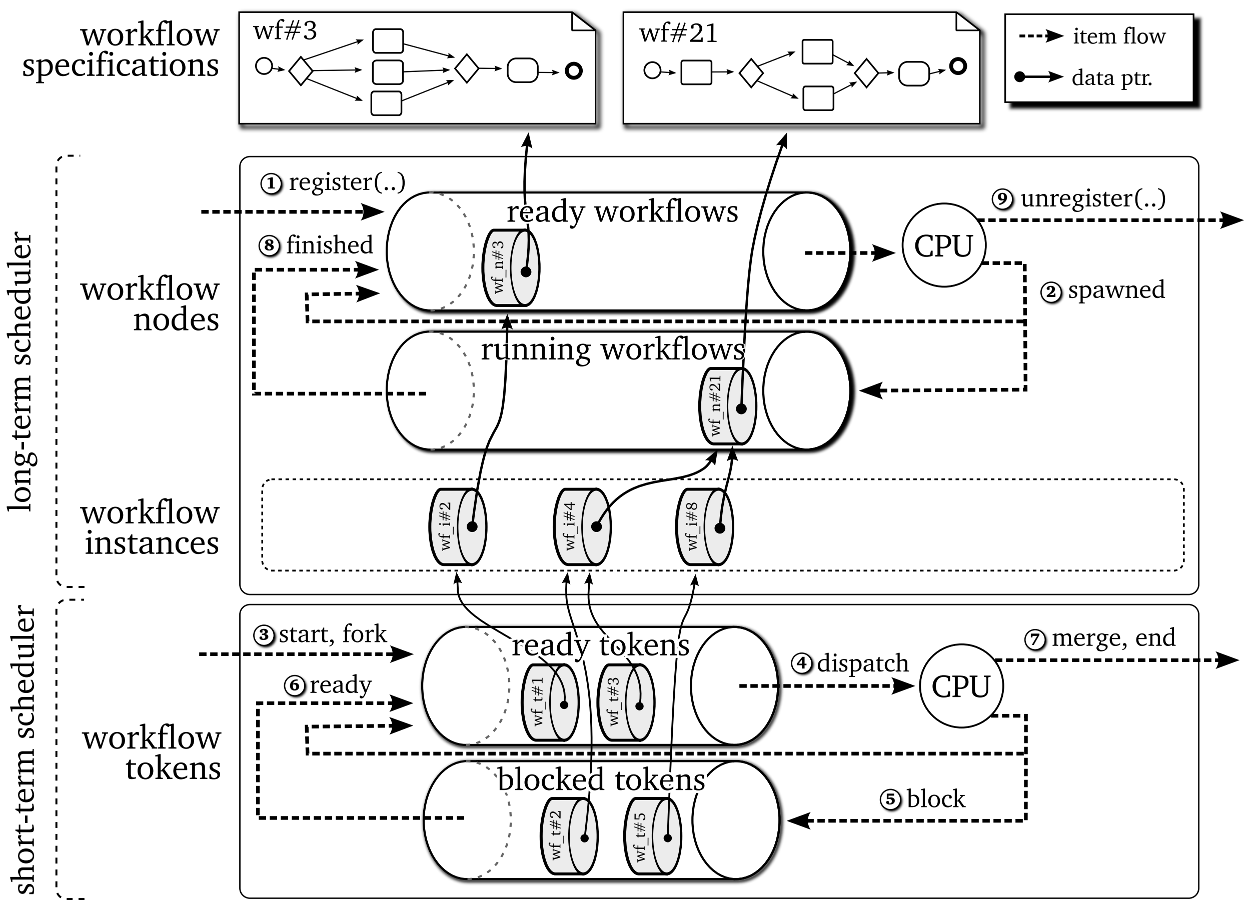


Figure 4.8.: ukuFlow engine's two-level workflow scheduling

capabilities and available RAM of the target platform, and are listed in Table 4.2, together with values for TelosB-type of nodes (Sky, jCreate, Z1 and XM1000).

Table 4.2.: ukuFlow engine configuration parameters

parameter	description	TelosB values
`MAX_REGISTERED_WORKFLOWS`	max. # of workflows registered in a single node	5
`MAX_WORKFLOW_INSTANCES`	max. # of instances that one node can execute in parallel	10
`MAX_INSTANCES_PER_WORKFLOW`	max. # of instances per workflow	`MAX_WORKFLOW_INSTANCES/2`
`MAX_ACTIVE_TOKENS`	max. # of tokens at a single node	20

The implementation of the ukuFlow Manager and Engine components is mainly event- (or interrupt) based: the system's core remains idle until relevant events occur. The logic of each type of workflow operator (start event, script task, gateways, etc.) is implemented in *event handlers* which run till completion. This broad modularization increases code readability, facilitates debugging and improves extensibility (e.g., if further workflow operators were to be included). The long and short-term schedulers, in contrast, are realized with protothreads [35], i.e., procedural code that uses *conditional blocking* statements to wait until relevant events occur (e.g., until a workflow or a token is ready, respectively). Waiting protothreads enable the CPU to go to sleep mode, and thus save energy. When these events occur, the corresponding protothread is dispatched by the operating system, allowed to run (e.g., invoke the corresponding workflow operator's event handler), and then sent to the wait state again. The communication between protothreads is implemented via a lightweight message-passing mechanism offered by the operating system that uses a first-come, first-served queue to buffer the messages. This enables an asynchronous operation, for instance, to deploy workflows or operate on radio messages.

4.4 Data Management

As described in Section 3.2, the ukuFlow macroprogramming framework uses a simplified approach to data management that builds on the notion of every node in the WSAN being an independent, addressable entity which owns and manages its data. Similarly to systems such as Agilla [41] or Abstract Regions [139], data is organized in a repository of name-value pairs. Data management is split into two functions: a) data storage and access functions, and b) expression evaluation functions.

Data storage and access is accomplished through an API to create and remove (local) data repositories, as well as to lookup, add/set and remove entries in these repositories. The data manager running on each node maintains a set of repositories as requested by the calling modules. Conceptually, each data repository contains two types of name-value pairs: WSAN-specific data (such as a node's sensor readings), and repository-specific data. This means that sensor (and node) data is accessed indirectly through the data manager as name-value pairs, i.e., under a name such as `SENSOR_TEMPERATURE_CELSIUS`.

Since a node's repositories share the sensor data entries, these are internally only allocated *once*. When the module is initialized, a hidden shared repository is created. Depending on the actual node platform and connected sensors and actuators, the necessary common entries are populated. Internally, the data manager stores two types of entries: those that are manually (i.e., explicitly) updated by the user/repository owner, and those that are automatically updated. For the latter, the user specifies a *time-to-live* (`ttl`) attribute when a repository is created. Whenever a name-value pair is requested, its *age* is calculated and, if it exceeds the repository's `ttl`, the entry is automatically updated. The data manager implements update functions for all of the sensor-data type of entries. These make use of the low-level sensor drivers to obtain the necessary samples and eventually convert the raw ADC readings into the corresponding units. Additionally, the user might create extra name-value pairs and declare them as auto-updateable, for which a pointer to an updating function must be given.

The `ttl` of the shared repository is automatically set to the lowest common `ttl` of all other local repositories whenever a repository is created or removed. Note that the `ttl` value is not necessarily the update interval, since data is *only* updated when the data of an entry is requested. Data will be updated that often only if the user module requests data at that (or a higher) frequency. This organization enables high-level modules to request data asynchronously from each other, and still only update respective entries once per `ttl` period, thus ensuring that sensor data is as fresh as requested by the repository with most stringent requirements, and simultaneously avoiding costly sensor sampling that result into new samples only marginally different from the previous ones.

In the current implementation, data is stored in RAM. This has the advantage of being very fast, at the cost of being completely volatile and thus not resilient to node crashes. An alternative to this is to implement the data manipulation operations to store data in the flash unit. While this has the advantage of being persistent after node reboots, it incurs a considerable energy penalty with every read/write operation (∼31 and 44mJ, respectively [133]), as well as an increased latency.

In addition to looking up name-value pairs, the contents of a repository can be used to check the validity of an entire expression. For this purpose, the data manager implements a simple parser that takes logic/arithmetical expressions in polish notation and outputs a result. An operand in these expressions might be any value stored in the repository (either automatically or manually updateable), or an 8/16-bit integer constant.

The services offered by the data manager module are used throughout the ukuFlow system. The Scopes framework, for instance, uses it to validate membership of a node into a scope specification. The ukuFlow engine, in turn, uses it within script tasks' statements to calculate mathematical expressions and to evaluate the expressions contained in conditional sequence flows. Finally, the event manager uses it to retrieve data from sensors and evaluate complex event operators.

4.5 Networking with the Scopes Framework

Workflow execution in a WSAN inherently requires collaboration (and communication) between various nodes. To this end, ukuFlow implements its services on top of the Scopes framework. The Scopes' application programming interface (API) offers three network primitives: *open* (i.e., create) a scope, *close* (remove) a scope, and *send* data between scope members.

A scope is opened from a *root* node, which disseminates the scope specification throughout the network. Upon reception of a scope open message, nodes decide whether they belong to it by checking their properties against the provided expression. In addition, the root node takes care of *maintaining* the scope membership through a refresh mechanism which periodically resends the scope specification throughout the network so that newly joined nodes can discover the active scopes, and subsequently join if necessary. Inversely, nodes that have not heard the refresh messages for a specified period will assume disconnection from the network (a potential failure of the root node) and leave the scope. Both the scope refresh frequency as well as the refresh miss threshold can be specified by the upper user modules. Through this mechanism, transient node failures that cause network topology changes are dealt with by adjusting routes automatically. Eventually, when a scope is not needed anymore, a scope close message is disseminated.

Since nodes' properties change dynamically, a node's membership to a scope can change during its lifetime. There are two alternatives to managing this variability. With a *default* scope, only nodes that are members of it retain the scope specification in their memory. Upon reception of a scope refresh message, the specification is re-evaluated against the current node state. This has the advantage that scope specifications do not unnecessarily occupy memory at nodes that are not member of the scope, which enables a larger number of open scopes in the network. The alternative to this is a *dynamic* scope, which cause all nodes to retain the specifications heard regardless of positive membership evaluation. When a dynamic scope is deployed, an additional protothread is activated that re-evaluates the node's membership to the scope, at a higher frequency than that of the refresh mechanism. This not only enables non-member nodes to become members between two refresh cycles, but also allows a finer scope membership detection. Dynamic scopes are energy-intensive, since the CPU needs to wake up more frequently and also sensors might need to be sampled (depending on the `ttl` attribute of the associated scope data repository, as described in the previous section).

Sending data through a scope is possible both from the root node towards the member nodes –Root-to-Members (RtM) traffic- or vice versa –Members-to-Root (MtR). The initial implementation, by S. Kilb [74], offers end-to-end messaging for both directions. This means that upper layer applications are notified of message reception only at member nodes. Nodes which are not member of the scope, but are located on the route between the sender and the receiver(s), act purely as forwarders, not sharing the data with upper modules. Together with additional security mechanisms [70], this design policy makes the Scopes framework adequate for its usage in multi-tenant (i.e., multi-company) environments.

The Scopes framework separates the grouping functionality from the routing layer. In [72] we have investigated two routing strategies: a gossip-based and a tree-based mechanism. The first, gossip-based strategy uses the idea of *polite gossip* of the Trickle algorithm [84] to reduce the overall network traffic as compared to the pure flooding mechanism, where the message dissemination cost is linear to the number of nodes in the network. In *polite-gossiping*, a node is only allowed to rebroadcast a message if, after a predefined interval, that message was not overheard from more than a certain number of neighbors. At the cost of an increased latency, this optimization reduces considerably the number of broadcasts required to disseminate a message, and thus lowers the chances of collisions. The gossip-based mechanism is used for scope creation, maintenance and removal operations, as well as for data traffic to and from members to the root. The lack of an overlay structure for MtR traffic makes the usage of this routing strategy inadequate for ukuFlow, because it prevents an efficient implementation of in-network, convergecast operations such as data aggregation.

The second, tree-based mechanism builds on the previous one and uses one general routing tree (from each scope root node) to forward traffic. Its operation is split into two steps. Initially, there is no network activity. When a node is instructed to open a scope, it first creates a routing tree by disseminating (via polite-gossiping) a tree construction request. Nodes that receive the tree construction message from multiple neighbors choose the parent with minimum hop count (ties are broken by order of arrival). Immediately afterwards, in a second step, the scope specification is disseminated through the overlay tree built previously in the first step. When a node receives a scope specification for which it is member, it notifies its parent node (a mechanism called *activation*) so that it is aware that downstream nodes are interested in receiving data. To avoid that multiple member child nodes send activation messages, activated parent nodes broadcast a *suppression* message. Activation messages are propagated upstream towards the root, unless a node has been suppressed. Later, when scope data is sent from member nodes towards the root, this is efficiently realized via multi-hop unicast, as the route is (distributedly) known by all intermediate nodes. The ukuFlow framework employs this tree-based mechanism.

Scopes is implemented on top of a number of Contiki Rime [34] communication stack primitives, as presented in Fig. 4.9. Tree construction, scope management messages and root to member (RtM) traffic is implemented by the best-effort network flooding module, `netflood` (which uses the identified polite-gossiping module, `ipolite`). Activation messages are sent to the next-hop parent node through reliable unicast (`runicast`), while suppression messages use the (identified) `broadcast` primitive. Finally, members to root (MtR) traffic implements multi-hop routing on top of the normal `unicast` primitive. In total, the Scopes framework makes use of 5 (virtual) Rime channels.

4.5.1 Scopes Optimizations for ukuFlow

In this work we have introduced a number of *optimizations* that enable taking full advantage of the Scopes framework. First, we introduced the concept of *interceptable* scope. Instead of offering end-to-end traffic, interceptable scopes have the special property that intermediate nodes are both notified about traffic (i.e., messages are caught and provided to the upper modules) and also given control of the forwarding procedure (i.e., modules may decide whether the forwarding should continue or not). As we will see in Section 4.7, this functionality is vital in order to enable in-network event detection.

Second, we incorporated support for *multi-application* scope usage. The original design allowed only one application to use a scope. If, within that application, multiple independent modules need to use a scope, a mechanism is needed to track whether a scope is already open, or inversely whether no module requires it anymore, and thus avoid the cost of re-opening the scope (i.e., flooding), or closing it while other modules still need it. We have incorporated this functionality into the scopes layer. For this purpose, a usage counter was added to the scope data structure that resembles a *reference counter* of garbage collection techniques. Since scope creation and removal is only allowed at the root node, this is entirely implemented locally.

Finally, we note that the data transmission primitive offered by Scopes is *packet-oriented*. If the size of the data message to be transported is larger than what is supported by the underlying MAC or physical

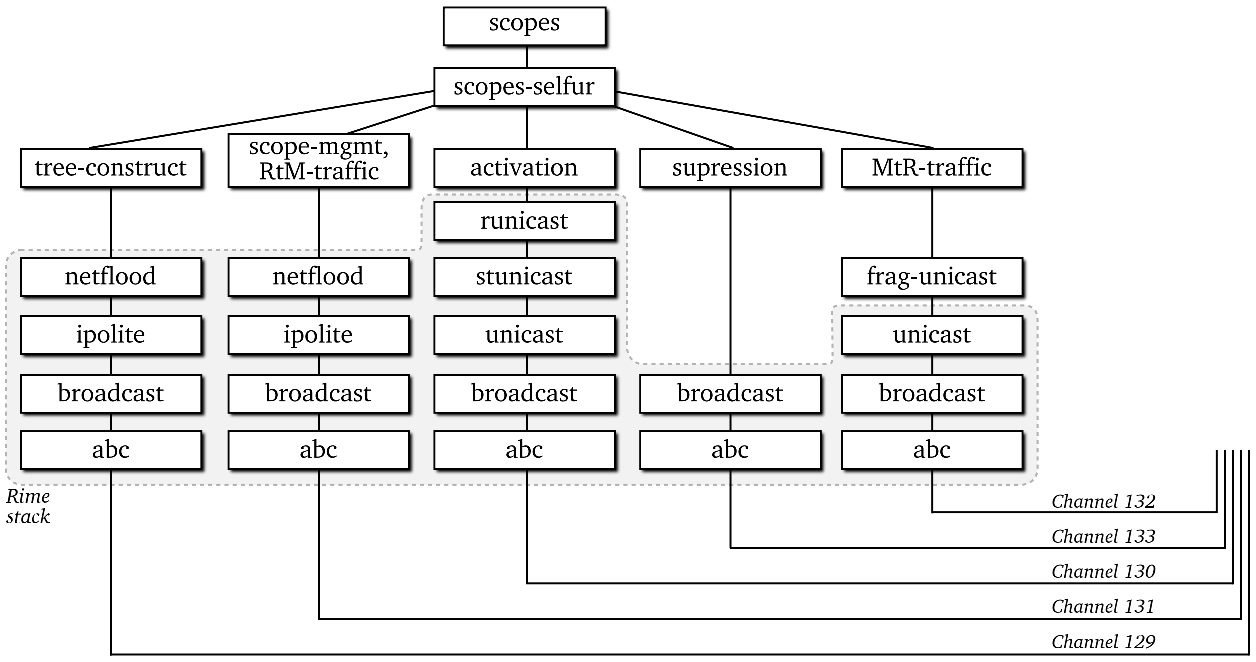


Figure 4.9.: The complete Scopes routing framework. Components over the gray background are provided by Rime, the upper modules by Scopes, and the `frag-unicast` was developed as extension to support ukuFlow. Each of the 5 stacks uses its own virtual communication channel.

layer, the upper modules have to deal with this issue themselves. For instance, IEEE 802.15.4's physical layer has a maximum packet size of 127 octets [99]. Subtracting the Rime and Scopes framework headers, around 100 bytes are left as payload for upper level modules. While this limit is acceptable for messages sent in RtM traffic, it falls short in MtR traffic, specially when collecting data. For this purpose, we have developed a unicast fragmentation layer, `frag-unicast` (also shown in Fig. 4.9). This module builds on the normal unicast primitive of Rime, and implements a hop-by-hop fragmentation and assembly mechanism. At the sender side, packets are divided into smaller units called fragments and sent sequentially one after the other, in order. Each fragment contains a sender ID, a packet ID and a fragment number, which suffices for reassembly at the receiver. Upon reception of a fragment, in turn, the receiver starts a timer and begins assembling the final packet by concatenating the received fragment at the corresponding offset. If two or more nodes simultaneously try to send long packets to the same receiver node, issues are likely to arise. In case of fragment *collisions*, it is not simple to reconstruct the packets (without retransmissions). In case of *interleaved* fragments, it is technically possible to reconstruct each packet. However, this would require the ability to keep track of multiple receptions in parallel, which is expensive in RAM (for example, a single 10-fragment packet requires ∼10% of the node's RAM). For this reason, fragmentation unicast only keeps track of packets from a single sender at a time. If, during an assembly process, a fragment is received from another node, the fragments received so far from the previous sender are discarded and a new packet assembly is begun. This leads to a higher goodput, as we will see in Chapter 6.

4.6 Command Runner Engine

The command runner engine is an important component in the ukuFlow framework, responsible for execution of statements in script tasks. The execution of each individual statement depends on its type; as introduced in Section 3.4, there are two main types of statements.

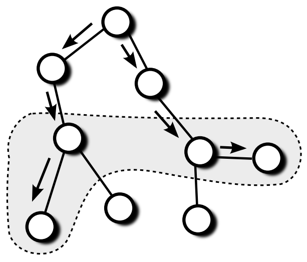


Figure 4.10.: Command dissemination through a scope

Computation statements are composed of an identifier, the *l-value*, and an expression, or *r-value*. These expressions are simply passed to the expression evaluator of the data manager module (cf. Section 4.4) running locally on the node where the workflow has been deployed – the *workflow manager node*. The result of the evaluation is then assigned to the corresponding name-value pair of the workflow instance's data repository (the name-value pair is created if it was not already present).

Function statements, in contrast, are executed by the underlying operating system's *shell*. The Contiki shell architecture is a compact service to execute commands that has a rich set of around 45 commands readily implemented (in Contiki 2.6) from which domain experts can choose. Implementing additional commands is considerably simple for WSAN experts, since these operate on the device's local resources. Execution of function statements works in two steps. First, the command name is verified for availability and the parameters are resolved (which involves evaluating expressions and variables). The output of this step is an alphanumeric string that contains the actual invocation to be passed to the shell. In the second step, the corresponding node's (or nodes') shell instance is instructed to execute the command string. Local function statements are passed directly to the Contiki shell engine running locally on the workflow manager node, while scoped function statements are disseminated by the workflow manager via the corresponding scope to its member nodes and then passed to the shell (cf. Fig. 4.10).

The message arrangement used to disseminate scoped function statements through Scopes over the network allows us to handle a large variety of command names and their parameters in a *single* network packet. For instance, when running on a IEEE 802.15.4 radio, the Scopes implementation leaves around 100 bytes of net data payload. In the current catalog of Contiki shell commands, the largest command name consists of 17 characters, which leaves considerable space for the variable-length list of parameters.

Since a lengthy task could prevent other tokens (belonging to the same or to other workflow instances) from doing progress, the command runner partitions the execution of a script task into its atomic statements, returning control to the workflow engine. This leads to an interleaved execution or processing of tasks and gateways. Next we look at how each of these statements are actually executed.

4.6.1 Synchronous vs. Asynchronous Command Execution

Abstractly speaking, statements can be executed in one of two ways: *synchronously* or *asynchronously*.

Computation statements are CPU-bound operations that perform calculations on workflow data (the exception to this are operations requiring sensor data that was not previously available or whose cached value is outdated). For this reason, computation statements are always executed in synchronous fashion.

Most commands used in function statements, in contrast, are IO-bound operations. The execution of a statement can be split into: 1) the command dissemination, and 2) the actual execution of the command. Each of these can become an important bottleneck:

1. Dissemination time:
   This depends on whether the function statement is local or scoped. Local statements communicate with the peripheral device very quickly –in the order of *microseconds*– since this communication is realized through a local bus such as SPI. Scoped commands, in contrast, require multiple *seconds* to be communicated, since multiple nodes need to be contacted over the lossy wireless connection through Scopes.
2. Actual execution time:
   Consider, for example, an operation that makes a servo rotate to a certain position. Depending on the start and end positions of the servo, alone the execution of this operation might take anywhere from dozens to some hundred milliseconds to complete.

As a consequence, function statements are always executed in asynchronous fashion. Due to the inability of the Scopes framework to send multiple messages in parallel, however, scoped function statements do block for the period during which the network is busy. This avoids network congestion and simplifies the data structures required. Scoped function statements are still processed in a FIFO basis.

A more complex alternative would consist in blocking until the command finished executing, including finishing the operation at remote nodes. This requires a bidirectional communication from the targeted nodes back to the root node to collect the confirmations that the command has finished executing successfully. In such variant, the user could additionally specify a percentage of nodes (member of the specified scope) that are required to confirm reception of the function statement and its execution.

4.7 Event Management

The last major component that completes the ukuFlow middleware is the *event manager*. In essence, the event manager implements a simple, distributed event processing engine that is in charge of processing tokens arriving at an *event-based gateway*. As presented in Section 3.5, in a uWDL model, event-based gateways are accompanied by a number of outgoing branches, each pointing to a *receive task* that contains an event expression (cf. Fig. 3.10, page 26). In this section we describe design decisions and implementation details of this component.

The goal of the event manager is to identify the first occurrence of a matching event from the specified event expressions, and then forward a token through the corresponding sequence flow. There are three steps required to process each of these event expressions:

1. *subscribe*, i.e., deploy the required subscriptions in the network by disseminating the event expression to the right subset of nodes affected;
2. *trigger* the event operators in a reliable fashion, eventually notifying the workflow manager node once a matching event is detected; and
3. *unsubscribe*, i.e., undeploy the subscriptions from the network.

The event model in ukuFlow (the hierarchy in Figs. 3.21 and 3.22) presents a modular approach to specify the events of interest by combining several *event operators* in an event script. To support this expressiveness, ukuFlow employs a design that enables connecting event operators with each other arbitrarily (but under their cardinality rules). Each event operator outputs its events into an *event channel*. Event operators that accept input are fed from one or more channels, depending on their cardinality, as specified in Table 3.3. Figure 4.11 illustrates how event channels connect event operators for a given event expression. Conceptually, event channels are the entities that connect event operators with each other or with the final consumer, both locally *within* a node and/or *across* nodes. Each event channel carries an identifier that must be unique within an ukuFlow network.

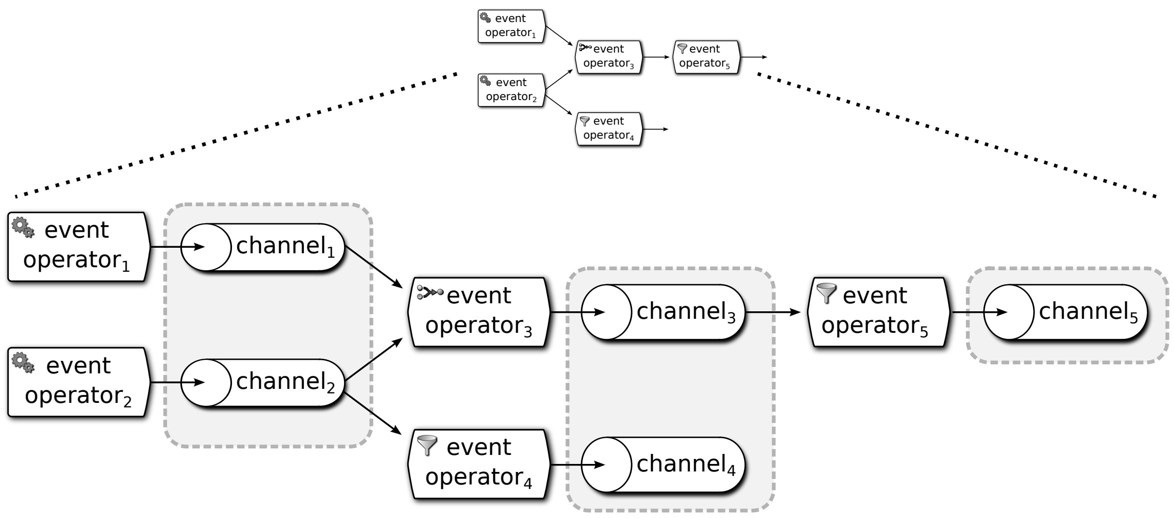


Figure 4.11.: Connection of event operators with channels.

For the design of an in-network event processing engine it is crucial to consider where in the network to execute which logic in order to detect the event of interest – the event operator *placement*. In line with the rest of the ukuFlow framework, the driving design objective is to minimize energy consumption. A naïve approach is to send the entire raw data from the nodes generating this data towards the sink node, and have the base station run the filtering, aggregation or composition operations centrally. In most cases, this approach leads to a large number of messages with the subsequent energy cost. The tradeoff between communication and processing costs in a sensor node is well-known. The current draw of a TelosB node, for instance, is 1.8mA when the microcontroller is in active mode, but 19mA when in transmit mode and 20mA in receive mode. When looking at the power consumption at byte-level, the difference is even higher: 9720nJ for transmitting 1 byte (in a 100-byte packet), but 1.35nJ to process a 16-bit instruction – a difference of three orders of magnitude, as pictured in Fig. 4.12.

As a result, it is crucial to reduce packet transmissions and shift processing into the network. A number of techniques exist to reduce the number of radio packets at all layers of the communication stack (physical, MAC, topology control and routing layers), or to optimize particular operations (e.g., detection of a maximum or minimum value with Chaos [78]). However, in this work we look at techniques that optimize the event detection at and above the routing layer.

The actual steps to perform for an event expression depend on the constituent event operators and their attributes. A determinant aspect to minimize communication is whether an event operator is able to

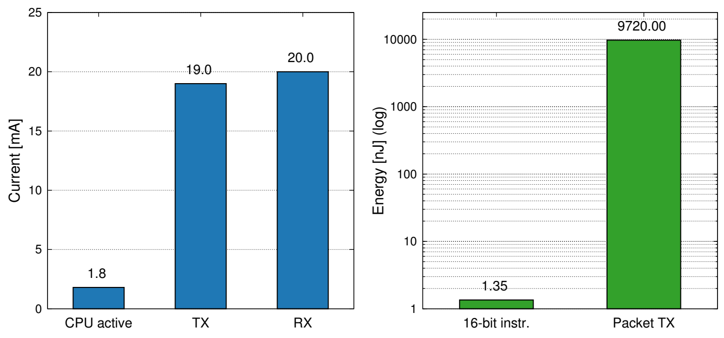


Figure 4.12.: Energy tradeoff for fundamental operations in a TelosB node. Note the logarithmic scale of the plot to the right.

Result	Event Generator	Event Filter	Event Operator												
			Event Composer												
			Logical			Processing Function						Temporal	Change		
			AND	OR	NOT	MIN	MAX	COUNT	SUM	AVG	STDEV	SEQ	INC	DEC	REM
a) local	x	x													
b) in-network			x	x	x							x			
c) at root			x	x	x	x	x	x	x	x	x	x	x	x	x

Table 4.3.: Result computation of ukuFlow's event operators.

produce its output locally on a node alone from its input. In general, this is the case for event generators and simple event filters. Event composers, in contrast, normally require aggregating or reducing a set of input events to produce an output event, and thus inherently involve network communication. Some of the event composers can readily compute the final result within the network, while others require combining information from all the affected nodes, e.g., at a root node. Table 4.3 summarizes the analysis of where the final result can be obtained: local, in-network or at root. Before we jump to the event expression deployment and execution plan, we describe the mechanisms to deal with event streams and event compositions.

4.7.1 Event Composition Mechanisms

When composing multiple events into higher-level ones, there are two aspects to consider to disambiguate the event algebra, which we describe next.

Dealing with Event Streams

First, we must define how to deal with the potentially *unbounded* stream of input events. In particular, event operators that require the entire input set of events, like `min` or `count`, would not return an output event until the input stream concludes. This is typically solved by analysing *finite* portions of the input set by specifying an *input window* over which the event operator is applied. A comprehensive comparison of types of windows is provided in [20], here we present window concepts of interest to WSANs.

Windows can be defined as a *logical* interval, specified by two input events (a start and end input event), or as *physical* interval, based on time. Physical windows are defined by a window start time, ws, and window end time, we. A window's start and end can be updated with a policy, in which case it is called a *sliding* window, i.e., its start and end events, or its start and end times (ws and we) change continuously. Physical sliding windows can be of three types: rolling, tumbling or disjoint. Consider two successive, sliding windows w_1 and w_2. A rolling, or overlapping window is such that $w_1.we > w_2.ws$; a tumbling window is such that $w_1.we = w_2.ws$, and a disjoint window has $w_1.we < w_2.ws$ (cf. Fig. 4.13).

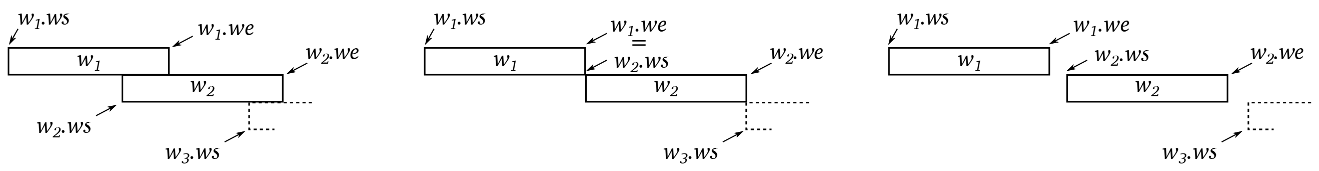


Figure 4.13.: The rolling (left), tumbling (center) and disjoint (right) window mechanisms

While the window mechanism reduces the input events to consider for an event composition to a certain finite interval, it is still possible for an operator to receive, within a window, many input events from which alternative output events could be generated. Consider, for example, the partial event script `AND_EC (eo`$_1$`, eo`$_2$`)`. If, during the event composition procedure, the sequence of events e_1, e_1', e_2 occurs (where e_1 and e_1' are events matching the nested operator eo$_1$ and e_2 matches eo$_2$), a rule is needed to determine which of the combinations (e_1, e_2) and/or (e_1',e_2) are generated as output events.

In [21], Chakravarthy et al. introduced the concept of *parameter contexts* to restrict the set of input events to use in an event composition. Parameter contexts combine both composition rules and consumption policies (therefore also known as *consumption modes*). To understand these contexts, the authors consider an *event log*, i.e., a buffer where the entire set of occurrences of input events is stored. The following four contexts were identified:

- *recent*, where only the most recent occurrence of the input events are composed and consumed from the event log. Newer input events received before all necessary input events are available simply replace the previous input events of the same type. In this consumption mode *not all* input events are used. This is very reasonable in a sensor network application where the user expects the most up-to-date information to take a decision.
- *chronicle*, where the oldest input events are combined and consumed from the event log. Newer input events received before an output event is produced are kept in the event log, until consumed. In this consumption mode, input events are used *at most once*. This context is used whenever events must be processed in order of arrival.
- *continuous*, where each initiator input event leads to a possible combination. An initiator event is removed only after at least one combination was possible with it. In this consumption mode, input events are used *at least once*. This context is useful in scenarios where multiple detections need to be considered.
- *cumulative*, where all possible combinations are produced as soon as the composed event is detected, i.e., *all* constituent input events are used. (Note that the cumulative context is equal to the continuous context when only considering an event log of a single event composer. Differences arise when considering an event script with nested event composers).

We argue that Chakravarthy's parameter contexts are less suitable for a WSAN for two reasons. In the first place, context computation places unlimited or high storage requirements (except for the recent context):

- *recent* is $O(n)$, where n is the number of event operators in the overall event expression, each operator buffering a constant number of input events depending on its cardinality.
- the requirements for the rest, *cumulative* < *continuous* < *chronicle*, is theoretically unbounded, although it could be restricted to $O(n \cdot d)$, where d depends on the composite event duration and the frequency of occurrence of input events. For example,

```
TOP = AND_EC ^2:00 (eo$_1$, eo$_2$)
eo$_1$ = PERIODIC_EG SENSOR_CO2_PPM ^0:10 @floor1
eo$_2$ = PERIODIC_EG SENSOR_CO_PPM ^2:00 @floor1
```

  In this case, the `AND` event composer's buffer would require 12 slots. Using the continuous and cumulative contexts would result in 12 output events.

The second, and more important reason is that these parameter contexts assume the availability of the event log as a *global* buffer that can be analysed centrally. This is analogous to concentrating all the information at one (or more) nodes in order to perform the detection, which we want to avoid in the first place.

Finally, providing support for different parameter contexts is not viable in a resource-constrained network of sensor nodes. While the choice of window type or parameter context is dependent on the application, to realize an in-network event detector it is necessary to sacrifice some flexibility. We next describe the considerations to make event composition feasible in ukuFlow.

4.7.2 Event Composition in ukuFlow

In ukuFlow we restrict an event composer's input event set exclusively through *tumbling* windows defined by a time interval. Nodes participating in the event detection initiate the tumbling window mechanism as soon as the event operator is deployed on them. This is implemented by means of a Contiki callback timer that is reset periodically (leading to stable intervals over time). Since the dissemination of the subscription over a multi-hop network does not occur instantaneously, minor asynchrony between the node's windows exists. In Chapter 6 we investigate the jitter effects and discuss mechanisms to palliate it further.

Second, in this work we propose a new parameter context that relaxes some of the properties of the previous contexts in favor of accommodating a low-power data collection mechanism, and that we call the *collection context*. The collection context works on a per event operator-basis, i.e., each event operator computes its context based on an own event log. This is different from the previously described contexts which looked at a global event log. Event scripts which contain multiple, nested event composers will thus have multiple event logs, one per event composer, with a buffer size given by their cardinality. In these logs, the order of events is based on the arrival time.

In cases where the event composer receives its event inputs directly from multiple nodes (i.e., over a scope), the collection context employs a distributed approach where each node in the network participating in the collection mechanism has a slot within the operator's tumbling window in which it partially contributes its computation to the final event combination. This is similar to the concept of interval (a division of an epoch) in the Tiny AGgregation service (TAG) built into TinyDB [87], and is the crucial aspect enabling the reduction of network packets.

Finally, the collection context only uses the most recent event inputs observed by a node in the interval. A node receiving multiple events from the same input channel within a slot simply replaces the old input for the new one.

The ukuFlow event manager receives subscription requests from the workflow engine, and interprets them to decide how to disseminate the event script and to whom. Next, we detail the deployment and execution procedures that employ the collection context.

4.7.3 Event Script Deployment Plan

Cost-effective sensor network deployments will be composed of a landscape of heterogeneous nodes, each fulfilling different roles [137, 115]. It is thus not reasonable to flood the entire network with the event description (unless explicitly needed by the application). Instead, only those nodes that must participate in the event detection should be addressed. The goal of this procedure is to create an *event script deployment plan* that minimizes the set of nodes addressed.

For this purpose, the event manager scans the bytecode of the event script. This bytecode specification is organized in polish notation – similarly to the data expressions presented in Section 4.4. The event manager traverses the script through the event operators, and inspects the specified destination of each event generator found, adding the destination to a set T of *target* destinations that need to be contacted. There are 3 possible destinations for an event generator:

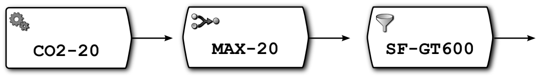
- `local` → event generator must be deployed locally at the event manager node,
- scoped → event generator must be deployed at the respective scope, or
- `world` → the entire set of nodes is required to participate in the generation of these events.

As a result of this scanning process, the set T will consist of a number of scope IDs, the `local` node and/or the `world` scope. In cases where `world` $\notin T$, the deployment plan consists in disseminating the corresponding event operators to the identified targets (scopes and/or the `local` node). If, by the contrary, `world` $\in T$, the deployment plan simply consists in flooding the event script once through the entire network – an inevitably expensive plan.

We illustrate the general procedure with the previous example for detecting high CO_2 concentrations (which we replicate below for convenience). Figure 4.14 shows the steps required to disseminate the subscription. In the left part, 4.14a, the entire set of nodes and links between them is shown[1]. Dotted lines represent 1-hop neighbors, and straight lines represent the parent-child relation, which builds the Scopes' overlay tree routed at the node with ID 1. Fig. 4.14b presents a snapshot of the scope membership for two scopes S_1 and S_2. Note that for each of these scopes to be created, the entire network needs to be flooded as described in Section 4.5, since it is not known in advance which node might belong to the scope or not. We note that:

- nodes 2 and 5 are member both of S_1 and S_2;
- node 3 is member of S_1 but not S_2, nodes 6 and 7 are member of S_2 but not S_1;
- nodes 4 and 8 are not member of S_1 nor S_2, but node 4 is *forwarder* of S_2 due to its child node 7.

```
TOP = SIMPLE_EF [ MAGNITUDE > 600 ] ( maxF )
maxF = MAX_EC ^0:20 ( co2egen )
co2egen = PERIODIC_EG SENSOR_CO2_PPM ^0:20 @floor1
```



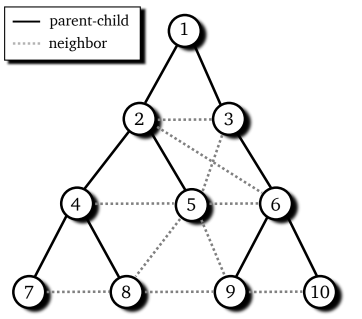
Scanning the event expression results in $T = \{$`floor1`$\}$. The event manager requests the creation of the *interceptable* scope `floor1` and disseminates the event script through it. We consider two scenarios for `floor1`: S_1 and S_2. Since nodes know whether they have (direct or indirect) member children below them, the actual traffic for the dissemination through S_1 requires only two broadcast messages (from nodes 1 and 2) to reach all the depicted member nodes 2, 3 and 5. If the subscription was targeted to S_2, four broadcasts[2] would be necessary (from nodes 1, 2, 3 and 4).

The event manager sends the *complete* event script through the scope instead of only the event generator that is needed at each scope since other event operators might be needed in the network. The subscription message that is disseminated is typically very compact and taken directly from the workflow specification's bytecode.

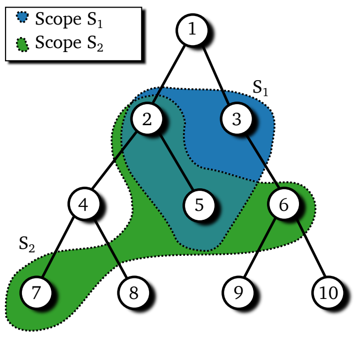
We note that the dissemination of the event script to *multiple* scopes requires one call to the `send` primitive *for each scope*, since the Scopes API does not allow sending a message to multiple scopes simultaneously (i.e., with one single call). The efficiency of multiple dissemination calls vs. a single flood depends on the network topology and the actual set of nodes that are member of each scope. In the worst case,

[1] For clarity of the discussion, we assume that all links are bidirectional.

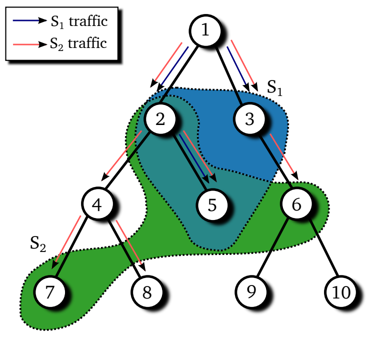
[2] Since data is disseminated with the `netflood` primitive (trickle-based), this scenario requires *at most* 4 broadcasts, but possibly fewer for the depicted network topology.



(a) Network topology and overlay tree



(b) Instantaneous scope membership



(c) Traffic over respective scopes

Figure 4.14.: Subscription dissemination on an overlay tree, highlighting the traffic optimization to reach only the necessary nodes.

all target scopes contain *all* nodes in the network, and thus the cost of the deployment plan is $n-1$ times higher than that of a single flood (where n is the number of target scopes). Here is where the selectivity features of the Scopes framework play an important role, enabling a fine-grained scope membership, and make the cost of multiple individual send calls cheaper than a single network-wide flood.

Also, the dynamic nature of the sensor network makes it difficult for the scope root node to know, at all times, the current topology or the mapping of nodes to scopes, since in essence the scope creation process is a unidirectional procedure (i.e., from the root towards the rest of the nodes) with very limited feedback. Having an approximation of this distribution, however, would enable optimizations of the subscription deployment plan – an aspect that opens new avenues but is left as future work.

Upon *reception* of a subscription message, each node needs to determine its role. Nodes process the received event script recursively, in *post-order* fashion. For every event generator found, nodes follow these rules:

1. if the node is *member* of the event generator's target scope, the event generator is instantiated locally, and the processing continues with considering the parent event operator (i.e., backtracking)
2. if the node is not member but *forwarder* for the event generator, the generator is not installed on this node, but the processing continues with considering the parent event operator (like in the previous case)
3. if the node is *neither* member nor forwarder of the event generator, the processing of the event script continues until the next event generator

In both cases 1) and 2) the processing continues with the parent event operator. In the case that the parent event operator is a simple filter, the operator is instantiated locally, and the processing continues. If the operator is an event composer, it also must be instantiated locally, but processing for this branch of the event script is stopped. This is because the final result for an event composer will be obtained only at the root node of the corresponding scope, hence no other operators will be able to obtain an output event from the event composer at a leaf or inner node.

Finally, the root node also participates in the event detection, since it is forwarder to all scopes. As such, event composers and their parent operators are likewise instantiated at the root node.

We illustrate the instantiation of event operators with the previous example using S_2. Figure 4.15 depicts, on the left, the event script diagram for the original event expression (note each event operator coded in a different color), and on the right, the network with an indication of the operator placement. Nodes 8, 9 and 10 do not participate at all; nodes 2, 5, 6 and 7 participate in both the event generation and the identification of the maximum value; and nodes 1, 3 and 4 participate only in the composition. The final result of the composition is obtained at the root node, which is the only place where the simple filter is instantiated. In total there are 12 instances of event operators for this event expression placed across the depicted network.

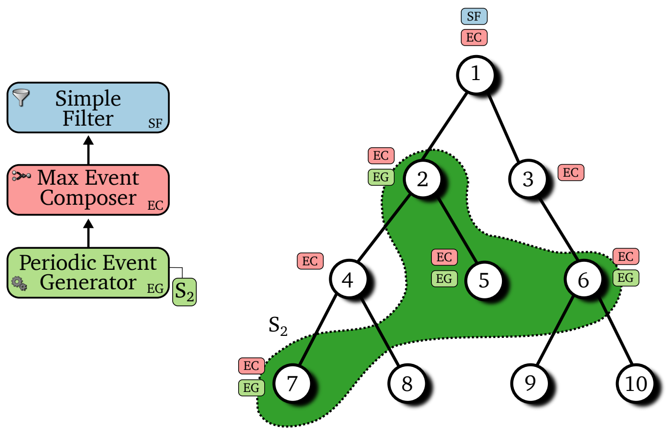


Figure 4.15.: Event script diagram and operator placement for original CO_2 example

In Fig. 4.16, we modify the event expression (and thus its event script diagram) by swapping the order of the simple filter and the event composer as presented to the left. The difference resides in that now, nodes with event generators also run the simple filter (which is not placed any longer at the root node). The modified expression results in an event operator placement with a total of 15 instances. Although in this latter case the final output event of the modified expression is semantically equal to the original one, it leads to less network traffic due to the earlier selectivity of the filter operator – events not passing the simple filter will not be used in a composition.

Unfortunately, this modification can not be applied in all circumstances, since the resulting event might be different. In particular, such a swap is safe if the simple filter's constraints operate on the magnitude of the event, and if the composing operator is of *exemplary* nature, like `min` or `max`. It is otherwise not straightforward to automatically determine if a swap yields a semantically equivalent result. With the current design, users specifying the event expression need to be aware of this implication. Further investigation in an analogous direction to *query optimization* in databases is left open.

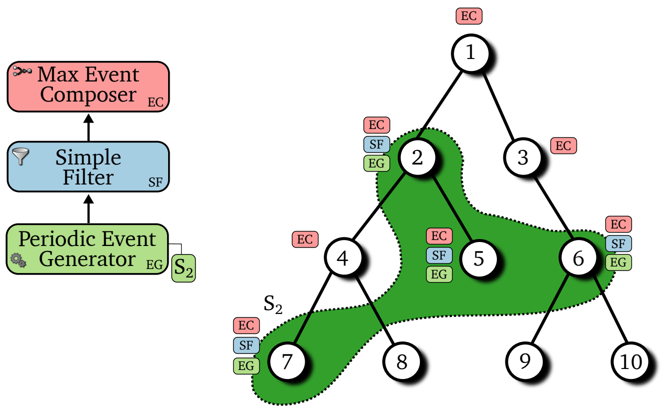


Figure 4.16.: Event script diagram and operator placement for modified CO_2 example

The *unsubscription* procedure is analogous to the subscription deployment plan: each participating scope is contacted to remove the event operators instantiated for the event script, also releasing the allocated resources and deleting events that might have been generated and produced in the meantime.

Network Dynamics

The dynamics of the network are an important aspect to consider. Topology changes occurring at runtime due to nodes (temporarily) leaving the network or new nodes joining it pose the difficulty that the overlay routing tree breaks – and with it, the mechanisms that rely on it. A node that is temporarily isolated from the rest of the nodes causes its child nodes to become disconnected from the rest of the network. Events sent from child nodes to the unreachable parent node get lost, and since upward traffic is not reliable (i.e., not acknowledged), the child nodes continue their operation as usual, unaware of the issue. At some point in time, however, these child nodes will notice that they have not received the scope refresh message disseminated by the root node, and will thus leave the scope (the time or number of refresh messages that must be missed is configured through a policy in Scopes). The event manager implements a cleanup mechanism so that when it is notified that the node left a scope, all event operator instances associated to that scope are also freed, similarly to what is done when an explicit unsubscription request is received.

For a node to join (or return into) the network and start contributing to the detection of an event, there are three mechanisms in place. First, the scope root node periodically rebuilds the overlay tree, which enables new (or existing) nodes to create a routing entry towards the root (or update an existing route with a better one, respectively). Second, each scope is refreshed at a specified interval, which lets nodes discover the current scope specifications and evaluate their membership to them. Finally, the event manager re-announces the actual event scripts periodically, allowing nodes to create (or update) local event operator instances according to the previously described rules.

In conjunction, the automatic removal of event operators and scopes, and the periodical scope refresh and event script reannouncing mechanisms support nodes in establishing a consistent state with regards to the current workflow operator being executed, and address topology changes at runtime.

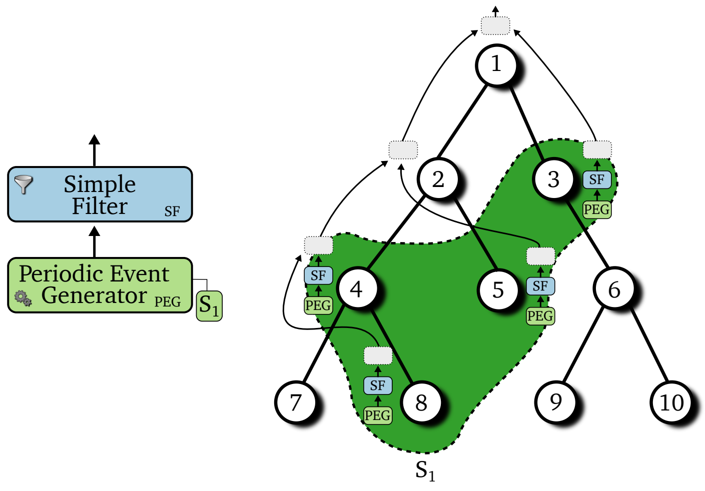
4.7.4 Event Generation and Processing Mechanism

We now describe how the event generation and processing takes place within the network. Once a node receives an event script deployment request, event operators are instantiated by creating a number of objects in the node's RAM. The subscription request includes all the necessary information that an event operator instance requires to begin carrying out its task, such as timer intervals, sensors to query and tumbling window size.

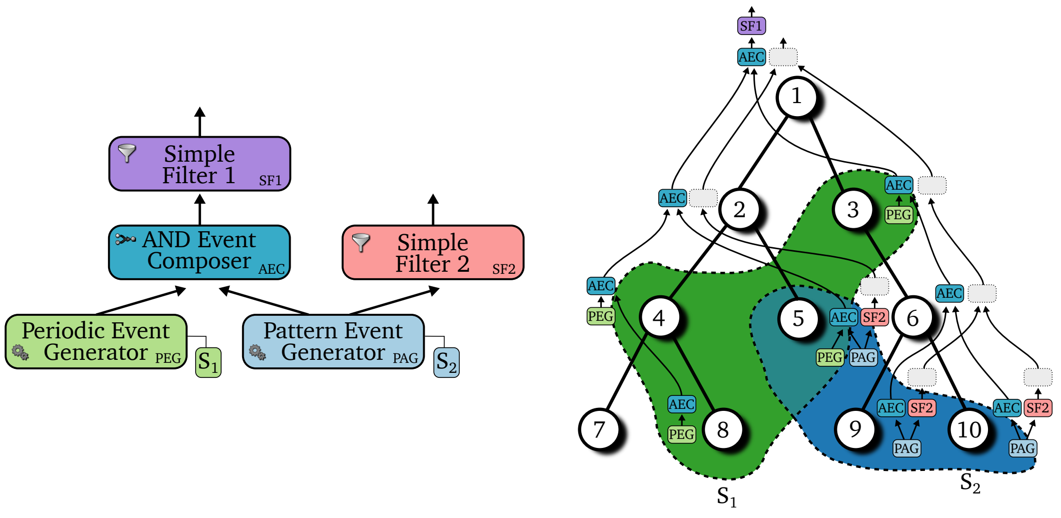
An important attribute for the correct processing of events is the ID of the event channel into which the event must be published. Within a node, each event operator works independently of the others, only communicating indirectly through the event channels. Whenever an event operator is instantiated, its inputs are configured to consume events from the event channel of the nested event operator. These channel subscriptions are kept in a local list of *running* event operators.

The processing of events is then straightforward and consists in iterating through the list of running event operators, and passing the event to each matching subscribed event operator. In the case that no event operator is found locally, the event must be forwarded to the parent node. Ultimately, at the root node, the output of an event operator is the final matching event and is thus passed to the upper layer's subscribing application, which is a callback function in the workflow engine. This concludes the detection of an event. The workflow engine then knows which of the outgoing branches of the *event-based gateway* needs to be activated, and hence requests the unsubscription of all of the gateway's event scripts. Events received afterwards, but before the unsubscription process is concluded, are discarded.

We illustrate this procedure with two examples. Figure 4.17a presents on the left side a simple event script containing two event operators. Since these operators can obtain their output locally, they both are pushed deep in the network. The right side depicts the network topology, the event operators deployed on each node, and also the implicit event channels among them. Grayed boxes represent the case where the output of an event operator does not find a local consumer, in which case the event must be forwarded to the parent node in the tree, as mentioned above. Node 8, for instance, begins the processing when the periodic event generator creates an event. This event is consumed locally by the simple filter operator. The output of the simple filter (if any), is not consumed locally, so it must be passed to the parent node 4. That event does not find a local consumer (note that in this example there is no aggregation), and thus is forwarded to node 2, and then to node 1, where it is used as a notification to the workflow manager. A similar process is followed for events from nodes 3, 4 and 8.



(a) Example with simple filter



(b) Mixed example with shared event operator

Figure 4.17.: Event script diagrams and their event channels connecting local and remote event operators

The second example (Fig. 4.17b) illustrates the deployment of two event scripts that, in addition, share an event operator: the pattern event generator. The figure on network topology and operator placement contains several important cases:

- At node 8, the periodic event generator feeds the AND composer which, in contrast to the previous example, does not produce a final output event locally, but an intermediate result that is forwarded to the parent node's instance of the event operator. This intermediate result is known as partial state record in TAG ([87]).
- At node 4, the AND event composer combines the local input event with that one coming from the child node (using the collection context as described previously). This intermediate output is forwarded to node 2 and then to node 1, where the final output event is certainly available.
- Node 5 is the busiest from the network because it is member of both scopes S_1 and S_2. The output event from the periodic event generator feeds the AND composer, while the output event from the pattern event generator feeds *both* the AND as well as the simple filter SF2. If the AND composer at that layer receives the input events that satisfy the conjunction in its corresponding slot of the collection context, it can forward an intermediate event with this information. Events output by the simple filter SF2 are forwarded directly towards the root.

Note that event composers which receive their input from other event composers are deployed at the root node, similarly to the simple filter SF_1 in the second example.

Realization of the Event Manager Services

Subscriptions, unsubscriptions, as well as event processing requests, are asynchronous operations. These are packed in a *request object* and put into a queue, which enables continuing with the next task immediately. The processing of these requests is carried out internally by the event manager node with a separate thread. This enables a correct serialization of operations.

Event operators have all a common structure, composed of a name and four functions for:

- initialization of the operator,
- removal of the operator,
- consumption of an event, and
- evaluation of an event.

At a node's boot-up time, the event manager initializes a vector containing the available event operators' structures. This organization facilitates the localization of the callback function pointers for the processing of an event. Extending the event manager with further event operators simply requires adding a pointer to its structure in the vector.

4.8 Reliability Considerations

The design of the ukuFlow's runtime engine enables an in-network execution of workflows, making external infrastructure superfluous. The system presents a semi-distributed architecture that tolerates many failures (in particular, fail-stop failures), as well as node churn, occurring at nodes at intermediate and leaf levels of the overlay tree. In these situations, the workflow execution continues with a degraded quality. For example, if a workflow instance requires obtaining the average temperature from a scope, and a fraction of its member nodes fail, the average will consist only of the nodes that remain alive. Routes will be adjusted dynamically, and the workflow execution will proceed with this inexact value.

Failures at nodes acting as workflow managers for a given deployed workflow, however, are more severe, since they cause the deployed workflow instances to be lost. Therefore, manager nodes are single point of failures, but only for the workflows deployed on them (i.e., their failure does not prevent other nodes that are managers for other workflows from continuing their execution).

Approaches to increase the tolerance to failures, however, are interesting to consider. Redundancy can be added by replicating the workflow management functionality to multiple nodes. Upon workflow deployment on a node, called the leader node, a subset of alternative nodes receive replicas of the workflow specification. If, later, it is detected that the manager node fails, the alternative nodes run a leader election process and the winning node takes over. The larger the set of alternative leaders, the more reliable the execution, but the higher the maintenance overhead. An initial implementation of this functionality for the Scopes framework was carried out in the context of this thesis by the exchange student Ignacio Brasca [14], the more general version of the problem was investigated by Guerraoui and Schiper in [50].

4.9 System Code Distribution and Workflow Upload

The system's runtime is installed on sensor nodes through the traditional reprogramming mechanism, i.e., the bootstrap loader. This is a one-time operation that accesses the sensor node's program space via a USB-to-serial converter, rewrites the node's ROM with the ukuFlow firmware and reboots the node with the new image.

For the deployment of workflows, a special module called `ukuflow-serial` is also included, which is in charge of opening a connection through the serial port and accepts workflow (un)deployment commands.

The BPMN2uku plug-in offers a context-sensitive menu, reachable from the IDE's package explorer, through which the user can request the validation and generation of ukuFlow bytecode for a selected workflow. After passing all formal validations, a workflow deployment command can be sent via the USB port of the developer's workstation to a node directly connected to it.

The bytecode representation of a workflow can contain zeroes in the middle of the stream. To avoid the serial-line driver from interpreting these characters as *end of lines*, the entire (un)deployment requests are encoded into Base64 on the PC side and decoded by the `ukuflow-serial` module back into its binary version. The overhead in workflow size (of around 33%) for the transmission is not a major issue at runtime, since the Base64 representation is not retained in memory after decoding.

Availability

The ukuFlow software is available under a BSD license. The system is split into two parts: the runtime for the sensor network, and the Eclipse plug-in. The system runtime is available via a Google Code public repository project page at `code.google.com/p/ukuflow`. The plug-in can be obtained and installed into Eclipse's Juno release, from a download site at `download.dvs.informatik.tu-darmstadt.de/ukuFlow`.

4.10 Summary

In this chapter we have presented the requirements and high level architecture of the ukuFlow framework. Adhering to the Reference Architecture of the WfMC, we presented the separation between the build-time functions, i.e., the IDE plug-in, and run-time functions, i.e., the ukuFlow runtime.

The plug-in, BPMN2uku, is a tool to design, compose and edit workflows using the ukuFlow WDL described in the previous chapter. The plug-in furthermore offers means to validate, convert, deploy and undeploy workflows onto sensor nodes connected to the workstation.

The ukuFlow runtime faces the hardest challenge of fitting a relatively complex workflow system that runs autonomously on a resource-constrained sensor network. We have presented the architectural decisions that make the ukuFlow system feasible for mote-class devices. We demonstrated how a two-level scheduler can be used to execute workflows in an asynchronous way, enabling pseudo-parallel execution of workflows and their instances. The usage of Scopes as networking component has been shown to be effective, although it requires a number of adjustments for the efficient in-network processing of events. Finally, we presented various mechanisms for the deployment of event script plans that resort to a modular approach which causes only related nodes to participate in the event detection procedures.

Such a system offers many perspectives on which to analyze its performance and efficiency. The next chapter paves the way for a systematic and realistic evaluation by presenting a set of testbeds that enable a rigorous analysis of sensor network systems.

5 Empirical Evaluation of Sensor Network Systems

An individual developer like me cares about writing the new code and making it as interesting and efficient as possible. But very few people want to do the testing.

Linus Torvalds (1969-*)

In this thesis we make particular emphasis on the feasibility of a workflow system for real-world sensor networks. Such analysis is only possible by going beyond simulation and evaluations with few nodes. Indeed, empirical experimentation has slowly become an important requirement for high-quality research in WSNs, as can be observed in work presented at premiere conferences. In this chapter, we step back from the main problem of providing an easy to use framework for the domain expert, and delve into the challenge of realistic evaluation of sensor network software. We discuss the ongoing efforts to build the TUDμNet testbed federation, and present the results achieved so far.

5.1 Simulations vs. Real-World Deployments

Current research in low-power sensor/actuator networks has mainly concentrated, on one side, on lab work and simulation experiments that are reproducible yet simplified in nature, and on the other side, on realistic deployments that show feasibility yet make it difficult to explore parameters. Sensor network *simulators*, like COOJA [107] or TOSSIM [82], facilitate parameter exploration and are able to scale up to thousands of nodes, but do not always capture all phenomena from the target environments. Working directly on *deployments* in situ, as done at the Great Duck Island [130] or precision agriculture scenarios [79], exposes the system to the conditions of the real environment, but logistical hurdles, such as mounting hardware, installing batteries, programming (i.e., flashing) sensor nodes and instrumenting them for experiment data collection make it difficult to repeat experiments, and even harder to make them reproducible.

The practical solution between simulations and realistic experimentation are *testbeds* (cf. Fig. 5.1). A testbed comprises a real sensor network instrumented with an additional infrastructure to, e.g., provide a permanent energy supply (in contrast to the traditional battery source), as well as equipment for logging node's activities. Testbeds provide users with the functionality to define experiments, e.g., selection of sites and node types, scheduling of jobs, and validation of experiment parameters.

Related work in testbeds includes projects such as TWIST [60], an indoor testbed including 204 nodes (TelosB and eyesIFX), where users resort to a set of scripts to indirectly program sensor nodes and collect debug data. MoteLab [142] (no longer active) included around 190 TelosB nodes spread through offices in a three-story building, where test jobs were defined and scheduled through a web interface while debug data is logged into a centralized database for later evaluation. The Kansei testbed [5] features higher sensor node heterogeneity at a comparable scale (15x14 grid with Stargates and XSM nodes).

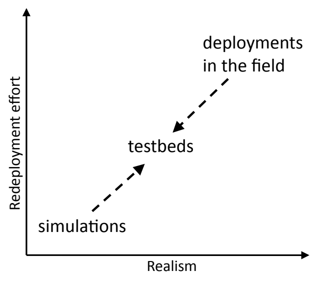


Figure 5.1.: Testbeds tend to lead to more realistic experiment data than simulations, while their software redeployment effort is generally lower than in situ studies.

In order to reach an even larger scale, testbed *federations* have emerged. In the EU, the WISEBED project [25] aimed at aligning several testbeds located in multiple european countries through a unified, loosely coupled management interface, while in the U.S., the KanseiGenie [124] had a similar aim. There is a growing interest from both academic and industrial researchers in the efficient, cost-effective construction of WSAN testbeds. Most testbeds, however, have been built using an unsystematic, best-effort approach, which leads to systems that exhibit high deployment efforts, high maintenance and operation costs, and do not offer the level of precision and validity required in research.

This chapter presents our ongoing efforts in addressing this issue through the development of TUDμNet, a *metropolitan-scale* federation of sensor network testbeds that spans several sites within the city of Darmstadt. Table 5.1 summarizes the aforementioned experimentation approaches, and puts our work in context. Section 5.2 provides an overview of TUDμNet, and describes its constituent sites. We describe our control infrastructure as a solution for managing a variety of experiments at a metropolitan scale, present the challenges of building a reliable backchannel, and discuss the related applications.

	realism	scale	control	examples
deployment	+++	++	+	GDI [130], Agriculture [79]
simulator	+	+++	+++	TOSSIM [82], COOJA [107]
testbed	++	++	+++	MoteLab [142], TWIST [60], Kansei [5]
federation of	++	+++	++	WISEBED [25], KanseiGenie [124]
testbeds	++	++	+++	**TUDμNet** [52]

Table 5.1.: Approaches to doing sensor network experimentation, with prolific examples from past and current projects. As a metropolitan-scale federation, TUDμNet is a highly controllable testbed with focus on realistic and reproducible benchmarking.

5.2 TUDμNet Overview

Similarly to other testbeds, TUDμNet's architecture is structured in three tiers (cf Fig. 5.2). The first tier is composed of the sensor nodes which run the software being tested. This can be generated from a normal `build` system like Contiki's or TinyOS's. Our testbed currently contains a mixture of TelosB, Z1, G-Node and XM1000 nodes, all based on MSP430 microcontrollers. In turn, each node's board is populated with a variety of sensors. The second tier is composed of simple gateways which are permanently connected to a number of sensor nodes via USB cabling. Between 2 to 10 sensor nodes are managed by a gateway, and each testbed is currently composed of 1 to 30 of these gateways. For this tier, we have opted for network routers such as the Buffalo WZR (on which we run OpenWRT), as

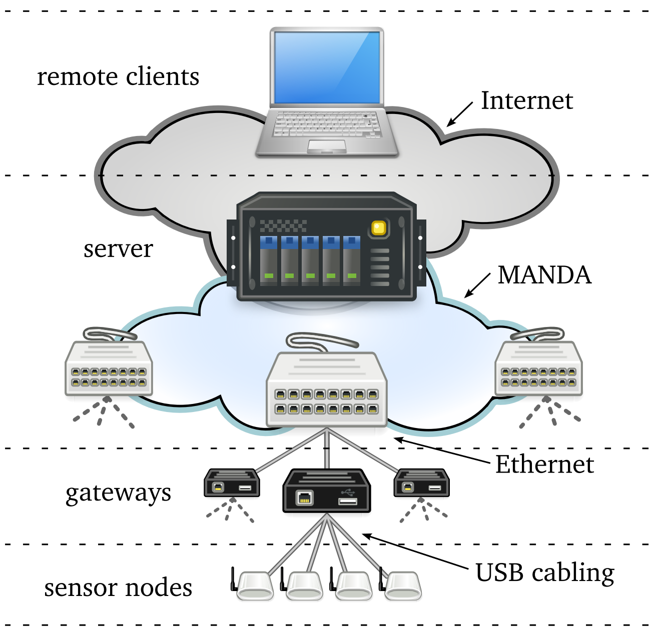


Figure 5.2.: TUDμNet's architecture

well as Raspberry Pi devices. These are customized with sensor node management tools such as a serial forwarder and the bootstrap loader. Finally, a central server orchestrates the entire federation activities and stores the experiment results. The traffic between the various testbeds (and in turn their gateways) and the server is routed through MANDA (Metropolitan Area Network DArmstadt), which is operated at Gbit/s speed.

5.2.1 Testbed Sites

Metropolitan areas typically span a number of different environments: urban areas with legacy buildings, green parks, commercial/industrial districts, and even new neighborhoods following modern construction techniques. TUDμNet matches these areas with corresponding experimentation playgrounds that remain of a manageable scale, yet offer enough scientific fidelity to yield results applicable to similar environments. The TUDμNet federation spanned four sites [51], three of which are currently operational. Each of these testbeds were motivated by certain well-defined scenarios, as presented next.

Old Buildings

In Europe, around 40% of energy consumption is due to building usage [92]. Buildings also are the largest source of CO_2 emissions [61]. Since energy is used mostly during the operational stage (i.e. during user occupation), sensor networks become a key element for monitoring building use and enabling intelligent control of, e.g., heating, ventilation, and air conditioning (HVAC) systems.

The testbed constructed at the main (Piloty) building of the Dept. of Computer Science of the TU Darmstadt enables the development of this type of applications. This is a 3-story building renovated after its first erection in 1937, consisting mainly of offices. The testbed at this site currently spans 25 offices in the north wing, each with 2 to 4 TelosB and XM1000 sensor nodes. Nodes are equipped with an MSP430 micro-controller unit (MCU) and an 802.15.4 radio operating at 2.4GHz. The sensors attached to the nodes can measure temperature, humidity, and light intensity. Figure 5.3 depicts a typical office deployment. In each office, a Buffalo WZR-HP-G300N acts as gateway (octagon), which bridges departmental Ethernet with the nodes (circles) through a USB backchannel. This rather unconstrained environment has shown its own challenge: the USB backchannel. Figure 5.4 presents the positions and IDs of the nodes deployed in the first floor.

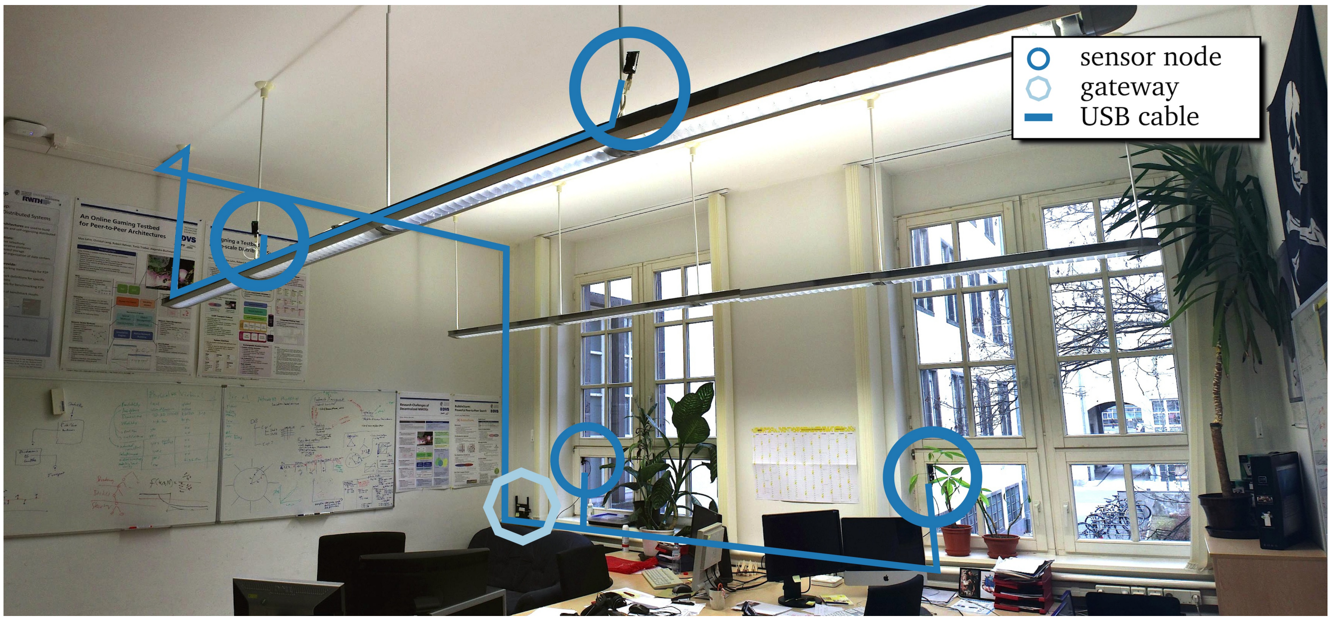


Figure 5.3.: An office in the Piloty building, with installed wireless sensors (circles) that are also attached to a gateway (octagon) for quick reprogramming.

Emergency Response

The usage of technology for supporting first-responders and their operations in emergency situations has become a critical tool for metropolitan cities. Identifying elevated physical or chemical concentrations of radioactivity, gases or fire, as well as their temporal spreading, is a challenging task where the interaction between autonomous aerial and terrestrial vehicles comes into play.

The joint laboratory of the Research Training Group 1362, *Cooperative, Adaptive and Responsive Monitoring in Mixed Mode Environments*, pursues exactly this goal. Located at the Technologie und Innovationszentrum (TIZ bldg.) in Darmstadt, the site features multiple offices and a larger disaster scenario arena constructed following the guidelines of the RoboCup Rescue competition [106], monitored with gas sensors, uniformly spread in a 5x12 grid organized into multiple *arrays*.

In this deployment, each of the 60 TelosB nodes counts with an external sensor board on which additional CO, CO_2 and temperature sensors are attached. Due to the increased power consumption of these sensors, the installation of an additional power line was necessary, as depicted in Fig. 5.5. The site presents a dense network where, at maximum transmission power, most nodes can reach all others. Experiments that require it can reduce the transmission power, which yields topologies of higher network diameter.

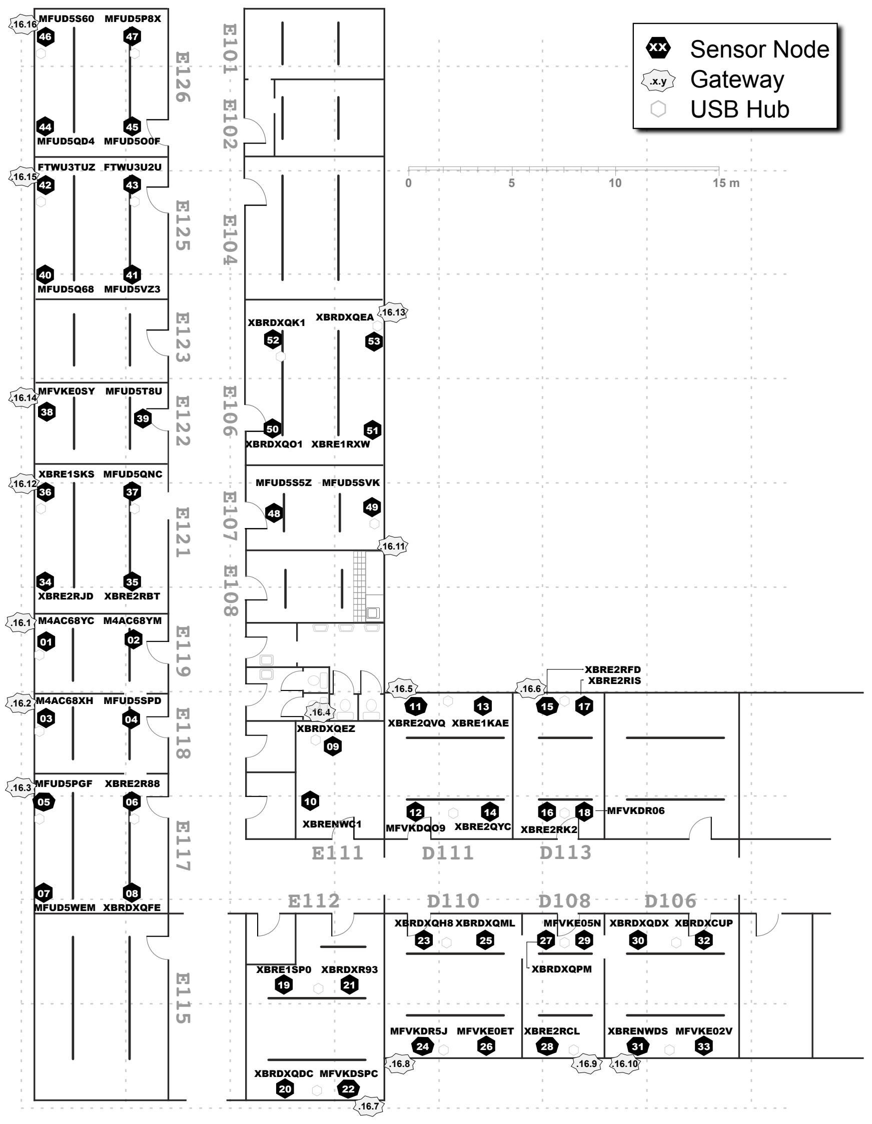


Figure 5.4.: Map of 1st. floor of the TUDμNet deployment in the Piloty building

Energy Efficient House Construction

The emergence of decentralized, micro-scale renewable energy sources (especially photovoltaic and geothermal heating/cooling) has led the sustainable construction of energy efficient residential homes into an interdisciplinary area of investigation beyond architecture and civil engineering, prompting ICT systems to come into play. The explosion of construction techniques and modern materials mobilized researchers and practitioners to establish a biennial competition, the Solar Decathlon [105], where participants measure their innovations applied to residential properties at a number of contests. These houses represent the state-of-the-art in low ecological footprint.

While many construction aspects are designed and validated through models and simulation, critical aspects of the construction remain unclear until a prototype is built: Do the HVAC systems work as expected throughout the inner space of the house? Are the (costly) materials of the ceiling and exterior walls correctly designed to tolerate the weather conditions to which they are effectively exposed (varying temperature and humidity levels)? Are solar panels acting optimally and delivering the maximum amount of energy as originally planned? Does the geothermal heat pump tunneling deliver the expected

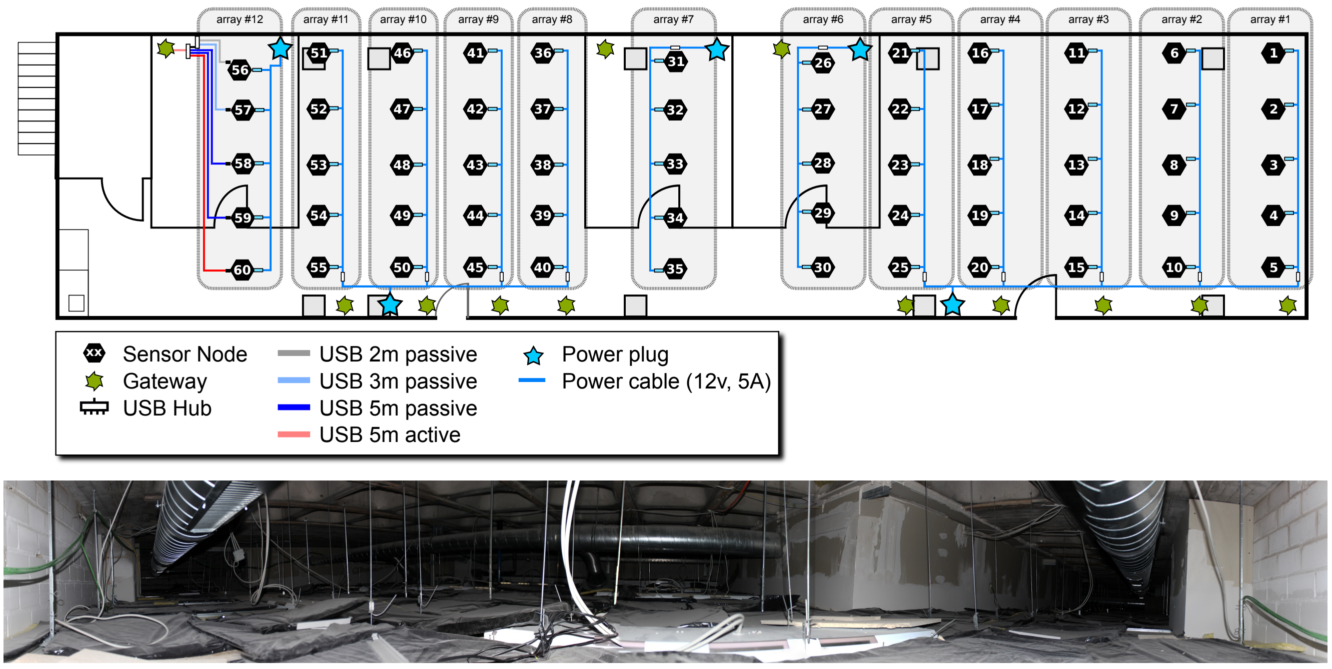


Figure 5.5.: Top: the TUDμNet deployment at the GKmM Laboratory, split into its 12 arrays. The diagram presents the two-level USB backchannel (for clarity, only for array #12), as well as the additional power infrastructure to feed the external CO/CO_2/temperature sensors. Bottom: panoramic view of the space above the dropped ceiling with some of the cabling.

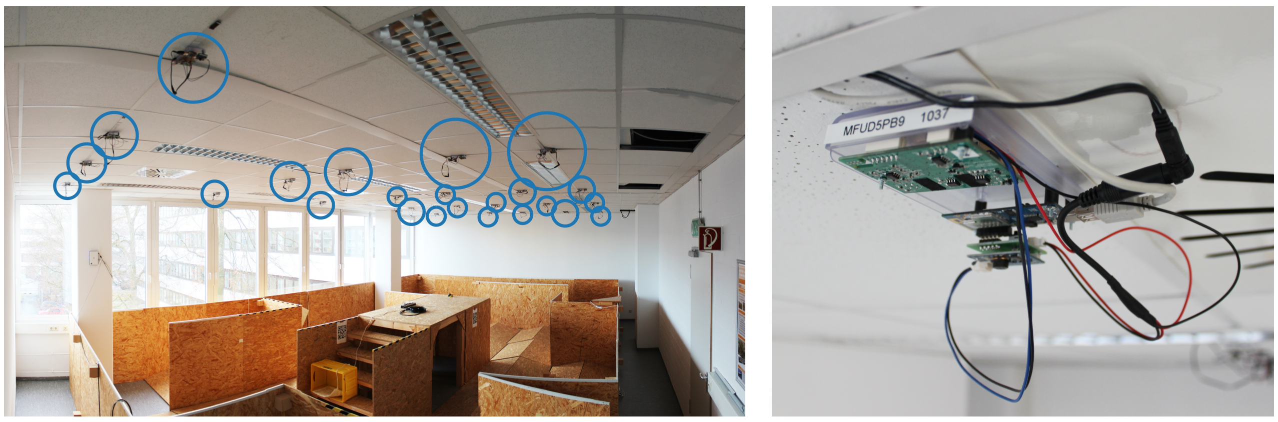


Figure 5.6.: Left: the RoboCup Rescue arena at the GKmM Lab, with 25 nodes on the ceiling (arrays #1 to #5). Right: fully assembled CM5000 (a TelosB clone) with adapter board for the external DS1000 set of CO/CO_2/temperature sensors, attached to the USB and power infrastructure.

water temperature? These questions represent some of the engineering challenges where WSNs offer an unprecedented monitoring resolution and can help to improve their energy consumption.

The third TUDμNet site was deployed at the Architecture Dept.'s solar house (the surPLUShome), an award-winning architectural design that produces surplus energy above what it uses. Together with colleagues from the chair on *Energy Efficient Construction and Design*, the site was designed and organized into three areas: 1) the inner room, 2) the east and south façades, and 3) the cooling ceiling (cf. Fig. 5.7). These areas allowed a deeper investigation of the properties of the individual subsystems.

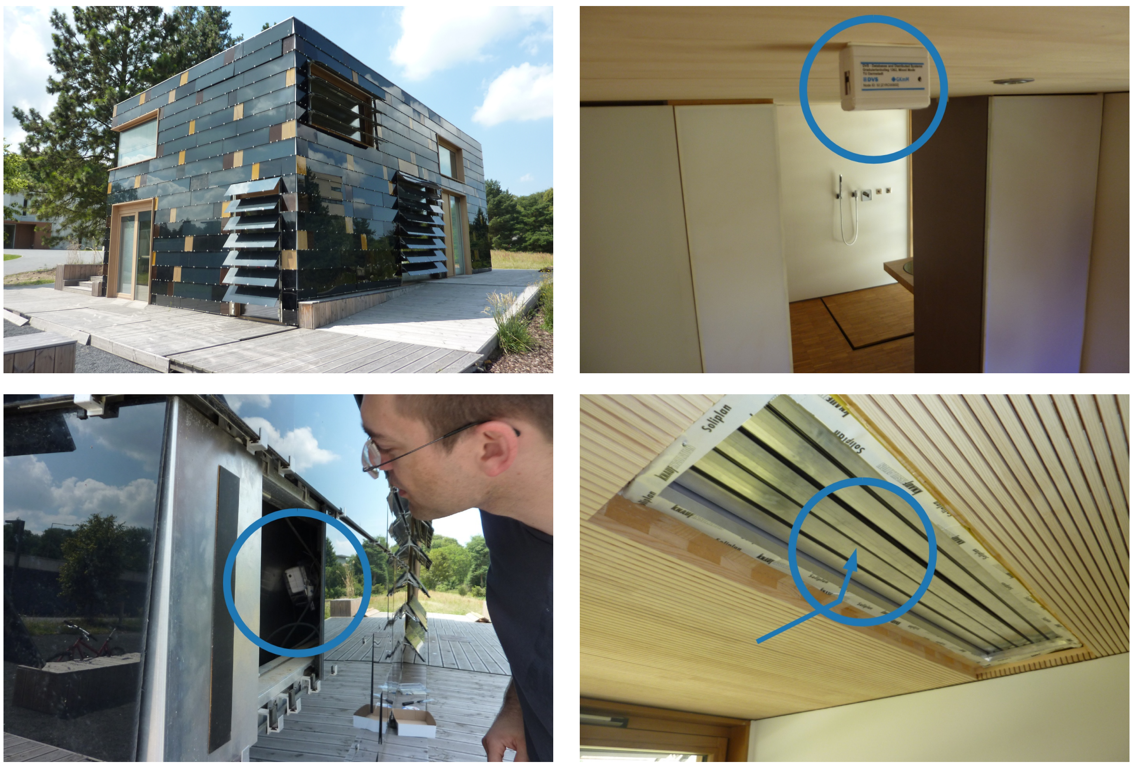


Figure 5.7.: Left to right, top to bottom: surPLUShome and nodes deployed in the inner room, façade and cooling ceiling

Urban Park Management

Parks, squares and other open spaces are ever more important in metropolitan areas due to their effect in reducing environmental pollution, besides encouraging citizens to an active lifestyle and reducing stress through the interaction with nature. Urban park monitoring operations (open spaces, water streams, visitor counts), park irrigation and other maintenance tasks like lawn-mowing, pruning of trees, bushes and plants, and emptying garbage bins, all offer room for optimization.

For experimentation with outdoor sensor network systems, we have chosen the Botanical Garden of the TU Darmstadt, a venue with especially changing weather characteristics (9.25 rainy days per month, and harsh temperature changes spanning between -2°C and 24°C in average). The initial deployment, carried out in the context of the Master Thesis of Iliya Gurov [56], targets the coniferetum, an area rich in types of trees, whose growing properties required further examination. The site includes above- and underground nodes operating at sub 1GHz frequencies, and sensors for monitoring the soil moisture (Fig. 5.8).

Since this outdoor environment is located at a public space, in order to realize our testbed services we had to take into account visitors' safety as well as measures against vandalism. We opted for a wired backbone (instead of a wireless one), which also leads to a more robust experimentation facility, at the cost of running cables through the environment. To protect the cabling from freezing, regulations for the installation of the electrical infrastructure require burying the cables at least 50cm below the surface. The backbone is a hierarchical structure containing a mix of power, fiber-optic, Ethernet and USB cables (Fig. 5.9). The site is split into the north and south sections, and each section has an above-ground main station, where a router merges power and the fiber-optic signals into regular Power-over-Ethernet

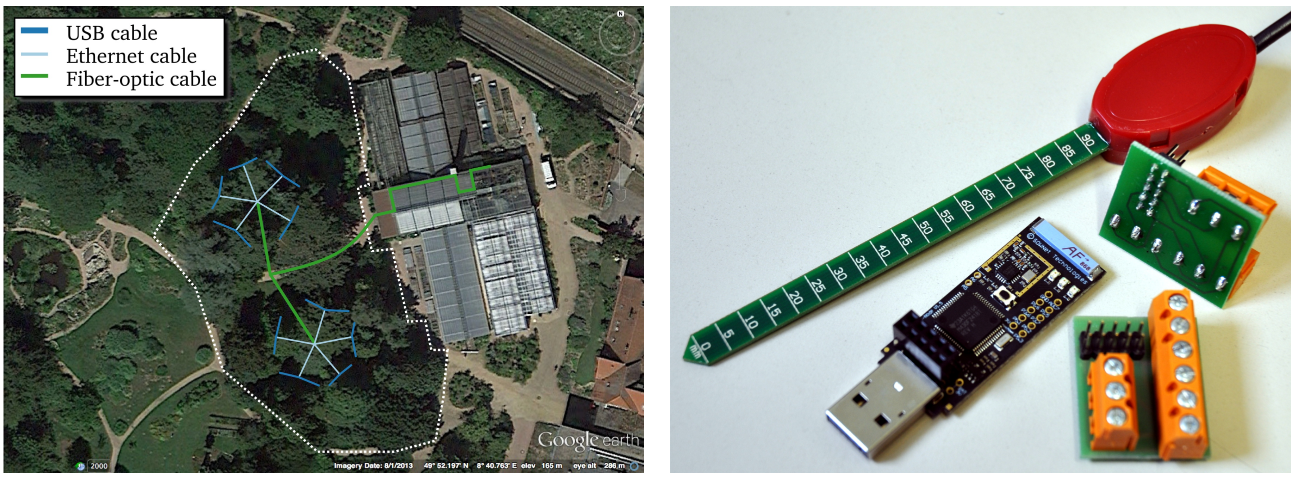


Figure 5.8.: Left: map of the botanical garden, highlighting the coniferetum area and the different types of cables used (imagery ©2014 AeroWest, Map data ©2014 GeoBasis-DE/BKG, ©2009 Google). Right: G-Node with adapter board, and the Vegetronix VH-400 soil moisture sensor probe.

(PoE) cabling. The PoE-ready gateways are connected to two G-Nodes, which ultimately have 2 or 3 soil moisture sensors attached to them.

The selection of sites discussed, summarized in Table 5.2, have a high potential for evaluating applications that target the lowering of greenhouse gas (GHG) emissions, and in particular CO_2 levels. In [51], we describe the activities performed in this direction.

5.3 Implementation

To develop TUDμNet, we have taken MoteLab's core and extended it in a number of ways. Beyond the physical structure of the federated sites, TUDμNet organizes all its sensor nodes by means of node *zones*. At their core, zones are simple subsets of a parent zone (at the top, the universal set contains all nodes). Figure 5.10 depicts the current structure.

The federation enables concurrent jobs (i.e., any two jobs that partially or totally overlap temporally), as long as they are scheduled for different zones. Given that we perform the management of the zone

Table 5.2.: The TUDμNet testbed sites

site	CS Dept.	GKmM Lab	Botanical Garden	surPLUShome
nodes	53 TelosBs, 30 XM1000s	60 TelosBs	2 TelosBs, 22 G-Nodes	20 Z1s
sensors	light, humidity, temperature	light, humidity, temperature, CO CO_2	light, humidity, temperature, soil moisture	light, high precision temperature, humidity, CO_2
focus	networking aspects, sensing, actuation	gas plume detection,	environmental monitoring	environmental monitoring

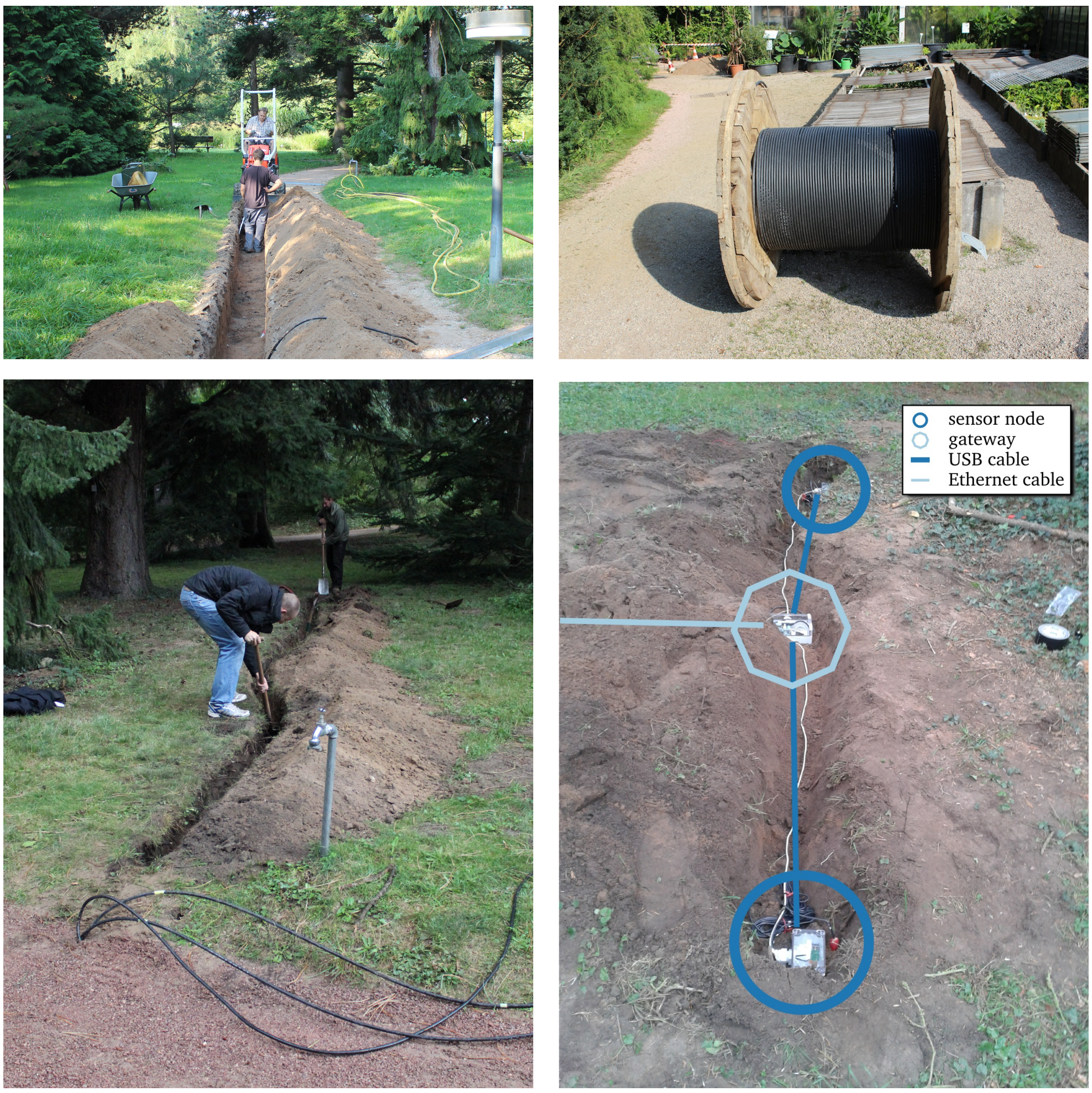


Figure 5.9.: Left to right, top to bottom: 1) digging work on main path for deployment of power and fiber-optic cables 2) at correct depth –minimum of 50cm–, 3) shoveling within the coniferetum to finally deploy 4) nodes and their enclosures

hierarchy centrally, verifying the availability of nodes for a submitted job is straightforward. As with any flat organization, as the system scales up, the benefits of a hierarchy become more evident. An administrative panel enables the reconfiguration of zones.

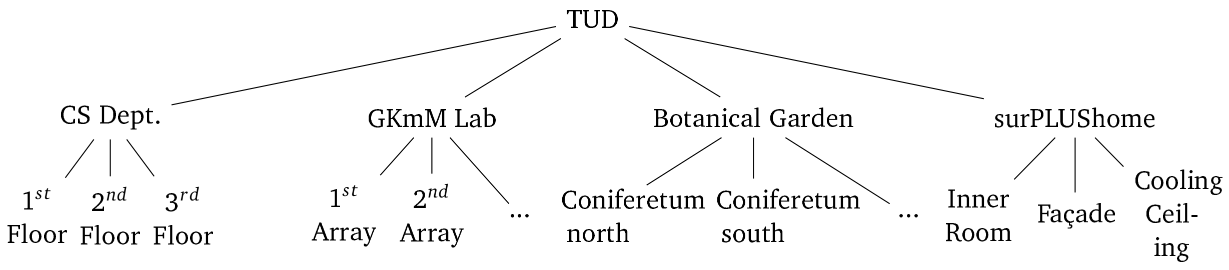


Figure 5.10.: Logical structure

The next sections describe two aspects which have required important attention: Section 5.4 presents the improvement of the backchannel's reliability, and Section 5.5 the redesign of the GUI for specifying and scheduling jobs.

5.4 Challenges of the USB Backchannel

In order to reduce the day-to-day maintenance costs, the design of testbeds of reasonable size aim at an *unattended* operation. Interconnecting the testbed's sensor nodes to a central gateway through a USB infrastructure has become the method of choice because USB can provide power to the nodes, be used to reprogram nodes, and act as data-logging backchannel.

However, the design and installation of this sort of USB infrastructures is often an underestimated task with pitfalls that can cause the testbed to become highly unreliable and costly to maintain. First, current WSN platforms include (both hardware and software) implementations of USB protocols that are not bug-free: although a testbed health monitoring system or a testbed engineer could troubleshoot these issues, frequent manual intervention to restart and reconnect sensor nodes is required on-site. In an unattended setup, this increases the maintenance costs and additionally reduces the testbed nodes' availability.

Secondly, this issue is exacerbated with larger USB topologies, where cabling quickly reaches longer lengths and contains hubs that fan out to many nodes. Despite employing USB topologies and parameters within the USB specification, even high quality off-the-shelf USB components do not play well with these rather extreme setups, exhibiting considerable instability for power and data lines and thus causing nodes to become unreachable. Although the unreliability of the gateway-nodes' USB backchannel is well-known in the WSN testbed community [60], this issue is typically bypassed by manually resetting individual nodes.

5.4.1 USB in Sensor Network Testbed Backchannels

Sensor Nodes

All sensor nodes (except perhaps for final products) require a programming and debugging interface. Early platforms like the Mica2 made use of a specialized programming board [96], to which they attached via a 51-pin connector. This connector would typically wear out after a number of reconnections. JTAG is another interface broadly adopted for device reprogramming and debugging (e.g., the EyesIFX node). To the best of our knowledge, USB was first used to interface the widely available TelosB node [109].

Table 5.3 summarizes various sensor nodes that employ USB to connect to a host computer. Some sensor nodes' microcontrollers, such as the TelosB's MSP430, export UART pins for reprogramming and debugging, thus require a USB converter chipset (which can be either *on-board* or on a *separate* device). Other nodes, such as the jNode [120] have microcontrollers with USB support already *built-in*. In this chapter we discuss the nodes chosen for the TUDμNet federation, which are based on the popular MSP430 microcontroller, with a wide variety of USB chipsets.

USB Topologies, Hubs, and Cabling

A USB topology connects sensor nodes with a gateway. Physically, the USB forms a layered star topology (Fig. 5.11), or tree, with hubs at the center of each star, and the *root hub* typically embedded in the host gateway device. Hubs can be passive (bus-powered) or active (self-powered). Due to timing constraints, up to 7 layers are allowed. Nodes and hubs connect to their parent hub via point-to-point USB cables.

Table 5.3.: USB interfaces of various sensor nodes.

sensor node	USB type	USB chipset / Microcontroller
TelosB	on-board	FTDI FT232BM
Econotag	on-board	FTDI FT2232HL
jCreate	separated	FTDI FT232RQ
Z1	on-board	Silicon Labs CP 2102
iMote2	built-in	Intel PXA271
SunSPOT	built-in	Atmel AT91RM9200
Egs, Opal	built-in	Atmel SAM3U
jNode	built-in	Atmel ATmega32u4

These cables can be passive (limited by power and timing constraints to a length of 5 meters) or active (extend the length to 10 or 12 meters by using signal repeaters and specialized circuitry). By chaining a sequence of up to 5 passive cables and active USB hubs, the distance can be extended to 30 meters and still conform to the USB standard. As we will show, this is the most robust topology, and comes at the expense of extra power lines for each active hub.

Gateways

The literature reports various host platforms being used as gateways. Due to its low-cost hardware, the Linksys NSLU2 was adopted early on as gateway for TelosB nodes (e.g., [142, 60]). Intel Stargates, featuring a broader set of ports (PCMCIA, CompactFlash, I^2C, etc.) and a faster processor, were used in [5]. Similarly powerful, a number of routers with USB ports, like Buffalo's, have been chosen as gateways and customized with slimmed-down Linux distributions such as OpenWRT, e.g. in [52]. All these platforms require some major involvement in setting up the gateway software due to their incompatibility with x86

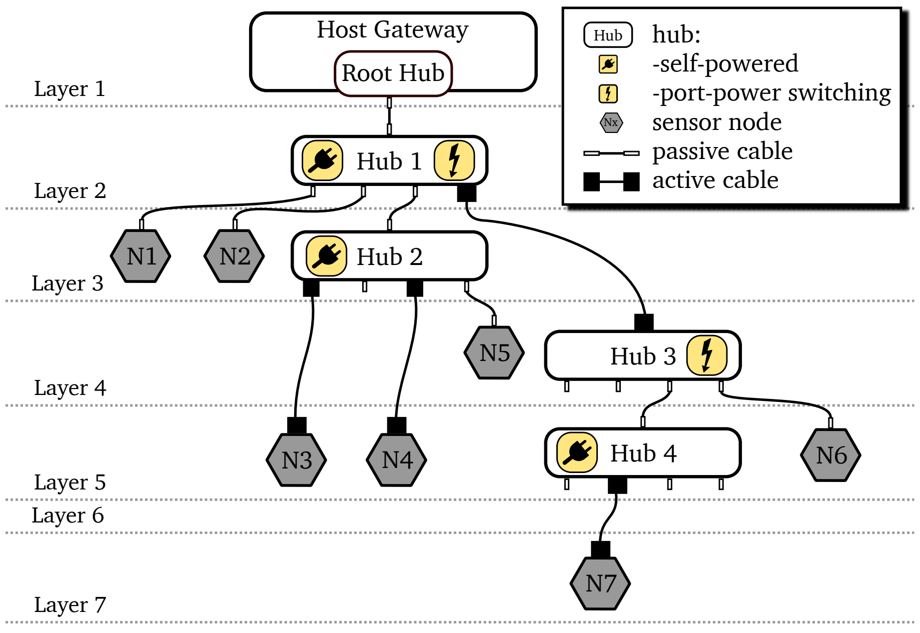


Figure 5.11.: Key components of a USB topology in several layers: root hub, active and passive hubs and cables, and nodes.

Table 5.4.: Common gateways: Linksys NSLU2 (Slug), Buffalo WZR-HP-AG300NH (Buffalo), and PC.

Platform	CPU type, speed	RAM / ROM	USB	Price ($)
Slug	Intel IXP42x, 266 MHz	8MB / 32MB	2x2.0	90
Buffalo	Atheros AR9132, 400MHz	32MB / 64MB	1x2.0	100
PC	Intel Dual Core, 2.5GHz	1GB / 80GB	4x2.0	300

architectures. Finally, some testbeds employ general purpose PCs as gateways as well (e.g., in [27] and [30]). Such added flexibility in the testbed software preparation comes at a higher price per gateway. Table 5.4 summarizes these, ordered by increasing processing power.

Gateway-to-Node Ratio and Scale

In some testbeds, only one sensor node is connected to each gateway, either directly (e.g., Kansei [5]) or with a very short cable (e.g., MoteLab [142]), thus a large number of gateways is needed. By employing USB cabling between gateways and sensor nodes, the number of design options increases considerably: In TWIST [60], USB hubs are used, enabling up to 7 sensor nodes to be connected to each gateway while achieving a similar number of nodes in the testbed. There, also a combination of passive and active USB cables is used to extend the distance between gateways and sensor nodes up to 15 meters. Indriya [30] resorts to *high quality* active USB cables which can be daisy-chained to cover a maximum distance of up to 25 meters. This enables covering longer distances with very few gateways. In SignetLab [27], 48 nodes are connected through a two-level USB hub hierarchy to a single gateway (a PC). We summarize these properties in Table 5.5[1].

The choice of gateway-node ratio is a precarious one: from a cost perspective, more nodes connected per gateway implies a lower setup investment and lower gateway maintenance effort for the duration the testbed will be deployed and active. However, reducing the number of gateways while covering the same area requires more USB cabling, with choices of topology and, as we will see, risk of failures rapidly increasing, as more complex USB infrastructure between gateway and nodes is deployed. Ultimately, up to 127 devices (including nodes and hubs) can be connected to a single USB host according to the USB standard [135], yet gateways will eventually require too much processing power as well as storage

Table 5.5.: Some well known testbeds with gateway-node ratio comparison for their USB backchannels.

Testbed	No. gateways	No. nodes	Ratio gw:nodes	Distance gw.⟷node (m)
Kansei	210	210	1:1	0
MoteLab	90	130	1:{1..2}	0.5
TUDμNet's Piloty site	25	83	1:{2..4}	{1..15}
KanseiGenie	112	432	1:4	0.5
TWIST	90	204	1:{4..7}	<15
NetEye	15	130	1:{6..12}	<10
Indriya	6	127	1:22	<25
SignetLab	1	48	1:48	<15

[1] Published details were not always specific; in this case, the respective authors were consulted and figures adjusted, so table data might be different.

capacity to cope with the management of the attached devices. Furthermore, power variations and timing errors during the transmission are likely to become a major obstacle to increasing the ratio to such a high scale.

5.4.2 USB Backchannel Issues

Testbed USB backchannels are exposed to several types of failures, which is why there is no guarantee that a topology will work reliably, even when adhering to the ranges of the USB specification. Variations in the input power of a sensor node (or its USB converter chip) can cause clock synchronization failures. Software glitches in the USB stack and the bootstrap loader, and increased bit error rates can lead to an inconsistent protocol state. These can render a node non-programmable and non-addressable after the error occurs.

Fig. 5.12 presents a histogram with the percentage of failed nodes in jobs submitted to TUDμNet's Piloty site during the first 10 months of operation. This set only contains jobs addressing more than 10 nodes, with other types of failures omitted (spanning around 200 jobs). This shows that only in 11.35% of the jobs no nodes failed, while in all other jobs at least one node did. It is important to note that the referred USB topologies conformed to the USB standard, and that none of these faults were due to faulty nodes, nor occurred at one specific position within a topology. *These node failures occurred in patterns that are hard to track down or reproduce.*

Such nodes resemble the so-called fail-stop behavior[2]: nodes stop being reprogrammable, which is identifiable through an error message, and remain in this state until being serviced. Two types of solutions exist: a) a purist approach, and b) a method to emulate reconnecting the node to the USB port. The first approach requires debugging and tracing the entire hardware and software stack, including USB drivers, the OS kernel, and the serial bootstrap loader. This proved to be hard in the target environment. The second approach emulates reconnections by removing power from the node and its USB converter, and later reenabling it. This is much simpler than the former approach.

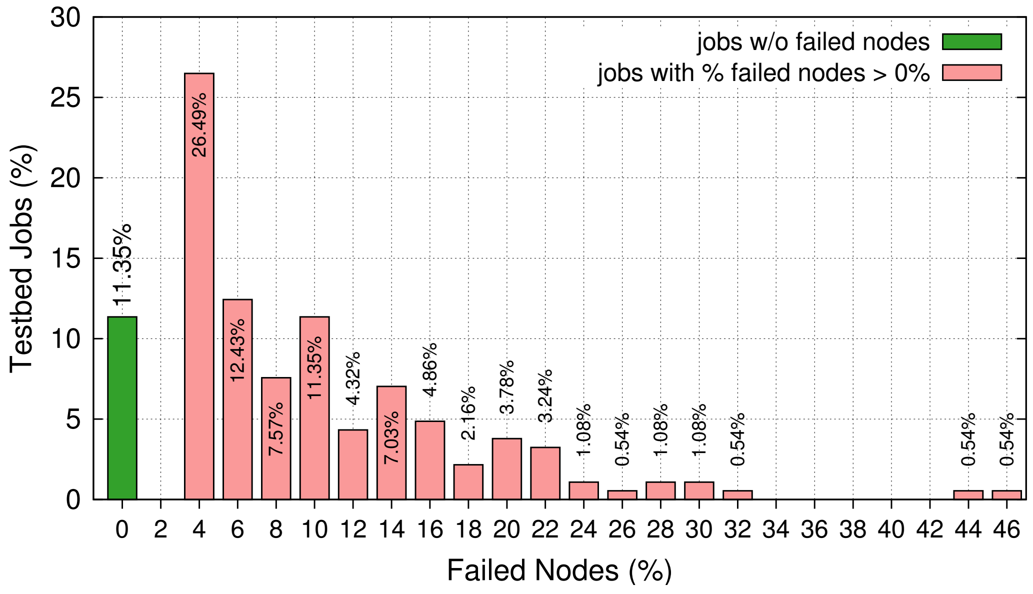


Figure 5.12.: Histogram characterizing USB reprogramming failures in the early Piloty site over 10 months: 11.35% of the testbed jobs deployed to all nodes; for the others, between 4% and 46% of nodes were not reprogrammed.

[2] This is in contrast to fail-silent failures, where the device would provide no hints that it has become non-programmable.

Without servicing failed nodes, the number of available (i.e., reprogrammable) nodes in the testbed drops *monotonically*. In a permanent and unattended testbed, non-programmable nodes that require manual intervention imply bad experiment repeatability. Next, we present a study of various USB backchannel configurations and characterize the failures for different topologies with the aim of providing insights leading to reliable backchannel designs.

5.4.3 Backchannel Evaluation

In order to evaluate the different backchannel scenarios, we proceeded with the following basic set of steps: a) physical preparation of the topology (connection of hubs, cables and/or nodes), b) verification of power on all nodes, c) verification of correct, stable node enumeration (i.e., registration) at the host gateway, and d) execution of microbenchmark. The microbenchmark software has minimal impact on the measured results, since it simply consists in repeatedly reprogramming the node(s), which was done with the default bootstrap loaders provided by Contiki and TinyOS for the respective sensor node and host platform combination. The implementation was written in Perl and exploits parallel processes for reprogramming the nodes as necessary. This approach allowed us to identify topology-related (i.e., spatial) issues. Although tests lasted from several hours to a couple of days, we could only capture a fraction of the temporal issues that can emerge on a long-term, permanent testbed as presented before.

Across our tests we resorted to binary images of three sizes. A `hello world` program represented our smallest image. As mid-size image we used a Scopes [71] test application. Finally, we use the binary image from ukuFlow as our largest test program. The program sizes for two popular platforms, the TelosB and Z1 WSN nodes, are summarized in Fig. 5.13a.

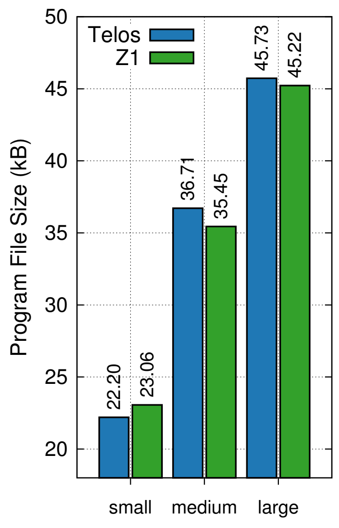
Gateway to Single Node Tests

We evaluated more than 50 cases with single-node topologies in total. For this purpose we elaborated a simple microbenchmark which consisted in sequentially reprogramming the tested node. This sequence was repeated until the node failed, or reached 1,000 iterations. In case of a failure, the procedure was restarted, either by automatically rebooting the gateway when the device's root hub turns off attached devices, or otherwise by manually reconnecting the node. This procedure was repeated 25 times to ensure statistical validity.

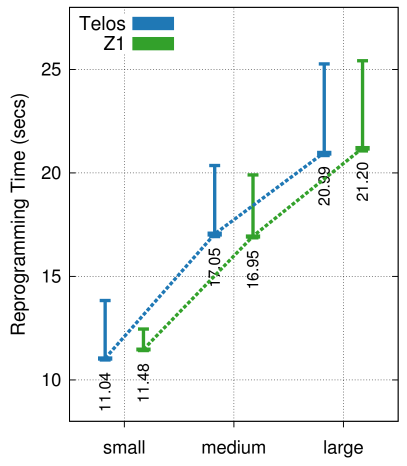
Figure 5.13b presents the reprogramming time for both TelosB and Z1 nodes with the three file sizes described earlier, when nodes were connected directly to a gateway. As expected, the larger the file size, the longer the average reprogramming cycle was (averages are connected by dotted lines). The error bars show an outlier, which is normally the first iteration, where the bootstrap loader and the program image files must be loaded into memory. Interestingly, no major differences were noticeable between the two node types, although they used a different USB converter chipset (FTDI versus SiLabs). A closer look at different manufacturers of these nodes revealed similar reprogramming times (cf. Fig. 5.13c), except for Moteiv's Tmote Sky nodes, which took longer and had a higher variability. We believe this could be due to these nodes belonging to some of the very early manufactured revisions. In terms of reliability, no differences could be observed across these sensor nodes.

Distance between Gateway and Node

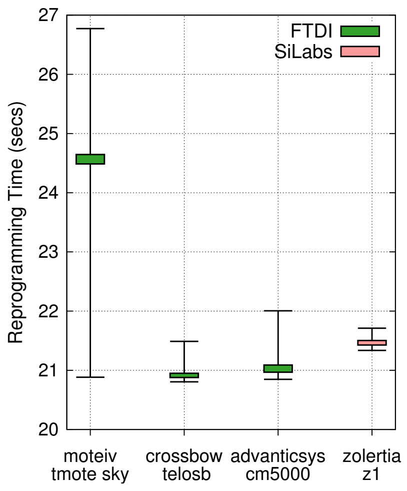
For many environments, long connections between a gateway and a node are advantageous. Since the USB standard dictates a maximum cable length of 5 meters, we evaluated various topologies with different cables and hubs. In Fig. 5.14a and 5.14b we present both the reprogramming time performance



(a) program file size

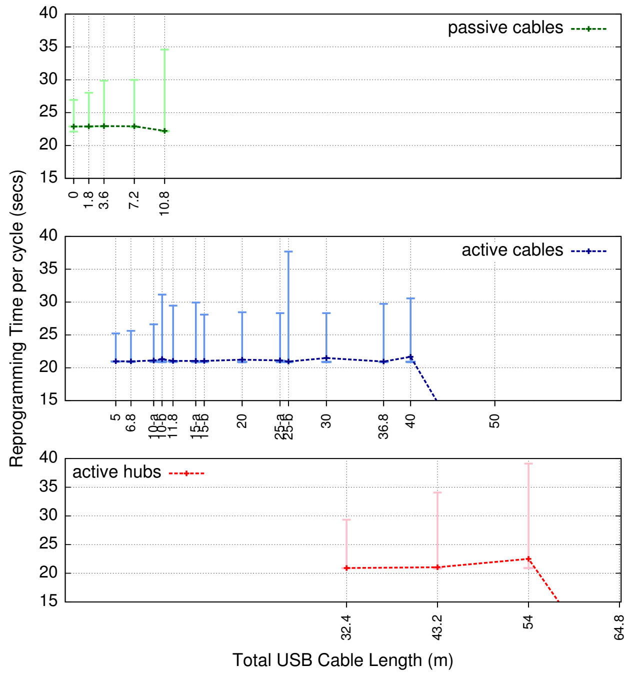


(b) reprogramming time

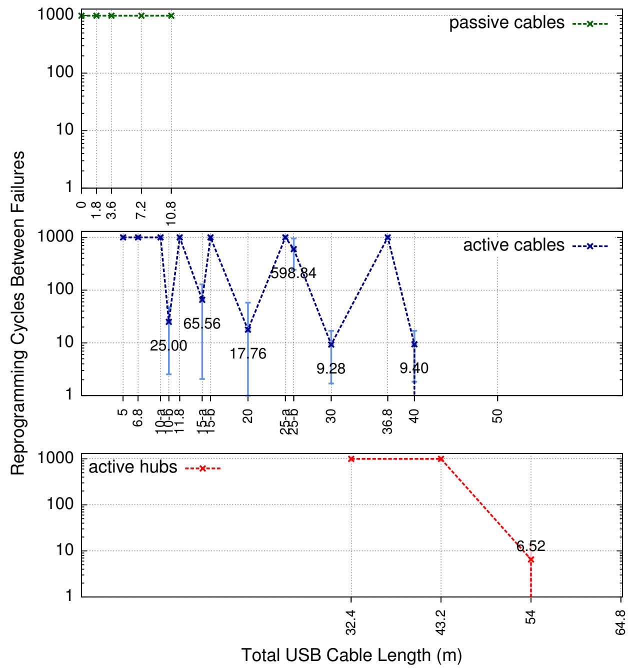


(c) reprogramming time for various MSP430 platforms

Figure 5.13.: The effect of the node manufacturer on reprogramming time for a single-node: with differently-sized programs, several platforms can be seen to have significant differences in programming time.



(a) Reprogramming time vs. total cable length.



(b) reprogramming cycles between failures (RCBF) vs. total cable length.

Figure 5.14.: Single-node reprogramming tests

as well as the reprogramming cycles between failures (RCBF), respectively. The components used in these topologies, their order, and the resulting number of USB layers, are listed in Table 5.6.

By chaining standard 1.8 meter *passive cables*, it was possible to power a node located up to 10.8 meters from the root hub. Though this is surprisingly well beyond the USB specification, nodes were correctly enumerated and worked reliably. As expected, the reprogramming time performance variance grew with

Table 5.6.: Some of the single-node topologies tested to reach a certain length between gateway and node. (p.c. = passive cable; a.c. = active cable; a.h. = active hub)

length (m)	components	USB layers
0.0	direct	2
1.8	1 x 1.8m p.c.	2
3.6	2 x 1.8m p.c.	2
7.2	4 x 1.8m p.c.	2
10.8	6 x 1.8m p.c.	2
5.0	1 x 5m a.c.	3
6.8	1 x 5m a.c. + 1 x 1.8m p.c.	3
10-a	2 x 5m a.c.	4
10-b	1 x 10m a.c.	3
11.8	1 x 10m a.c. + 1 x 1.8m p.c.	3
15-a	1 x 5m a.c. + 1 x 10m a.c.	4
15-b	1 x 10m a.c. + 1 x 5m a.c.	4
20	2 x 10m a.c.	4
25-a	2 x 10m a.c. + 1 x 5m a.c.	5
25-b	5 x 5m a.c.	7
30	3 x 10m a.c.	5
36.8	3 x 10m a.c. + 1 x 5m a.c. + + 1 x 1.8m p.c.	6
40	4 x 10m a.c.	6
50	5 x 10m a.c.	7
32.4	3 x 6 x 1.8m p.c., 5 x a.h.	7
43.2	4 x 6 x 1.8m p.c., 5 x a.h.	7
54.0	5 x 6 x 1.8m p.c., 5 x a.h.	7
64.8	6 x 6 x 1.8m p.c., 5 x a.h.	7

the total length. All nodes used in these topologies were correctly recognized and reprogrammed for the full 1,000 iterations. From 12.6 meters onwards, nodes were not enumerated anymore.

Conclusion 1:
Standard passive cables will work to cross a distance from gateway to a node of up to 10 meters.

When resorting to *active cables*, the 10 meter limit was overcome using various 5 and 10-meter cables of this kind. Since these work internally as a hub, technically up to 5 of these can be chained, thus potentially reaching 50 meters with active cables plus a last passive segment of 10.8 meters. With these components, however, correct enumeration was found to be limited to a maximum of 50 meters. Average reprogramming time and variance grew with cable length, with the reliability decreasing considerably. With three 10-meter cables, for instance, we observed an average of 9.28 RCBF. Remarkable was also that having a 10-meter cable as the last segment always led to poor reliability.

Conclusion 2:
Active cables extend the distance to the gateway, at the cost of decreasing reliability, to 40 meters.

By employing *active hubs* and passive cables, the length was stretched further. Inter-hub lengths of 5.4, 7.2, 9.0 and 10.8 meters were tried, for a total of 32.4, 43.2, 54 and 64.8 meters, respectively. The longest length achieved with a reliable behavior was 43.2 meters. At 64.8 meters, nodes could be correctly powered and enumerated, but not reprogrammed. All other topologies were either extremely unreliable, or nodes were enumerated but could not be reprogrammed.

Conclusion 3:
Active hubs allow extending the distance to the gateway, at the cost of routing power to the hubs, to 43 meters.

Gateway to Multiple Node Tests

When connecting multiple nodes to a gateway, reliability can be expected to drop as the USB backchannel's topology becomes more complex. This was already noticed in steps b (verifying power) and c (enumeration) of the evaluation methodology. When having more than 64 nodes, and thus more than 3 layers of 4-port USB hubs, enumeration became very unreliable. This was due to sections of the tree not being powered in a stable manner. We believe that this is an issue in the USB handshake protocols. We managed to power 64 nodes and have them registered with the OS, though this required some effort since at this scale, the topology became very sensitive to cable quality. Figure 5.15 presents the multi-node topologies that worked reliably (summarized in Table 5.7).

The microbenchmark for multiple-node topologies was parameterized to support several concurrent processes. Fig. 5.16 exemplifies two instances of its execution. In Fig. 5.16a, the number of nodes equals the number of processes (n=p=4). At the third inner iteration, nodes 2 and 4 fail to be reprogrammed; the others continue. Once all nodes fail, the system is reinitialized and the whole procedure is repeated (25 times). In Fig. 5.16b there are more nodes than processes (n=8, p=3), therefore, at least three rounds are necessary in each inner iteration.

Effects of Host Gateway

The selection of the gateway platform plays a major role in the overall testbed costs. We compared the reprogramming performance of the three host platforms of Table 5.4. The left plot in Fig. 5.17 shows that faster gateways also exhibited faster average reprogramming cycles, which suggests that the more nodes a topology has, the better suited a more powerful gateway is. This assumes that the topology is reasonably designed: the scenario with only the 10-meter cable shows that the reliability decreases with more powerful gateways (Fig. 5.17, right barchart).

Conclusion 4:
The choice of gateway platform should be determined by the speed at which all its nodes need to be programmed.

Table 5.7.: Multi-node grid topologies in detail.

grid	nodes	area (m^2)	density (n/m^2)	USB layers	USB hubs
a) 3x3	9	12.96	0.69	5	3
b) 5x3	15	25.92	0.57	7	5
c) 4x6	24	48.60	0.49	6	8
d) 6x6	36	103.68	0.34	5	21
e) 7x7	49	147.91	0.33	5	21
f) 8x8	64	147.91	0.43	5	21

a)

b)

c)

d)

e)

f)

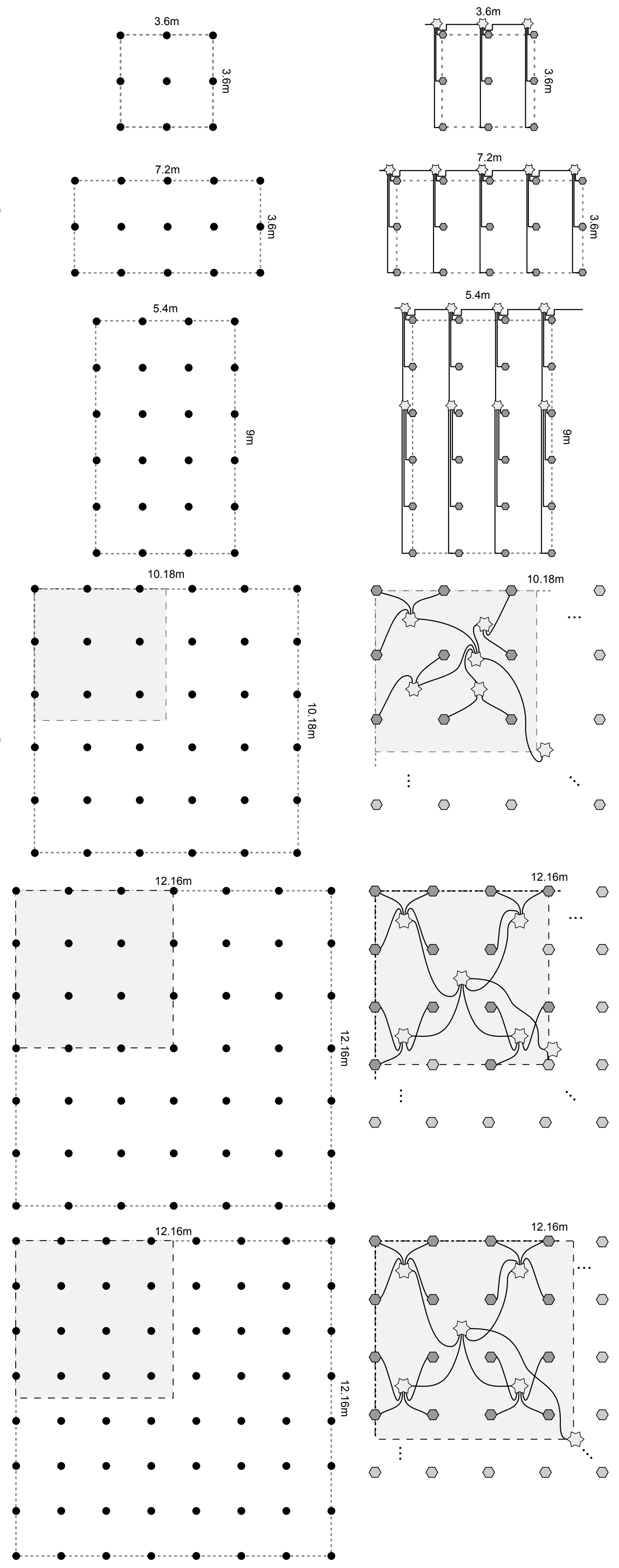


Figure 5.15.: Multi-node grid deployments (left) and underlying USB topologies (details in Table 5.7).

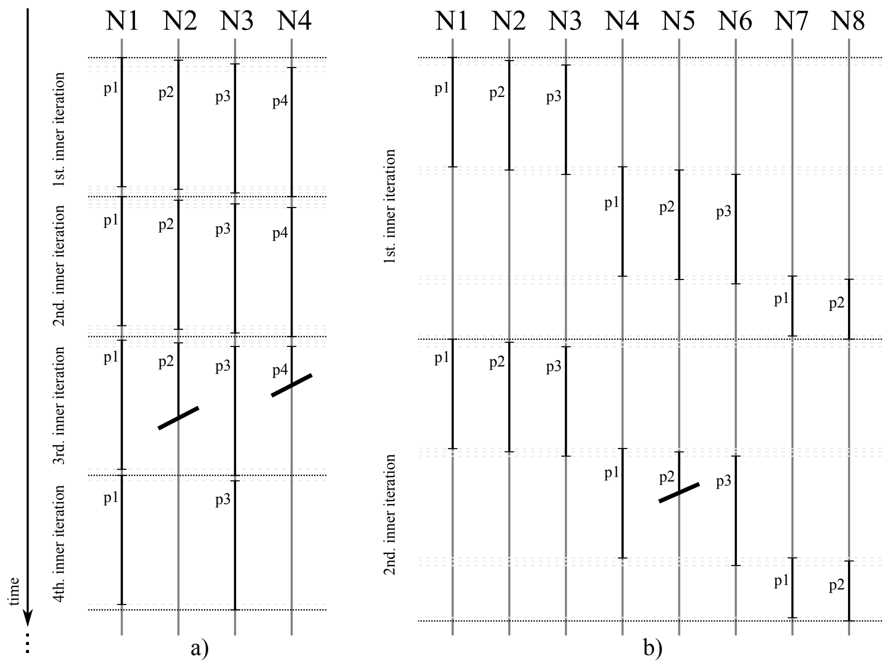


Figure 5.16.: Two instances of the microbenchmark to test reprogramming behavior. In a) there is the same number of processes and nodes (n=p=4), thus each process participates only once in each iteration. In b) there are more nodes than processes (n=8 and p=3), thus processes participate multiple times in each iteration.

Sequential versus Parallel Reprogramming

A topic that arises when reprogramming multiple nodes from a single gateway is that this should ideally be done in parallel since this might save time, compared to a sequential reprogramming. The limited resources of single-board computer gateways, however, constrain the degree of parallelism. The table in Fig. 5.18a presents our findings on the maximum degree of parallelism (row called *max* ||°) of each gateway; reprogramming more nodes caused the host platform to hang. (Note that a PC could probably reprogram more than 59 nodes, but this was a limit in our test topologies due to power and enumeration.) The bottom part of the table indicates in how many rounds a topology of a given number of nodes can be divided in order to exploit parallelism. Evidently, the slug will require many rounds to reprogram large topologies, while a PC could do it in one or two rounds. Fig. 5.18b presents the observed average time it took the Buffalo gateway to reprogram once all of the nodes in each of the topologies of Fig. 5.15, both with the maximum degree of parallelism (bottom) and sequentially (top). The diverging curves show that parallelism should be preferred. From the reliability perspective, it was not relevant how nodes were reprogrammed.

Conclusion 5:

Topologies with many nodes should exploit parallelism to reduce the reprogramming overhead time.

Effects of USB Hubs

In these experiments, we have inspected a total of ten USB hubs, with varying configurations of number of ports (4- and 7-port) and power supply (bus- and self-powered). In our experiments, no noticeable effects were obtained, neither in terms of reprogramming time, nor in reliability. The next section indicates, however, which hubs are to be preferred for a testbed.

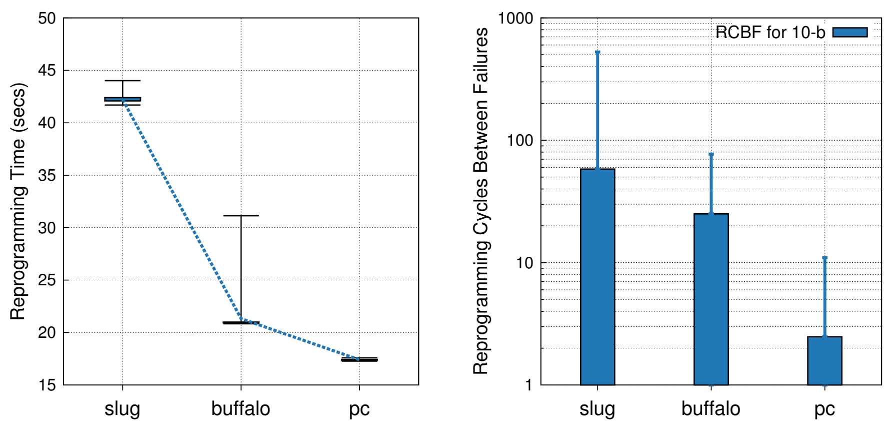


Figure 5.17.: Comparison of gateway platforms of Table 5.4.

USB Power Control to Enhance Reliability

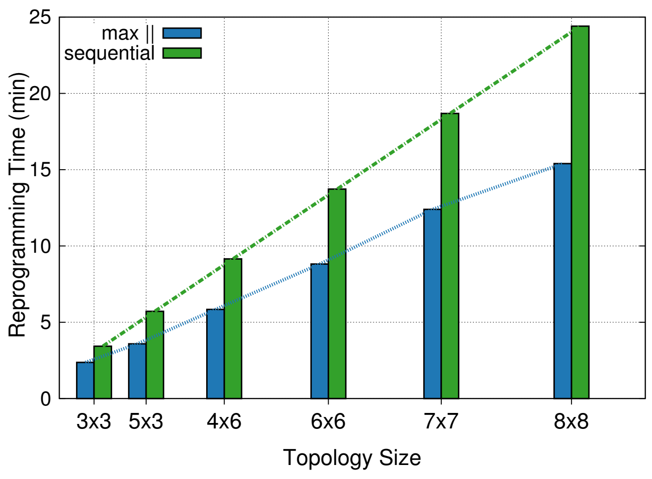
Manually reconnecting nodes to the USB cabling, in order to remove power temporarily and cause a hard reboot effect, is a costly solution to the backchannel problems. By using a feature of USB 2.0 hubs, namely hub port power control (HPPC) [135], it is possible to achieve the same effect, but without requiring manual intervention. Power control was first used in TWIST [60] to emulate node deaths. Here we resort to it in order to increase the reliability of the testbed.

The power control procedure begins by constructing a tree reflecting the attached USB topology (as for example the one shown in Fig. 5.11). This is done by exploring operating system's data structures. For each element in the tree, the OS provides metadata such as whether the element is a sensor node or a hub (and whether it is active or passive), its manufacturer, the product ID and other descriptors. Details matter, since many hubs that shared the vendor and product IDs were very different internally.

Once a gateway has constructed its USB tree, HPPC can be applied to all nodes or a selected one. Switching power of all nodes can be used, e.g., to do a testbed *soft reboot* of the lowest, sensor node

		Slug	Buffalo	PC
max ‖°		5	8	59
rounds for # nodes	4	1	1	1
	12	3	2	1
	24	5	3	1
	48	10	6	1
	64	13	8	1|2

(a) parallel capacity



(b) total reprogramming time

Figure 5.18.: Exploiting parallelism for reprogramming nodes through a backchannel. The table on the left shows the maximum capacity observed for the tested gateways. The plot on the right compares the time needed to reprogram all nodes in sequential vs. parallel fashion.

tier. For this case, the procedure starts traversing the tree from the root hub and, in a post-order fashion, switching power (on or off, as requested) of all of a hub's ports, assuming it supports HPPC. Note that it does not suffice to stop at the first USB hub that supports HPPC, since downstream hubs could be self-powered (thus connected nodes would remain unaffected).

Switching power of a particular node (without affecting the others) is used if for instance a test job is running on some nodes within the tree, and some other need to be restarted or reprogrammed. After searching for the target node in the tree, the procedure checks whether its direct parent hub supports HPPC. If it does, power switching is requested for the specific port of that hub to which the node is attached (this is the case for node N1 in Fig. 5.11). If it does not, the procedure backtracks through parents until it finds one that does. Since this could imply switching power to nodes in the common branch, care must be taken to consider this undesired side effect (e.g., N7 is safe through hub 3, but N4 is not safe through hub 1). We have additionally implemented a *force* option, which aggressively ignores switching other nodes' power. This is useful for troubleshooting tasks.

Reliability Improvements

Applying HPPC has shown to overcome many (though not all) of the issues in our testbed. Fig. 5.19 presents a quantification of the effects of HPPC on three of the problematic single node topologies (20, 30 and 54 meters), extended with an HPPC-able USB hub connected to the root. The plot shows the relative improvement in RCBF in these three cases, and suggests that the more complex the USB topology, the higher the improvement achieved by applying HPPC before reprogramming a node.

Enabling HPPC in a USB topology for any given node works best when having USB hubs that support this function as its direct parent. Although the USB standard specifies that bus-powered hubs are required to implement this function (self-powered hubs *might*) [135], in practice, few of all the USB hubs we tried were manufactured with the necessary circuitry to support it, to the point that it is very hard to find any on the market. Mass production, however, does not imply that this circuitry adds a significant price to the USB hub[3], and the reliability gains greatly outweigh its cost.

Conclusion 6:
Using HPPC is an inexpensive mechanism to improve reliability for an unattended testbed operation.

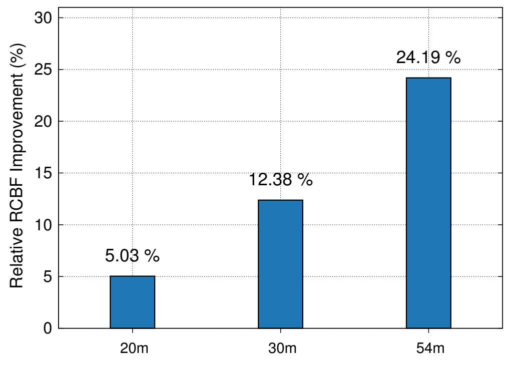


Figure 5.19.: Effects of hub port power control (HPPC) on the reliability within the testbed.

[3] One such hub is the DeLock 4-port 87445, which retails for around $7.

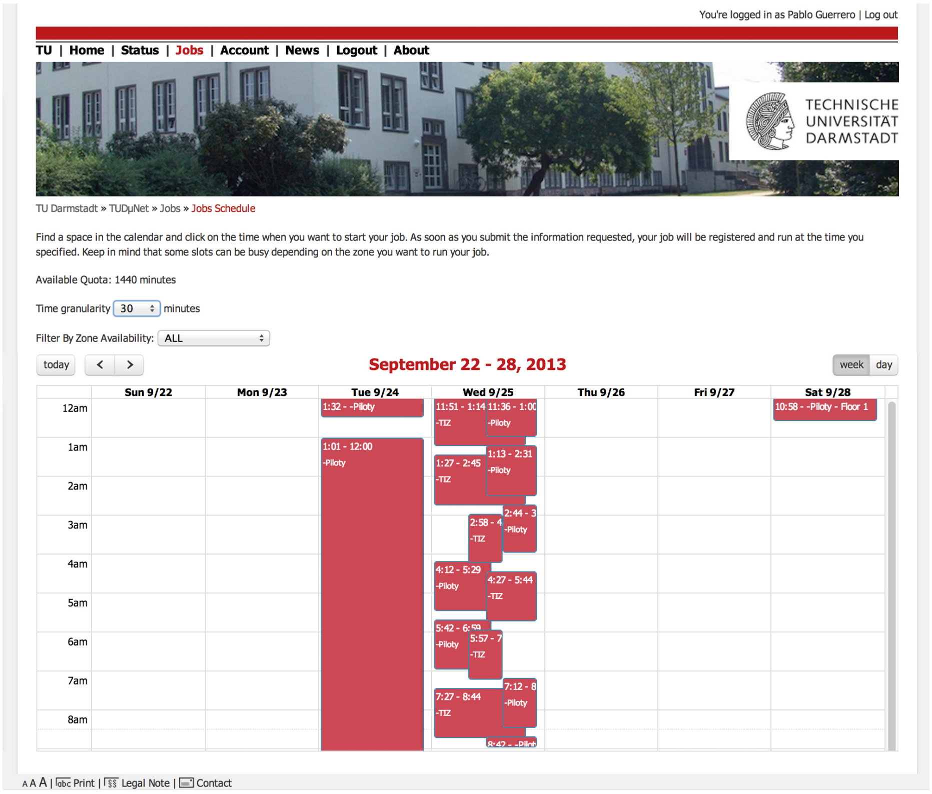
Alternatives

As indicated earlier, some gateways' root hub USB circuitry effectively powers down attached USB devices when doing a software reboot. Besides implying shutting down critical services running on the gateway, boot cycle time (especially of single board computers) can take minutes, reducing the testbed availability. Although we have not searched extensively, among the hardware we inspected only the Buffalo WZR-HP-G300NH (rev. 2.0) routers exhibited USB power-down during software reboot.

5.5 Job Definition and Scheduling

In order to increase the repeatability of experiments, we have not only improved the reliability of the USB backchannel, but also reworked and improved the graphical user interface of the testbed federation by simplifying the definition and scheduling of jobs. TUDμNet enables registered users to log into the web interface, upload binary images, schedule jobs at a desired zone, and retrieve a job's data, all by means of a series of mouse clicks. The interface for job scheduling (cf. Fig. 5.20) received particular attention since, as soon as the system is used by multiple users, it becomes necessary to obtain an overview of empty and reserved time slots.

In TUDμNet, users schedule jobs to run on a zone (or subzone) during a specific time window (e.g., from 3pm to 4pm). Like in MoteLab, users have a *quota* that limits the duration of their *pending* jobs. As an example, a user with a quota of 60 minutes can schedule one job of 60 minutes, or 3 jobs of 20 minutes each, etc. Only after jobs are finished (or cancelled) it is that the user's available quota is updated. Each user is assigned an initial quota of 30 minutes, and can be increased by contacting the system administrators.



TU Darmstadt » TUDμNet » Jobs » Jobs Schedule

Find a space in the calendar and click on the time when you want to start your job. As soon as you submit the information requested, your job will be registered and run at the time you specified. Keep in mind that some slots can be busy depending on the zone you want to run your job.

Available Quota: 1440 minutes

Time granularity 30 minutes

Filter By Zone Availability: ALL

September 22 - 28, 2013

Figure 5.20.: TUDμNet web interface for job scheduling. Concurrent jobs on different zones are shown next to each other. Users can stop jobs or access logged data through few mouse clicks.

In addition, we have implemented a number of features like *public/private* jobs (e.g. for sharing common jobs like data collection, setting node ids, etc.); a simple visualization of the federation zones' status as an overlay on a map; and administrative scripts for testbed maintenance, among others.

5.6 TUDμNet's Impact on Users and the Research Community

Besides technical achievements in the development of the testbed federation, it is important to highlight the effects of its utilization so far. Currently, over 100 user accounts are registered in the system. While most of these accounts have been created for TU Darmstadt's undergraduate students, there are around 10 *power* users, i.e., users who have executed more than 100 jobs in the context of Bachelor, Master and PhD theses. Also, first external users (e.g., from the US, Brazil and Australia) have gotten accounts.

Since the roll out of the web interface, in July 2011, users have scheduled experiments with the testbed amounting collectively to around 8,000 hours, with an ever increasing utilization, as presented in Fig. 5.21. In general, there is a correlation between the number of jobs in a week and the number of hours used in that week. An exception to this observation is found in some weeks where it was allowed to a subset of the users to run experiments for 5 or 7 days straight. More interestingly is that this utilization comes from more than 10,000 jobs. Considering that the manual reprogramming of a 30-node network costs a human about 30 minutes (ignoring the need to physically travel to the site), we can conservatively state that TUDμNet has saved researchers around 5,000 hours of reprogramming time.

5.7 Summary

There exist several ways to carry out system experimentation. We believe that in order to take sensor networks (and in particular systems like ukuFlow) one step closer to their promised ubiquity, empirical experimentation is necessary, and testbeds are an ideal research tool for this purpose.

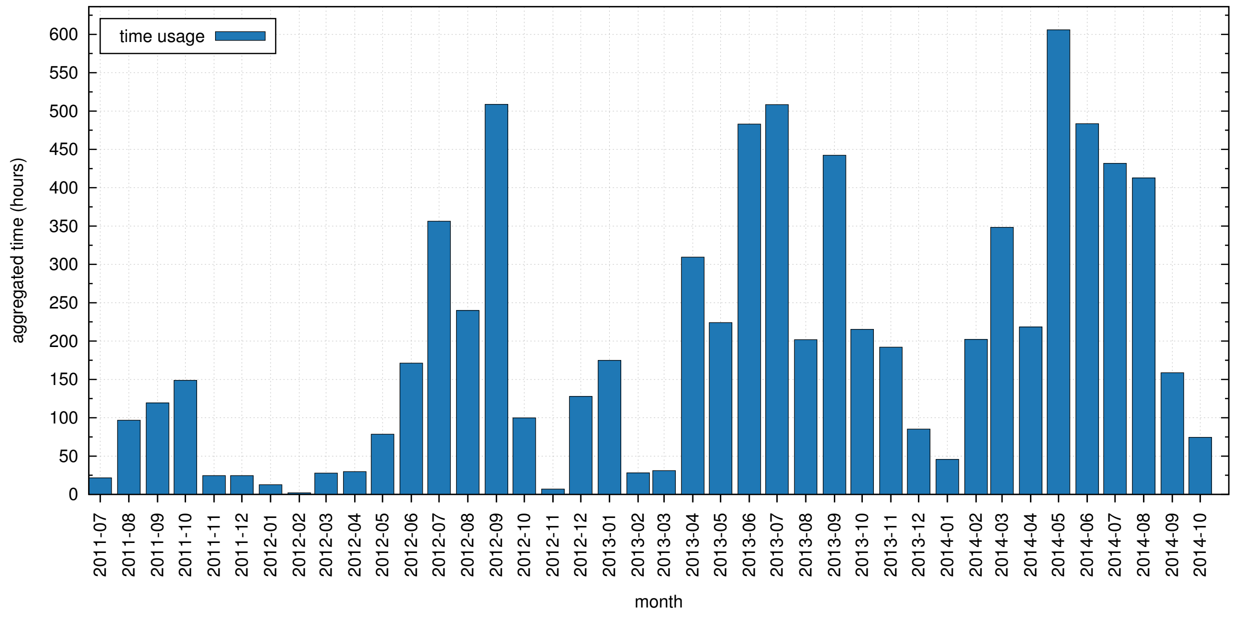


Figure 5.21.: TUDμNet's testbed usage since its early deployment mid-2011. The bars represent the aggregated time spent each month across all sites.

In this chapter we have described the TUDμNet testbed federation, which is composed of four experimentation sites (three of which are currently active), managed through the concept of zones. We have shown that the construction of a testbed site, if not done carefully, can yield high deployment, maintenance and operation costs, and provide a poor experimentation experience with regards to realism and repeatability. The work presented in this chapter to improve the USB backchannel, and the measures taken to improve the user interface, were fundamental in the development of TUDμNet to reach the level of scientific rigour required in research.

The usefulness of this research tool has shown its first fruits, not only for the experts of the domains in which the sites were deployed (TU Darmstadt's Dept. of Architecture and Biology), but primarily for the sensor network researchers using the testbed, as evidenced by its continuously increasing system utilization. In the next chapter, we too exploit the testbed sites to understand and evaluate the ukuFlow system in action.

6 System Evaluation

Statistics: The only science that enables different experts using the same figures to draw different conclusions.

Evan Esar (1899-1995)

This work argues that through a high-level workflow macroprogramming approach, domain experts can define the logic of a sensor/actuator network application, and have this logic run entirely in-network, with a low-power operation, tolerating network dynamics. In this chapter we study the feasibility of such a system by evaluating the most significant aspects that enable prototypical applications to run on real-world WSANs.

The evaluation is structured in two main parts, as presented in Fig. 6.1. In the first part we analyze *micro* aspects of the ukuFlow engine, addressing questions such as whether the system exhibits an adequate footprint for the targeted platforms (Section 6.1), evaluating data management aspects (Section 6.2), and workflow instantiation and execution (Sections 6.3 and 6.4). The second part zooms out from the internals of a single node, and considers *macro* aspects of workflow execution that concern an entire network. In order to correctly assess these results, we carried out a characterization of the test sites before investigating the scoping performance. The chapter is concluded with a study on a number of actuation (Section 6.7) as well as event detection scenarios (Section 6.8 and 6.9).

6.1 System Footprint

We begin discussing the feasibility of ukuFlow for the targeted mote-class devices. All mote-class sensor node platforms are *very* memory constrained. Furthermore, while many sensor nodes such as the TelosB have MCUs with a *single* address space for both data and code, these two are internally mapped to

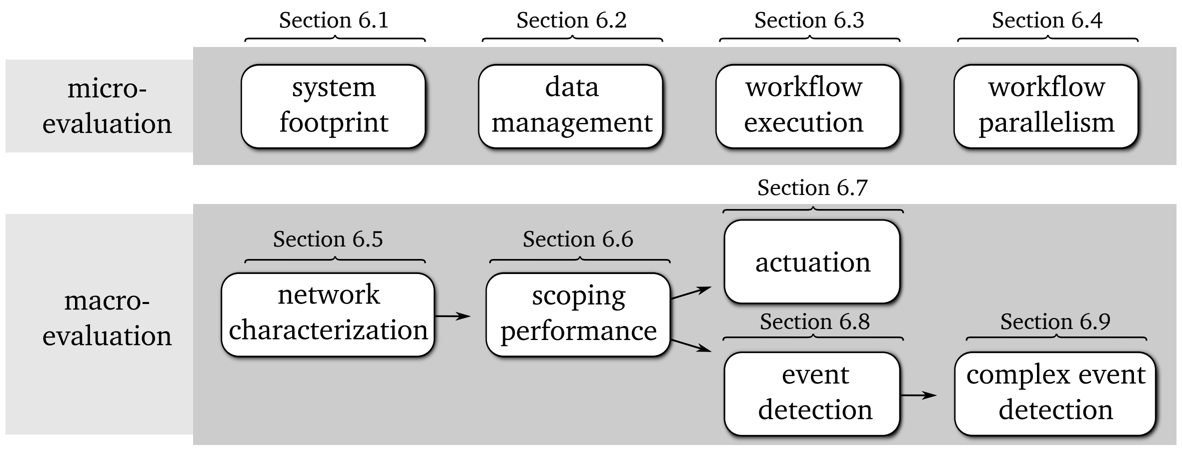


Figure 6.1.: Organization of this chapter

Table 6.1.: Mote-class devices' memory properties

Platform	MSP430 Microcontroller	Memory (KB) RAM	Flash
TelosB / Tmote Sky	F1611	10	48
Zolertia Z1	F2617	8	92
Advanticsys XM1000, SOWNet G-Node	F2618	8	116

different units: RAM (used for data) and flash (for the program space). Table 6.1 details the sizes of various platforms[1]. The system footprint thus needs to be measured from these two perspectives.

The asymmetry between RAM and flash requires a careful definition of the data structures. Therefore, most parameters in the ukuFlow stack are fixed at compile-time so that these can be allocated to the program space. A crucial aspect to decide what to place where is *volatility*, so that the node can eventually recover and continue its operation after a crash or a watchdog reset.

We measured the system footprint by generating the object binaries for the different ukuFlow modules. For this purpose, we use the compiler `msp-gcc` version 4.7.0, and the tool `msp430-size` to extract the size of the binaries (for the flash unit) and the variables (both initialized, `data`, and uninitialized, `BSS`).

Applying some simple techniques, the **entire** system is compiled into a firmware file of less than 45 KB of flash and 6 KB of RAM. This makes the current implementation of ukuFlow suitable even for the resource-constrained sensor nodes. Figure 6.2 presents the requirements for both flash (solid boxes) and RAM (patterned) of the three subsystems (the data manager, the Scopes framework and the ukuFlow runtime). ukuFlow modules themselves require around 18.6 KB of flash, while the remaining 26.4 KB belong to the operating system and libraries. The largest modules are ukuFlow's workflow engine and event manager, and to a smaller extent the Scopes manager and tree-based routing module. The requirements for *static* RAM amounts to 2.6 KB, mainly due to the `scopes-selfur` module (i.e., tree-based routing) that keeps a number of routing and scope entries simultaneously (this was configured to 10 entries of each type), as well as the size of the fragmentation buffer (configured to 8 packets). These parameters can be tweaked to reduce the RAM requirements, in particular in situations where it is known in advance what applications might run on the ukuFlow network.

The cited techniques include removing all process names as well as debugging information not necessary for the execution of workflows. During system development and evaluation, however, debugging messages are fundamental to understand the behavior and fix bugs. Since debug messages in Contiki are realized through `printf` statements, these are costly in terms of flash space. For this reason, an overhead-free debugging module was employed which enables to quickly choose the log level, from zero (no messages) to five (detailed logging). Recompiling with a particular log-level value enables or disables the corresponding log messages.

Besides the RAM statically allocated for ukuFlow's data structures, the system requires RAM to run code (in the stack space) and to allocate dynamic memory (in the heap). The next sections inspect the dynamic behavior, looking at memory requirements and execution time.

[1] Note that although newer platforms have a flash unit larger than that of the original TelosB node, current toolchains are not yet able to fully exploit MCUs with more than 64 KB (i.e., an address space of 2^{16} entries). Experimental extensions exist that add support for 20-bit pointers, which enable addressing the *far* memory region beyond the first 64 KB, but current bootstrap loaders have yet to be extended.

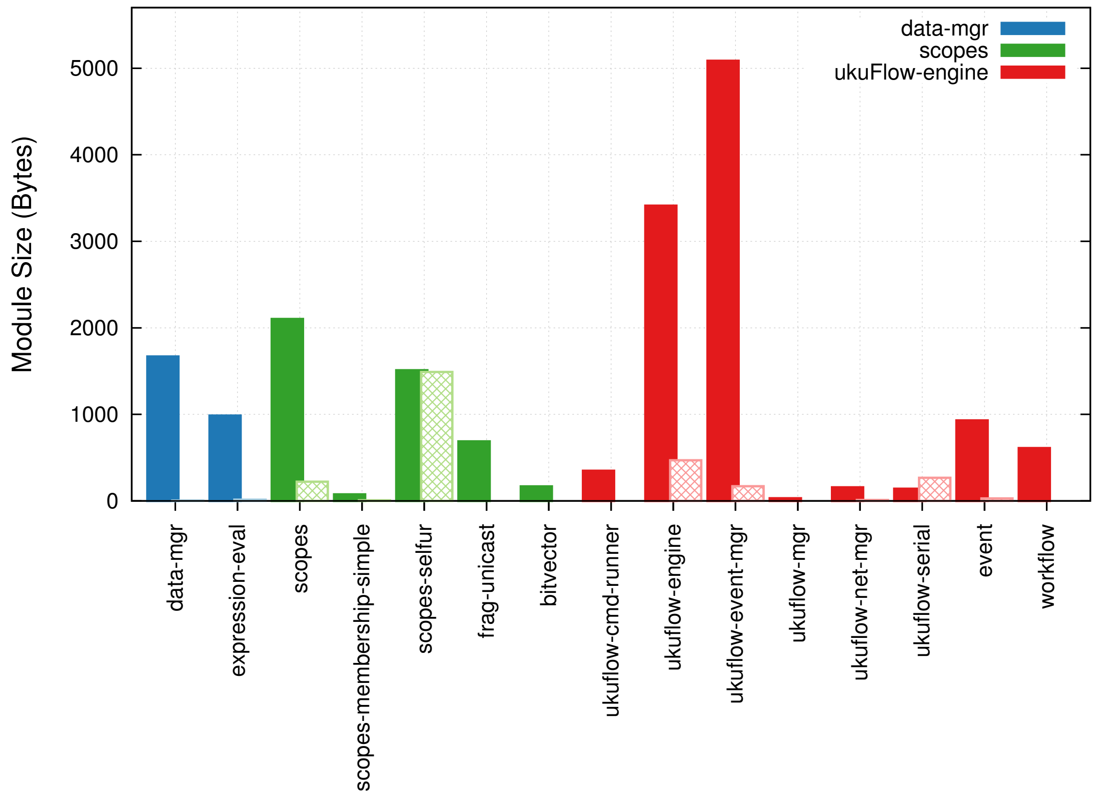


Figure 6.2.: The ukuFlow runtime modules binaries' size and their RAM requirements

6.2 Data Management

The data manager is a central component to the entire framework: it is used by the Scopes framework for storing node properties and checking a node's membership to the received scope specifications, and by the ukuFlow engine for workflow instance's data. The data manager was built as a flexible service that does not require a previous declaration of which and how many name-value pairs will be required, or how wide they should be.

Instead of using a preallocated, static memory region (some of which might remain unused), this flexibility is achieved using *dynamic memory* organized into 2 levels, as shown in Fig. 6.3. The first level contains representatives of each repository. A module that requires a repository is given a *repository ID*, which is used later as handle to access its name-value pairs. The second level contains the name-value pair entries themselves. The implementation makes extensive use of the `list` data structure offered by Contiki. We further reduce memory requirements by not storing summary information such as list lengths - this is traded off with longer execution time for its recalculation. Most operations have a complexity linear to the number of repositories and entries.

We have evaluated the usage of the data manager on sensor nodes with an MSP430 MCU, and found out that it is possible to allocate a total of around **4 KB** of RAM dynamically. Objects in the first level have a fixed width of 14 bytes. The width of objects in the second level, i.e., name-value pairs, depends on the length of the *value* field. This field can have 1, 2 or 4 bytes, leading to a total of 10, 12 or 14 bytes (after including the padding). Figure 6.4 shows the relationship between the number of repositories that a node can handle and the maximum number of entries per repository. This helps us to find a balance between the two dimensions. For instance, allowing only 1 repository would enable 398 name-value pair entries in it (at maximum), while 50 repositories would enable only 6 entries per repository. While

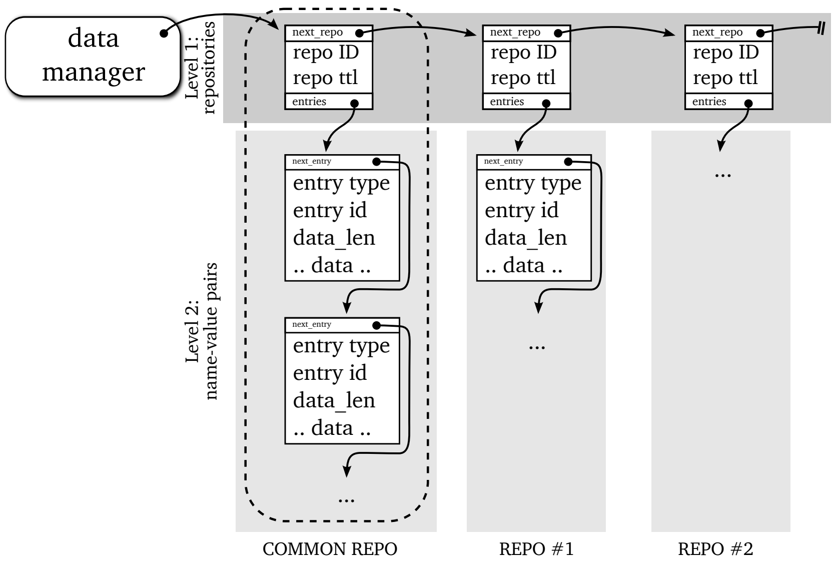


Figure 6.3.: Two-level lists organization of the data manager

these parameters can be adjusted, we have chosen to have a maximum of 20 repositories, each then of a maximum of 15 entries. Note that these 15 entries are for application-specific or user-defined values, and are in addition to the ~15 entries of the *common* repository described in Section 4.4.

The usage of dynamic memory in a sensor network system should not be taken for granted. Indeed, TinyOS and nesC were initially constructed around a completely static memory model, where dynamic memory and function pointers were not allowed in order to simplify the management and avoid the overhead of memory allocation. Only more recent OSs like SOS and Contiki departed from this restriction and re-enabled the usage of the `malloc` and `free` primitives.

We investigated this overhead on a TelosB node closely and found that the added flexibility can be afforded despite the higher cost. The data management operations occur at the two levels: repositories and name-value pairs. Figure 6.5 presents histograms of the execution times for repository creation (left, uses `malloc`) and deletion (right, uses `free`). The plots in Fig. 6.6 are the execution times for creating a new entry (left, uses `malloc`) and overwriting an existing one (right). In general, we observe that compared to the `free` operation, `malloc` is slower and has a higher variance. The average time to create a repository entry averages 1056 μs, creating a name-value pair with a 16-bit value field requires slightly longer, 1344 μs. To put this operation in perspective: a simple write onto 10 bytes statically allocated

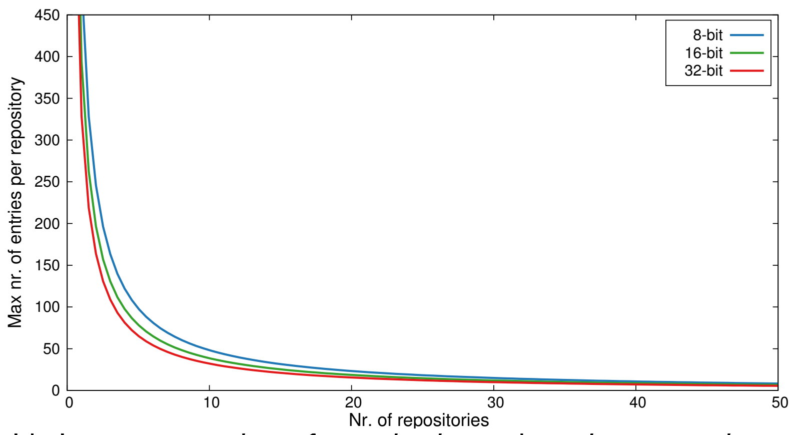


Figure 6.4.: Relationship between *number of repositories* and *maximum number of name-value pairs per repository*, for different lengths of the *value* field, on a TelosB node.

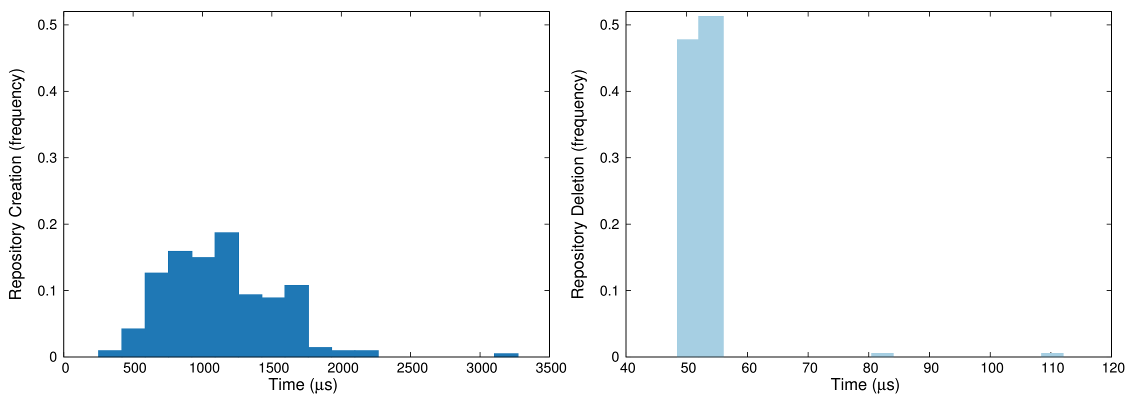


Figure 6.5.: Histograms of completion times for repository operations. Left: repository creation, right: repository deletion

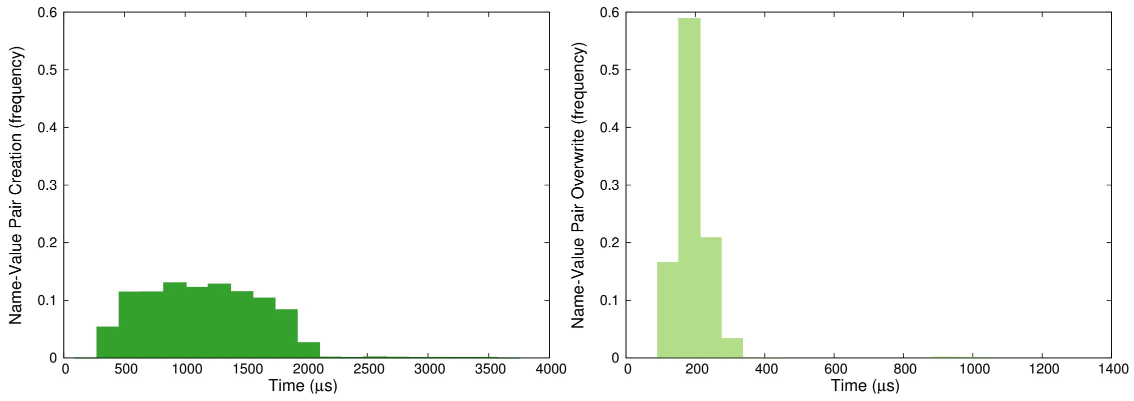


Figure 6.6.: Histograms of completion times for operations on name-value pairs. Left: creation of a new name-value pair, right: write into name-value pair in-place. The 4x overhead for writing 10 bytes of data is tolerable given the added flexibility.

requires 270 μs, which yields an overhead of 4x. Overwriting an already existing name-value pair, e.g., to update sensor data, is much faster at an average of 176 μs. Deleting an entry is considerably fast at 41 μs. We argue that the incurred overhead is tolerable, considering the benefits from dynamic memory of reorganization at runtime and allowing the reuse of unneeded, freed memory, for other purposes.

The available RAM on a node is not used exclusively for data entries: also the Scopes framework and ukuFlow workflow objects require memory dynamically. In the next sections we evaluate the interplay between these subsystems.

6.3 Workflow Execution

In this section we evaluate ukuFlow's execution performance. Since the execution of function statements is performed in asynchronous fashion, we concentrate on computation statements instead, which are CPU-bound, and look at the overall context switching *overhead* incurred by the virtual machine approach of ukuFlow. To evaluate the execution time of computation statements, we employed three different expressions (presented in Listing 6.1), and measured their individual execution times, as well as the total time.

Table 6.2 presents the results of the measurements for both TelosB and XM1000 nodes. The bottom row presents the total time required to execute all three expressions plus the assignment of the initial values

```
a=3; b=12; c=5;
d = a + b - c;                           // expression 1, d=10
e = ((a * d) + (a * b) - (c * d)) % 2;   // expression 2, e=(30+36-50) % 2=16 % 2=0
f = (e != 0) OR (b > c);                 // expression 3, f=1
```

Listing 6.1: Computation statements used for tests

to the variables. As a comparison, we implemented the same logic on an analogous, native C program, and compiled it with most compiler optimizations disabled. The native program took 61 μs on a TelosB node (30 μs on an XM1000), which is a factor of 50 to 60 times faster. From this analysis it follows that while it is possible and convenient to perform simple arithmetic and logic operations with the built-in computation statements, math-intensive workflows can greatly benefit from a custom component written natively for the target platform.

Table 6.2.: Execution times for several CPU-bound expressions

expression	time (μs)	
	TelosB	XM1000
expr. 1	610	244
expr. 2	1251	518
expr. 3	732	305
average	598	247
st. dev.	360	148
total	**3601**	**1495**

In ukuFlow, context switching from one running token to another involves a variety of operations. These include sending the token to the corresponding (ready or blocked) queue, posting an event to the respective scheduler protothread (short or long term), allocating and releasing resources for new tokens, etc., i.e., all the management activities not directly contributing to the workflow execution.

We measured the duration of these context switches for a subset of the workflows listed in Section 4.2.4, which yielded an average of 402 μs on a TelosB node, and 183 μs on an XM1000 node. While this penalty is smaller than the fastest operations presented previously (the CPU-bound computation statements), it is not negligible because it is incurred continuously during the lifetime of a workflow instance. Also, the standard deviation observed, 347 μs and 152 μs, respectively, was high. This was due to the varying circumstances in which context switches occur, such as the number of tokens running concurrently, and the length of token queues.

6.4 Workflow Parallelism

This section focuses on the possibility of the ukuFlow framework to deal with multiple workflows. Parallelism is possible only under the assumption that enough RAM is available to instantiate workflows.

In the memory of a ukuFlow manager node, each workflow specification is wrapped with a compact management object called **workflow node**, which is used to track the number of instances already created for that workflow, the state, and the number of instances that need to be created yet (in case this parameter was specified). Each **workflow instance** has a corresponding object in memory that has fields used to track the number of instance tokens, to link to the workflow specification, and to register the instance's repository ID. Finally, a workflow instance has one or more instance **tokens**, which are initialized to point to the start event of the workflow, and seed the execution of the instances. An instance token carries a pointer to the current and previous workflow operator IDs, to the workflow instance (through which it is possible to reach the workflow specification), to operator-specific **token state**, and to the parent token (if any).

In ukuFlow, we resorted to a *hybrid* memory model for these four object types. As shown in Fig. 6.7, these objects form a logical pyramid. It is not simple to predict the exact number of objects needed at all times for each type. However, the predictability increases from the bottom of the pyramid towards the top. For this reason, ukuFlow uses *dynamic memory* for workflow tokens and operator-specific token state objects, and a pre-allocated *memory pool* for workflow instances and nodes. This reduces the need to impose an artificial limit to the number of tokens, while enabling fast workflow registration and instance creation.

In order to evaluate the behavior of ukuFlow executing parallel workflows, we performed stress-tests to determine how many instances can be allocated, with a) a single-workflow and b) a mixed-workflow scenario.

The initial implementation of the workflow manager made intensive use of interprocess communication between the long- and short-term schedulers to notify about the completion of tasks and availability of resources. For this service, the Contiki operating system uses an array to store the IDs of the caller and callee modules, the notification type and the actual parameter. Since this array is for most platforms defined to have a size of 8 entries, when requiring a degree of workflow parallelism of above 4 workflow instances, this space would not suffice –and the nodes crashed. We modified the operation so as to require notifying a scheduler only if it does not have pending notifications already. At the cost of this extra verification, this optimization reduces the required number of notifications between these two modules to only 1 message at all times.

For the single-workflow scenario, we employed a linear workflow consisting of 15 operators, i.e., 13 script tasks (each with a single computation statement), plus the start and end events. Due to this linearity, each workflow instance does not have more than one token (and one token state object) at any time. The simplicity of this workflow allows us to reason about CPU and memory utilization. We specified the workflow to execute 100 times, and by varying the number of maximum parallel instances required, we observed how the system behaved and coped with overload. For each parameter, we repeated the

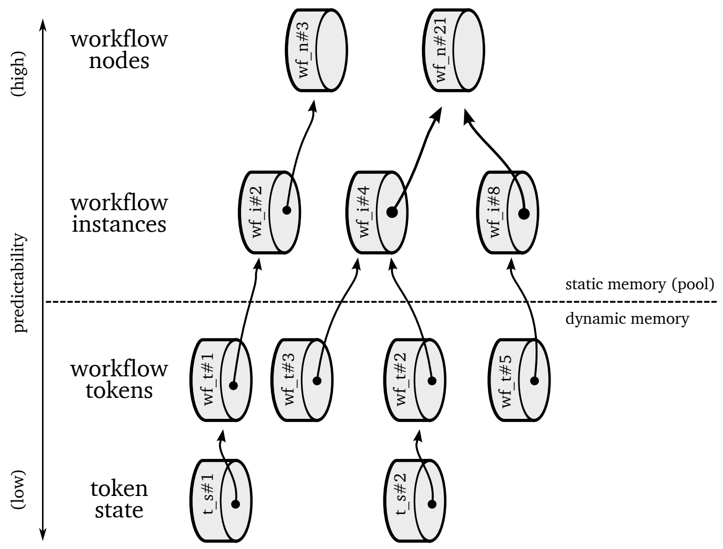


Figure 6.7.: In-memory objects for process management. The number of objects at the bottom of the pyramid is difficult to predict, but is better known towards the top.

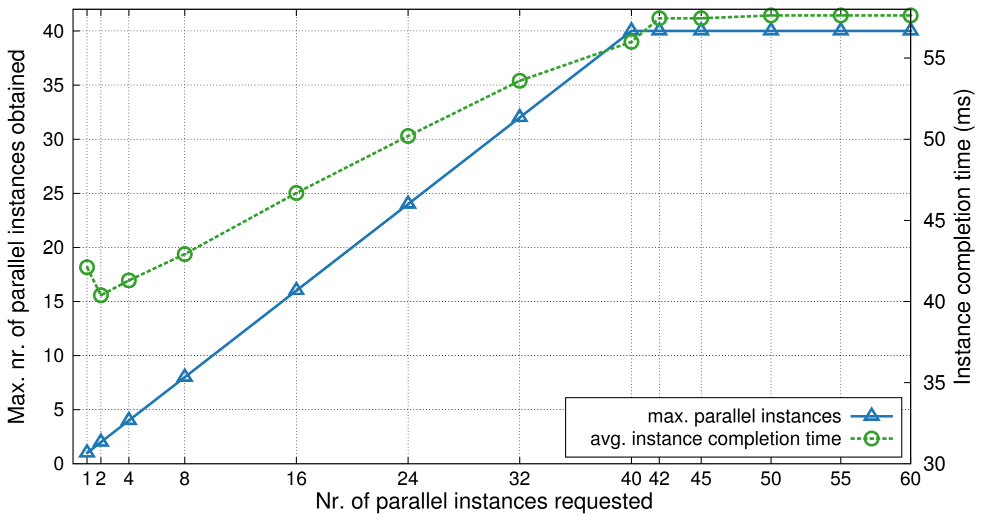


Figure 6.8.: Support for parallelism in a single-workflow scenario.

test three times and obtained the average values (the observed standard deviation was negligible, since this is a deterministic workflow).

Figure 6.8 presents the results for TelosB nodes. The horizontal axis presents the number of parallel instances requested (with the `max. # of instances required` parameter). The blue curve (which reads with the vertical axis on the left) represents the maximum number of workflow instances observed in parallel: a maximum of **40** instances were possible. When requiring more instances than that, the system failed to instantiate one of the necessary objects in dynamic memory (a data repository, a token or its token state object), and thus sent the workflow to the blocked state, where it waited until resources became available again. The green curve (reads with the vertical axis on the right) represents the time required in average by a single instance to complete its execution. It is worth noting that the average workflow instance completion time for required degrees of parallelism of 2 or 4 were **lower** than when requiring only one instance. By interpolation, 6 parallel instances were possibly at the break-even point for being just as fast as with only 1 instance. Parallelism degrees above 6 increased the average workflow instance completion time in near linear fashion. For parallelism degree values above 40, the system hit its maximum capacity, and thus the curve flattened.

For the mixed-workflow scenario, we employed three workflows: 1) a linear workflow, similar to the one used before, but limited to 8 operators (start and end events, and 6 script tasks), 2) a workflow with an exclusive decision gateway, and 3) a workflow with a nested fork operator. The first workflow only allocates 1 token during its execution; the second uses 2 tokens when executing the script tasks succeeding from the decision gateway. In the third workflow, this number varies from 0 to 3 across the duration of the execution of an instance, and is thus the most expensive in terms of memory utilization.

For this test case, we required each of the three workflows to be instantiated 100 times, and varied the number of parallel instances required. The results of the evaluation on a TelosB node are presented in Fig. 6.9. These plots show the active instances being executed in parallel across the duration of the experiment. Instances are encoded with a color corresponding to their workflow specification. In 6.9a we allow 1 instance per workflow; in 6.9b, 2 per workflow; and in 6.9c we tried with 6. The system was only able to cope with a maximum of **17** instances in this scenario (in contrast to the 40 of the single-workflow scenario). This is due to the lack of RAM to hold the necessary objects. As expected, the time required by instances to complete is proportional to the number of other instances running in parallel at that moment. Only towards the end of an experiment, where fewer workflows were active, the completion time was much shorter than when the maximum number of instances are active. From this we conclude that higher degrees of parallelism are possible, but for an increased total completion time.

In the next sections we switch from a micro to a macro view of the system and concentrate on the behavior at network scale.

6.5 Network Characterization

For the macro-evaluation of this work, we have resorted to two sites of non-trivial size: the deployments in the Piloty and TIZ buildings, which are integrated in the TUDμNet federation of testbeds. To understand the operation of ukuFlow, and in particular the behavior of Scopes, we investigated the network characteristics of these two sites. Table 6.3 presents their static properties:

Table 6.3.: Properties of selected TUDμNet Sites

Site	**Nodes**	**Physical**	**Distance (m)**			**Density**
		Size (m^3)	**min**	**avg**	**max**	**(n/m^3)**
Piloty	63	30×20×8	1.2	13.9	34.4	0.01
TIZ	60	31×7×3	1.2	10.4	26.5	0.09

From the physical location of the nodes, we characterized the wireless links according to the distance between the corresponding pair of nodes. While the average link length is considerably similar (10.4m in TIZ vs 13.9m in Piloty) between the two sites, their link length distributions (shown in Fig. 6.10) were different: in TIZ, the distribution is positively skewed: many links were short, and there were only few links with a length longer than 20 meters. In contrast, link lengths in Piloty follow more closely a gaussian distribution.

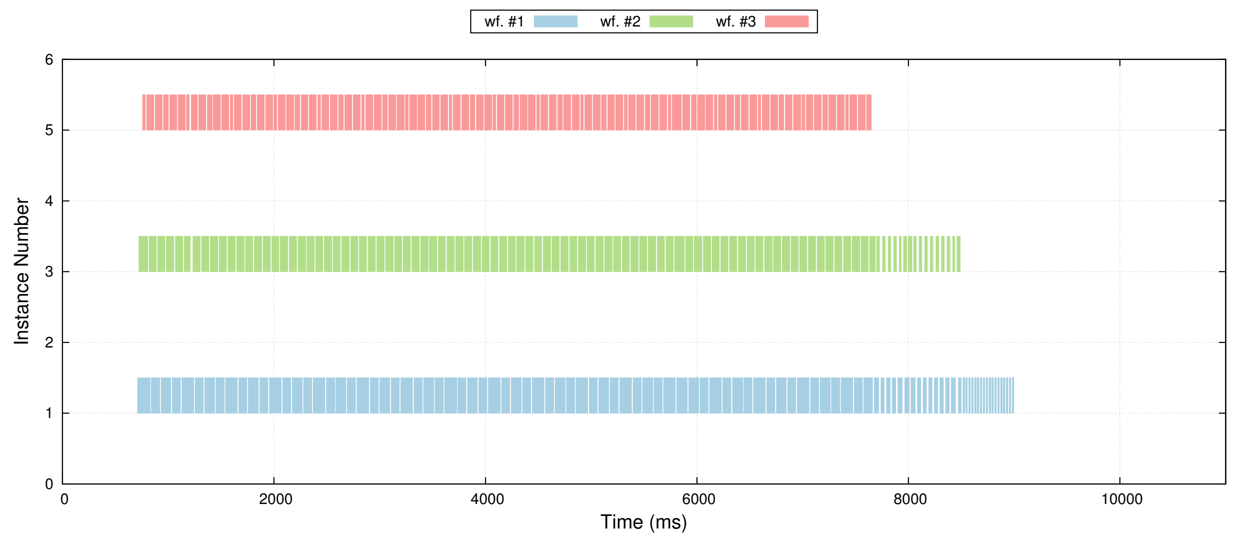
To capture the dynamic properties of these sites, we assessed the quality of every network link, i.e., the communication between any pair of nodes. For this purpose we developed a Contiki tool that exchanges probes among all nodes sequentially. In round-robin fashion, the *master* node:

1. designates one node as *sender* and all other as *receivers*,
2. instructs the sender node to begin sending probes –while all other nodes remain idle–, and
3. requests the collection of statistics centrally from all other nodes.

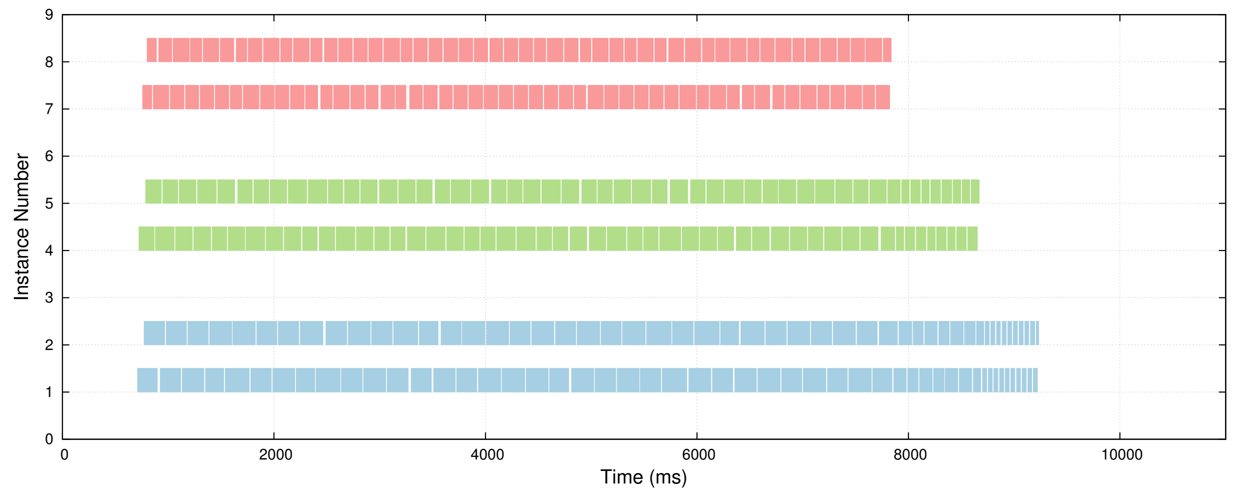
The statistics collected include packet reception ratio (PRR), received signal strength indication (RSSI) and link quality indicator (LQI) of each successfully received probe. From this data set, further dynamic properties of the site were calculated such as *network diameter* and *expected network delivery*. A detailed evaluation of the dynamic properties of these two sites was presented in [57]. Here, instead, we focus on an aspect that is relevant to our study, namely the *node degree*.

Node degree refers to the number of direct neighbors a node has, and results from factors such as node density, site layout and topology. At TIZ, the relatively obstruction-free and dense deployment yields an average node degree of 57.61, which means that nodes can successfully communicate with almost all of the other nodes in the site. In contrast, due to artifacts such as thick walls and other interference, in Piloty the average node degree is only 25.3. This implies that data needs to travel through more hops to reach its destination.

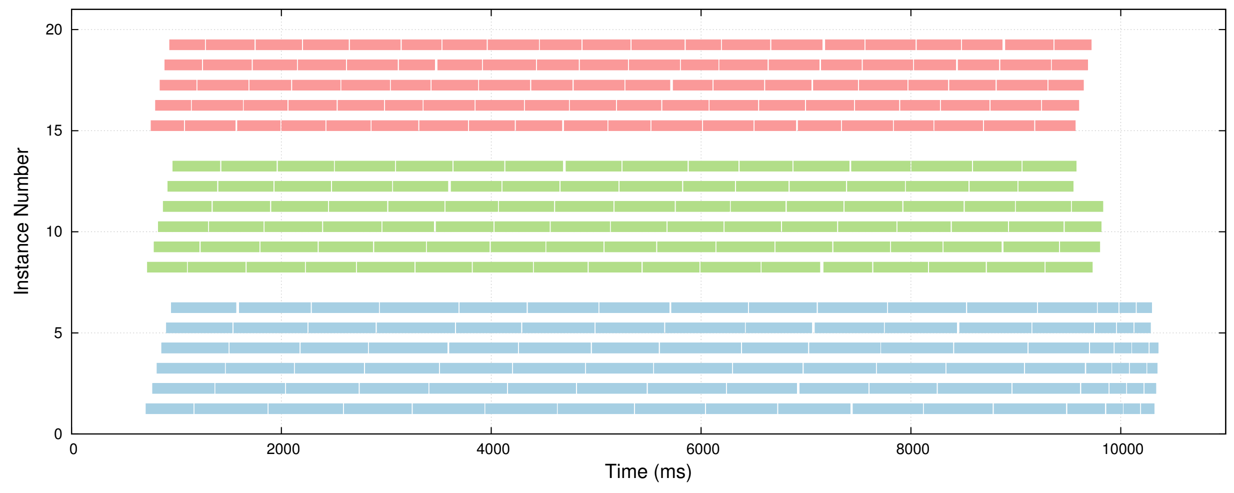
Table 6.4 presents the findings for both sites using different transmission (TX) power levels. The table includes the minimum, average and maximum node degrees observed, as well as a column with the



(a) Test with 1 instance per workflow



(b) Test with 2 instances per workflow



(c) Test with 17 instances (wf. #1: 6, wf. #2: 6, wf. #3: 5) - the maximum reached.

Figure 6.9.: Completion time plots describing the support for parallelism in a mixed-workflow scenario. Higher parallelism degree is possible at the cost of an increased total completion time.

Table 6.4.: The obtained node degree of the two sites for different transmission power levels

Site	TX power	Node Degree			% of links
	(dBm)	min	avg	max	with PRR>0
Piloty	0	12.00	25.30	39.00	41
	-5	5.00	15.71	26.00	25
	-15	3.00	8.03	17.00	13
	-25	0.00	0.31	3.00	1
TIZ	0	51.00	57.61	59.00	89
	-5	38.00	51.91	59.00	80
	-15	18.00	35.81	52.00	55
	-25	0.00	1.06	3.00	2

percentage of links with PRR>0, i.e., link in which at least one probe was received successfully. In the next sections, the evaluations refer always to the highest TX levels (also, at low levels the networks become partitioned).

6.6 Scoping Performance

ukuFlow relays all its network traffic through the Scopes framework. In order to facilitate the understanding of the behavior of the actuation and event detection mechanisms proposed in this work, we first investigate the behavior of the underlying Scopes framework. These results complement those co-published by the author in [72].

For this evaluation, we defined a single test that allows us to isolate and reason about different aspects of interest. In this test, a scope is opened and closed multiple times. Figure 6.11 presents the phases of a single iteration on the x axis, together with the percentage of nodes that were members of the scope at that time instant, on the y axis. Each iteration consists of:

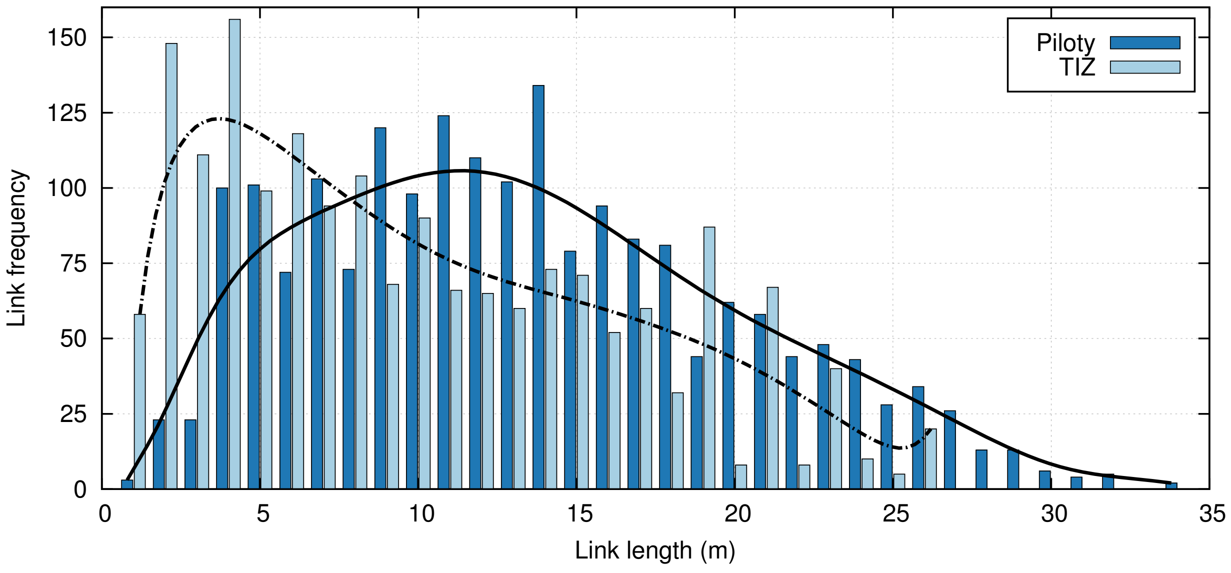


Figure 6.10.: Distribution of link lengths

1. [t_0,t_1), an initial delay *a* where intentionally nothing occurs,
2. [t_1,t_8), scope creation, followed by data traffic,
3. [t_8,t_{10}), scope removal, followed by an inactivity interval to allow for nodes that did not hear the removal message to automatically leave the scope, and
4. [t_{10},t_{11}], a final wait *b* before restarting an iteration.

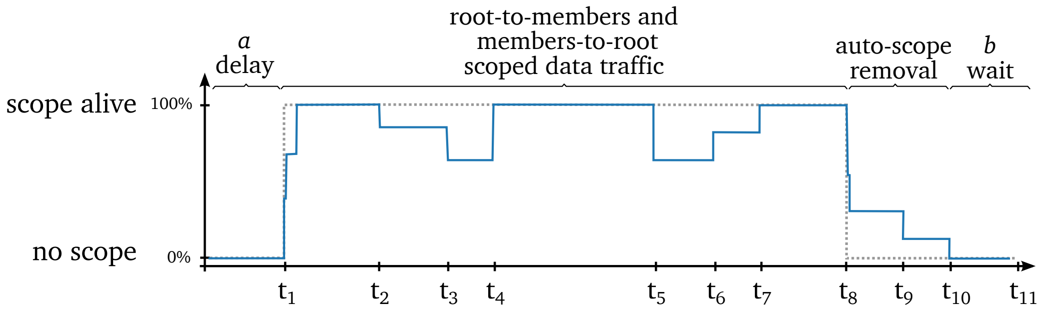


Figure 6.11.: Organization of an iteration for testing scopes

We ran this experiment in both the TIZ and Piloty sites. For each, a node located at a corner was chosen as scope root in order to increase the number of hops. Next, we split the study into three aspects of interest: a) the scope membership efficacy, b) the membership stability, and c) the traffic goodput (i.e., the percentage of packets that arrive at the destination node out of the total number of packets sent by all other nodes).

6.6.1 Scope Membership Efficacy

Given a scope specification and the set of states of each network node, *scope membership efficacy* refers to the percentage of nodes that become member of a scope out of the ideal set of member nodes. In information retrieval terminology, this is equivalent to the concept of *recall*. While false positives could occur[2], none were observed. A common problem, however, were false negatives (nodes that should become scope members but do not). This aspect is relevant to ukuFlow because all non-local workflow *actuation* depends on it to function correctly.

For many scope specifications, it is possible to calculate in advance which nodes should belong to it at a given time. For instance, given a scope specification that uses node positions or their node IDs, and a sensor network's set of nodes, IDs, their positions and data repository values, we can calculate how many nodes should belong to the scope when it is opened. Scope membership efficacy can then be measured empirically by comparing the observed member set against the expected (i.e., ideal) one.

To quantify this aspect, we ran the described test with different network sizes. We used 15, 30, 45 and 60 nodes in TIZ, and 7, 15, 30, 47 and 63 nodes in Piloty. Note that tests in Piloty employed both TelosB as well as XM1000 nodes, thus a certain node heterogeneity is taken into account. Figure 6.12 presents the results for the particular run at Piloty with 30 nodes. The curve shows the observed scope membership efficacy on the *y* axis. The upper plot includes all 10 iterations, and shows that, typically, membership neither is 100% immediately after opening a scope, nor it becomes 0% immediately after closing it. This is due to various network effects. The bottom plot zooms into the initial 10 seconds after opening the scope in the first iteration, and shows how the scope membership increases as the scope specification is disseminated through some (in this case 4) hops.

[2] For instance due to communication errors where certain bits in the scope specification could get swapped

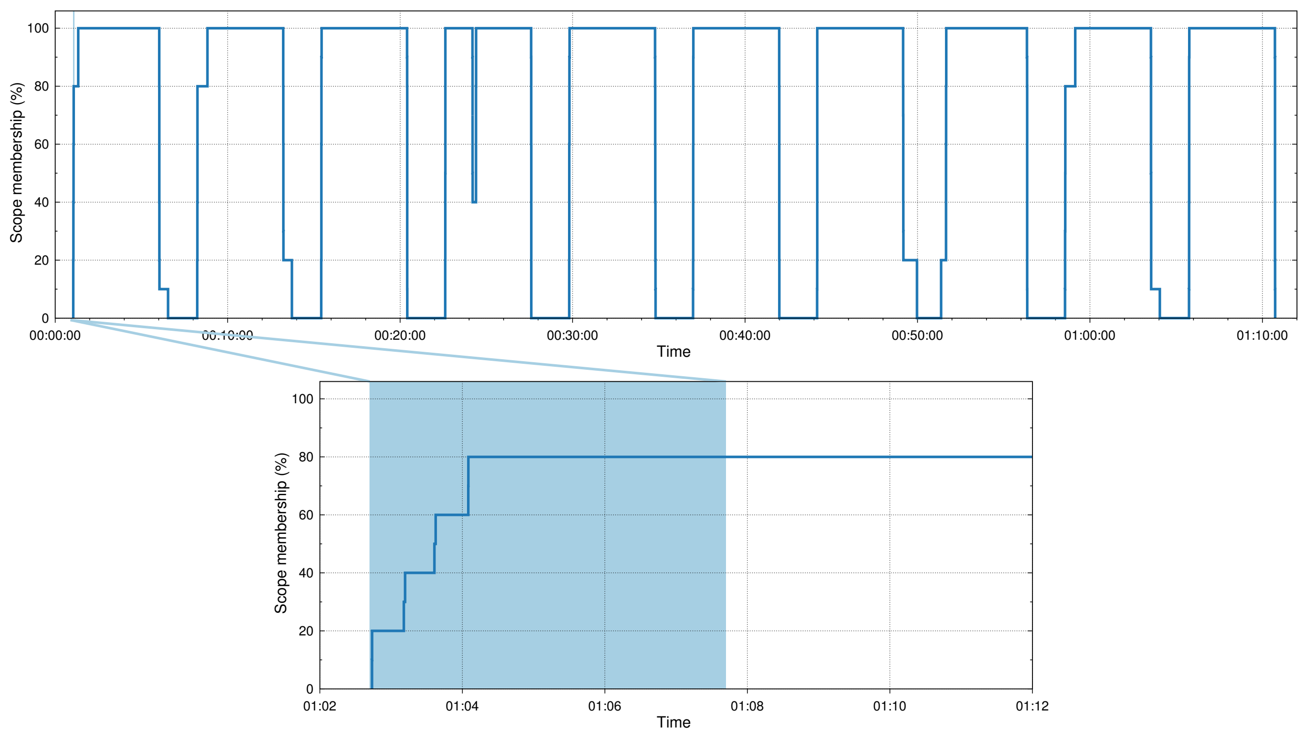


Figure 6.12.: Top: scope membership during the 10 iterations of the test. Typically, membership is not 100% immediately after scope opening, and also not 0% after scope closing. Bottom: zoomed-in view of first 10 seconds in first iteration.

When a scope is opened, the Scopes framework blocks applications from using the scope for a 1-second interval in order to allow the network to stabilize. Through empirical observations, we have found that it is best to extend this interval to reduce the chances of collisions. The ukuFlow Command Runner thus waits 4 further seconds before using the network to send the actual command. For this reason, we are interested in the scope membership achieved in the first 5 seconds *immediately* after opening a scope, since it is those nodes who will potentially receive and execute the issued command (i.e., for actuation it is not relevant if the membership efficacy increases or decreases afterwards). We defined the scope membership efficacy as the average membership in the first 5 seconds (as highlighted in the bottom plot of Fig. 6.12).

Figure 6.13 presents the results of both sites for the mentioned network scales. The green curve shows that in the TIZ deployment, a perfect membership for all network sizes is achieved. This is due to the extremely high node density and node degree properties of that site, where most nodes were reached without the need for multi-hopping. The blue curve, in contrast, shows good results for networks of size 30 or above, but relatively poor for network sizes of 15 or lower. This is mostly due to the physical properties of the site, which is characterized by intermittent links between physically nearby nodes. In Piloty deployment, only when a larger number of nodes is included, it is possible to have alternative paths to the nodes, through which a scope specification is disseminated. This is an important aspect to consider when deploying any wireless network, and neither ukuFlow nor Scopes make provisions to counter this.

Scope creation inevitably requires disseminating a scope specification throughout the entire network. In the *worst* case, the network has a topology in which the cost of opening a scope is *linear* to the number nodes. In addition, when the network is idle and no scopes exist, an initial routing tree needs to be

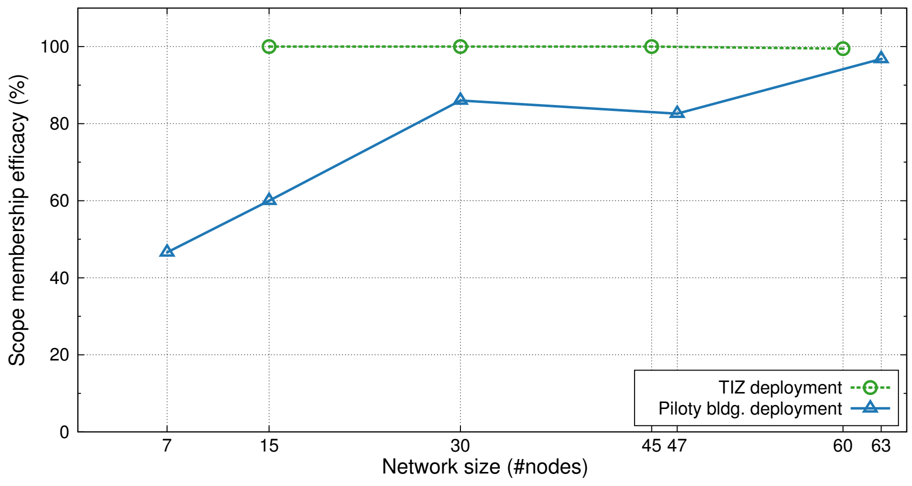


Figure 6.13.: Scope efficacy in both TIZ and Piloty deployments

created, which further duplicates the cost[3]. Finally, depending on the scope specification and the actual number of member nodes, there is a certain traffic for activation and suppression messages (as described in Section 4.5). In many setups, however, the usage of polite-gossiping greatly reduces the number of message broadcasts required.

Figure 6.14 presents the cost for the previously presented cases in terms of the number of messages sent for both the tree creation *and* the scope dissemination operations. The curves show that the cost is considerably lower than the worst case (around 2x the number of nodes). The high node degree of the TIZ deployment means that a single broadcast from one node will be overheard by many neighbors, which will politely stay quiet. In such setups, it is possible to open a scope with a very high efficacy at very low cost (e.g., it required only 11 messages in the 60-node network). The deployment in Piloty, however, proved to be a more challenging one due to the need for multihopping, and exhibited a rather linear behavior. This is still an improvement over a traditional flooding scheme. From these evaluations we conclude that creating a scope is an expensive operation, and should thus be used sparsely.

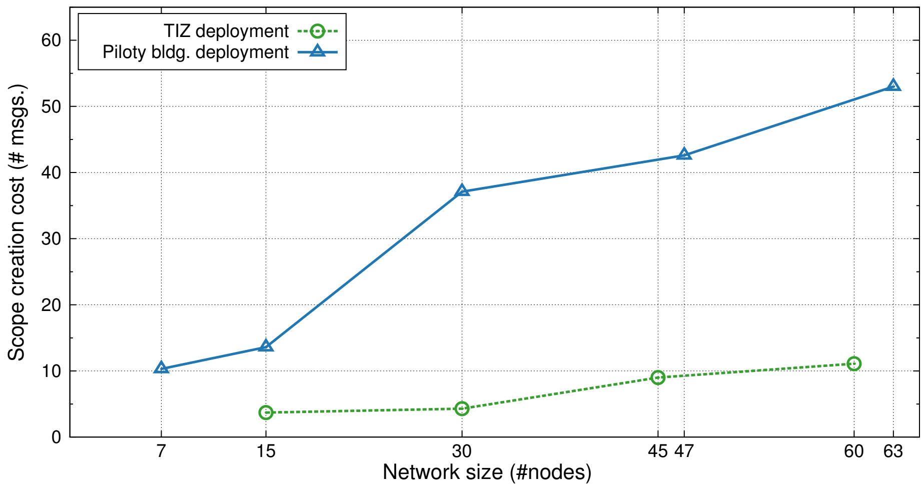


Figure 6.14.: Scope creation cost in both TIZ and Piloty deployments

[3] Recall that the creation of this tree is only triggered when there were no other scopes in use. Posterior scopes use the tree rooted at the scope root, so the cost is recouped for the normal case.

6.6.2 Scope Membership Stability

While scope membership efficacy refers to the *initial* percentage of member nodes, the *stability* refers to the unwanted fluctuations in the scope membership during its lifetime. This aspect is also important for *event detection* because ukuFlow requires nodes to continuously contribute to the final event during the time they are needed, i.e. during the time the scope is open.

In order to quantify the membership stability, we looked at the difference between the areas under the ideal and observed membership curves. We distinguished between a) the instability *during* the lifetime of the scope, and b) the instability *after* the scope is removed. These differences correspond to the size of the dark-colored areas in Fig. 6.15. Lifetime instability is due to nodes not hearing the scope specification message disseminated after the scope is created, or not hearing a sequence of scope refresh messages. Removal instability occurs when nodes miss the scope deletion message, and thus execute the auto-removal subsequently. High instability values represent larger dark areas, i.e., higher deviations from the ideal curve.

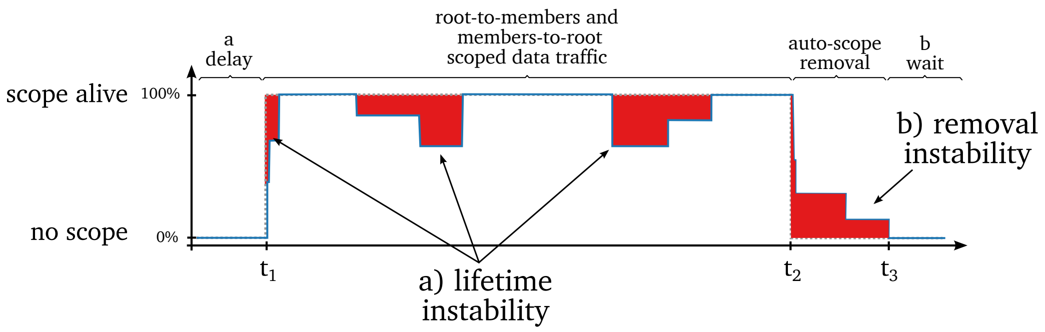


Figure 6.15.: Quantification of scope instability a) during the lifetime of the scope, and b) after scope removal

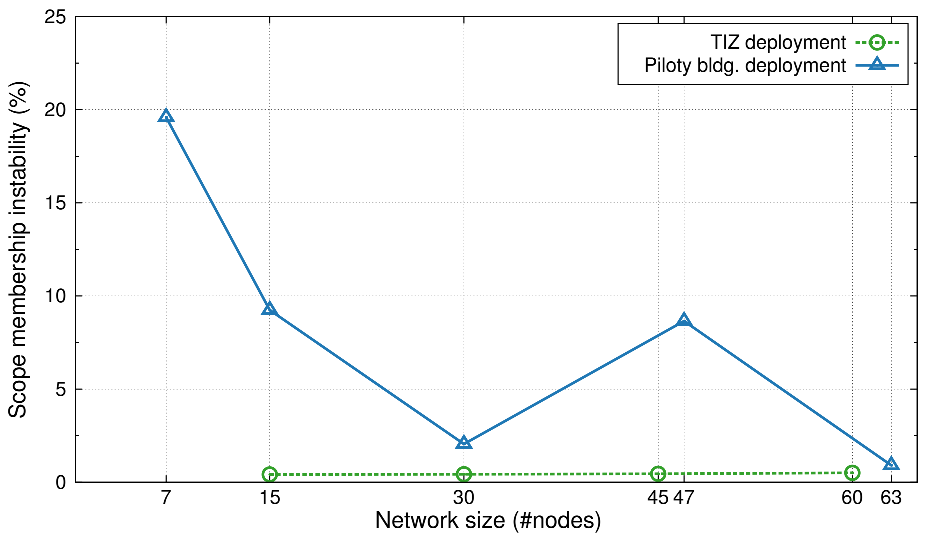
Figure 6.16 presents the results of the membership stability tests for both deployments. A positive result is that at larger network size, both the instability during the scope *lifetime* as well as after *removal* were low, below 5%. Similarly to the efficacy tests, the TIZ site exhibits an almost perfect behavior, while in the Piloty site the membership instability is non-negligibly higher. The curve from the tests at the Piloty site suggest that the instability decreases at higher network sizes, but there is not enough evidence to claim this trend. The variability could be due to the randomness with which trees were constructed, but requires further analysis. The relatively poor membership stability at low network sizes can be attributed to the effect that individual failing nodes have on the overall percentage.

6.6.3 Scope Data Traffic

We now examine the data traffic behavior. For this purpose, we look both at:

1. RtM traffic (i.e., multicast), and
2. MtR traffic (i.e., convergecast), with different payload lengths.

In these experiments, we let the scope root send 6 messages to the scope members. Nodes that receive this message had to answer back with a reply after a random timer. We experimented with three lengths, spanning 1, 2 and 3 fragments. Figure 6.17 presents the goodput in terms of the percentage of received messages for each direction and number of fragments. The plot for the TIZ site showed that RtM traffic



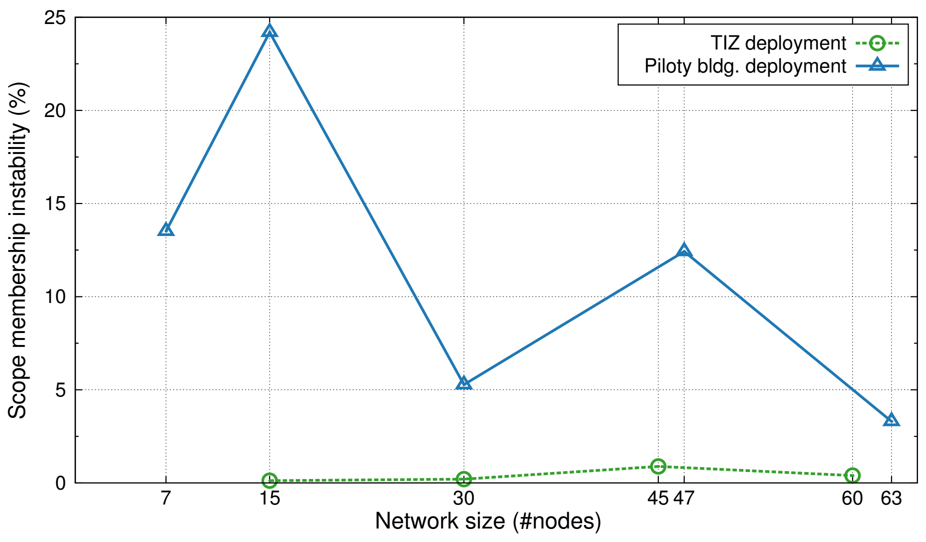


Figure 6.16.: Observed scope instability. Top: during the lifetime of the scope, bottom: after scope removal

was not affected at higher network scales: it remained constant at around 95%. In contrast, MtR traffic did worsen with a larger network scale. This is due to the increased chance of collisions in the network, since virtually all nodes were located in a single radio cell. Furthermore, in this site it could be observed that goodput decreased when the messages consisted of 2 or 3 fragments. This follows from the fact that longer messages cause sender nodes to block the receiver node for a longer time. If, during the reception of fragments from node a a fragment from node b is received, the former are discarded (for memory reasons, as described in Section 4.5.1). While this strategy simplifies the node's memory usage, it has a big impact on the throughput.

The bottom plot corresponds to Piloty. In general, the goodput results were not only lower as compared to the TIZ site, but also their variance is higher. This was mainly due to the larger number of hops through which retransmissions must occur. First, RtM traffic averages 73% across all network scales, a difference of 23% compared to TIZ. Furthermore, MtR traffic goodput was low, varying between 5% and 25%. Interestingly, no correlation was observed between the number of fragments and the goodput (for instance, MtR traffic with 2 fragments was more reliable than with just 1). In MtR traffic, the trend to have lower goodput when more nodes are members of the scope was also observed. As a result of being a true multihop environment, the routing trees constructed by the underlying protocol varied from one

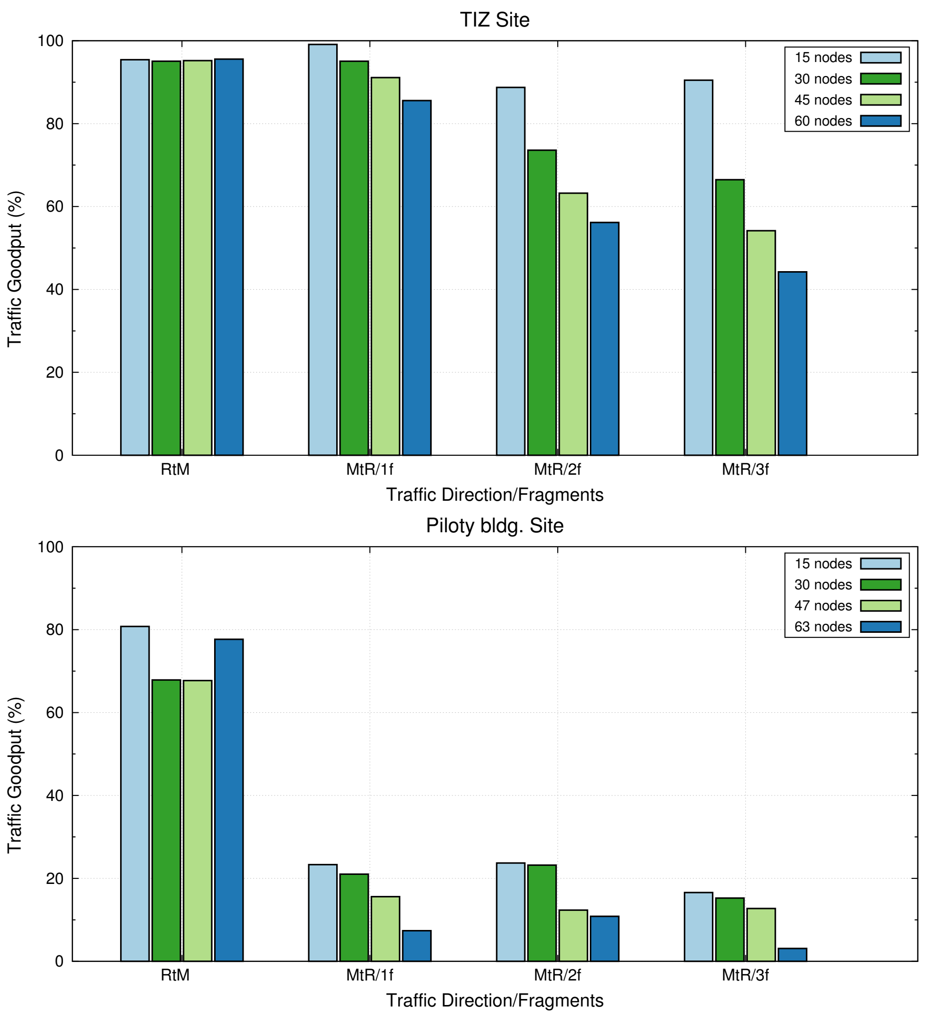


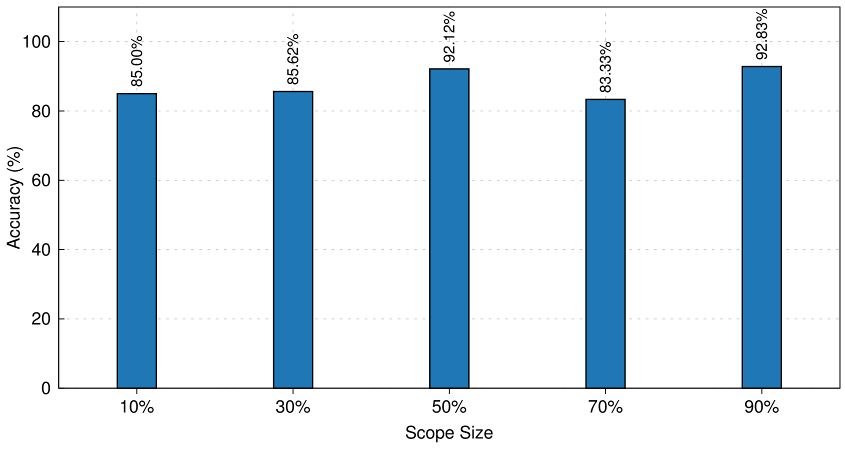
Figure 6.17.: Data goodput for both studied sites

run to the other, causing a higher variability in the performance results. The Piloty site thus exhibited much harsher conditions than TIZ, and is thus taken as a challenging setup to evaluate the ukuFlow concepts implemented in this thesis.

One promising optimization that is left as future work consists in improving the routing tree construction process by considering metrics other than minimum hop count. With this routing metric, long-lived routing trees tend to change over time to ones with shortest paths (as calculated by the minimum number of hops). For instance, a node at a hop distance of n might hear a scope tree construction request with which it would be at distance $n-1$, and thus choose it. While these paths have fewer hops, they might not be of better quality (i.e., have higher loss rates). In contrast, considerable performance improvements have been achieved in other routing protocols based on the expected transmission count (ETX) metric. Instead of choosing a tree parent based on the number of hops to the root, the *quality* of the respective paths is compared. As a result, a path with fewer hops but with poor quality will need fewer retransmissions than a path with more hops, each of which have better quality.

6.7 Actuation

In this section, we evaluate experimentally the actuation capabilities of ukuFlow from a macro, network perspective. To verify how well ukuFlow can execute scoped commands over the network, we carried



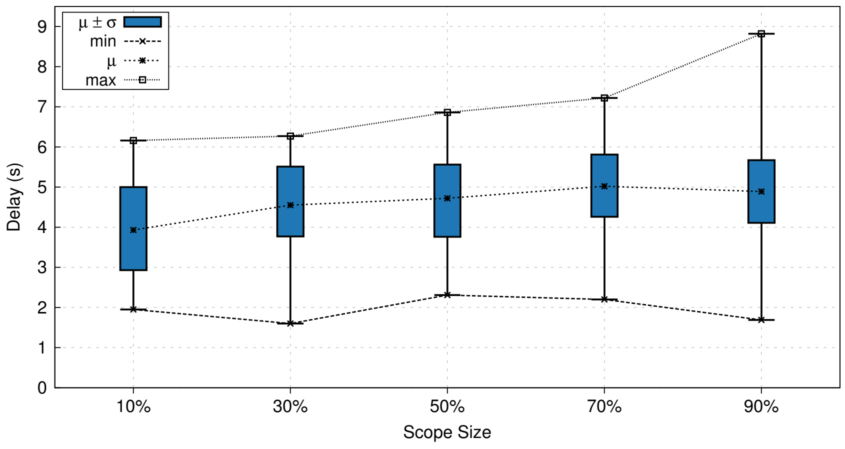


Figure 6.18.: Actuation performance for the Piloty site. Upper plot: average number of nodes, within the scope specification, that ran the scoped function statement. Bottom plot: mean and dispersion for the delay in executing the scoped function statement.

out an experiment in which we ran a workflow consisting of a script task containing a scoped function statement, and set its workflow looping property to iterate 10 times. Each iteration included a pause long enough to let the network return to a still state. We were interested in analyzing both the percentage of nodes within the specified scope that received and executed the scoped statement, as well as the timing properties, i.e., the jitter between nodes as a result of the network dissemination. In this test we focused exclusively on Piloty because of its more challenging nature.

We analysed these two aspects by keeping the network size constant and equal to the entire set of nodes (i.e. 63), while varying the scope specification to include an increasing number of nodes belonging to them, ranging from 10% to 90% of the network size. The two plots in Fig. 6.18 show these 5 scope sizes on the horizontal axis. The upper plot presents the total number of times that nodes executed the function statement, as a percentage of the ideal number, which we call *accuracy*. These values were averaged over the 10 iterations. The results show a slight tendency to increase with larger scope size. When compared to the values for data goodput of RtM traffic with 63 nodes (73.50%), the average actuation accuracy (87.78%) turned out to be higher. This could be due to the fact that the RtM values come from averaging 6 messages over a longer time period, in which the scope instability plays a major role, while the actuation accuracy values only depend on the scope efficacy in the first 5 seconds after opening a scope.

The bottom plot's candlestick chart describes the timing properties of scoped actuation. The measured time represents the latency between a workflow manager node initiating the execution of a scoped function statement, and the command runner on the scope member nodes executing the statement. This latency includes the creation of the scope tree, the 1-second block time, as well as the dissemination of the statement itself. The plot includes minimum, average, maximum, and ± 1 standard deviation from the mean. Here we observe a clearer trend to have a longer latency with larger scope size. This is because larger scope sizes include more nodes, in particular those more distant (in terms of hop-count) from the workflow manager node. While the $|min - max|$ interval increases with scope size, the standard deviation slightly decreases, which implies that more nodes execute the scoped function statement at around the same time. Finally, recall that in Scopes the tree creation and statement dissemination operations were implemented with polite gossiping using the `ipolite` primitive, which makes nodes stay quiet for a time interval before rebroadcasting a message. The length of this time interval is a useful parameter: reducing it indeed reduces the latency, but increases the chances of collisions as well as the energy consumption.

The next two sections focus on ukuFlow's *event manager*, a component that is crucial in making the workflow approach viable in power-constrained environments.

6.8 Event Detection

To evaluate the behavior of the event-based exclusive decision gateway in a real-world setup, our first set of experiments targeted a simple workflow using periodic event generators. For this test, we used a workflow with two event generators and a timer event, as depicted in Fig. 6.19. The periodic event generators were defined to produce their output at a 60-second interval. The timer event was defined to time out after 200 seconds, which allows for two rounds of missed periodic events.

We employed topologies comprising the entire set of nodes in the network. Both event generators were associated to a scope, S_1 and S_2 respectively. These scopes consisted of a disjoint subset of around 5 nodes each, all located at one extreme of the network, while the base station was located at the other extreme. The 3 actions consisted in making the root node blink a number of times, which we used as indicator of a positively detected event at the workflow manager node. The workflow was defined as executing in an infinite loop, and we let the experiment run for around one hour in each site.

Due to the sequential approach to disseminating event expressions, in this simplified scenario where both event generators have the same period and no complex aggregation or filtering takes place, the normal course of action should be that always a temperature event (corresponding to the first event expression) is detected first, and thus Action 1 is executed.

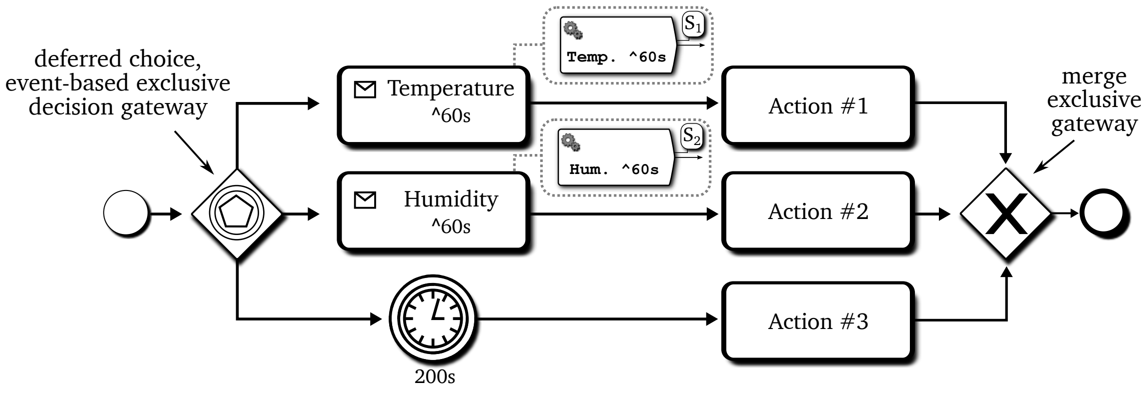


Figure 6.19.: Test workflow with an event-based exclusive decision gw. and periodic event generators

Table 6.5 summarizes the outcome for both sites. During the hour that the test lasted, there were 44 and 46 loops of the workflow, respectively. In the majority of iterations, an event from the temperature generator was received (and thus Action 1 was executed). We validated that this behavior is not dependent on the scope definition or the network topology (but rather on the order in which event expressions are disseminated) by testing with another workflow that alternates the deployment order of S_1 and S_2.

Table 6.5.: Statistics on event detection for the workflow of Fig. 6.19 on the two testbed sites

Site	**Action taken**			**% Events**
	#1	**#2**	**#3**	**Generated**
Piloty	84.09%	15.91%	0.00%	77.76%
TIZ	97.83%	2.17%	0.00%	98.91%

In some cases, an event from the humidity generator was received before all others from the temperature generator (hence, Action 2 was executed). There are a number of reasons why this happened: a) due to the scope creation message of S_1 not being received by the targeted nodes, b) when the event expression message was not received correctly by the nodes member of S_1, or c) when all of the events from the participating nodes got lost. In Piloty, this occurred much more often than in TIZ (15.91% vs 2.17%). In neither site did the timer event fire and trigger Action 3, which means that always an event from the temperature or humidity generators was received.

Finally, the rightmost column in Table 6.5 indicates the number of events generated collectively as a percentage of the ideal number of events that should have been generated if all nodes had become scope members and had received the event expression message. Once again, the TIZ site obtained an almost perfect behavior with $\sim$99% versus a $<$78% at the Piloty site.

Since the event manager uses the data manager's hidden shared data repository to retrieve sensor data, it is possible to tune the maximum frequency with which the costly sensor sampling operations will be performed (cf. Section 4.4). However, a drawback of only using event generators in the event expressions is that events are sent in their raw form to the event manager node, while the latter only needs one of these to decide which outgoing sequence flow to take. It does not matter which of all of the events generated for a certain expression is received, but rather the expression of the event received first. This causes unnecessary traffic (and indeed, many events arrive out-of-order after the first event and before the event manager has begun unsubscribing the now unnecessary event expressions). Next, we investigate the behavior of the two operators that contribute to reducing the traffic by pushing event operators to the network: filtering and aggregation.

The *event filters* are a key event operator to reduce energy consumption. To evaluate these, we used a second workflow consisting of a periodic event generator chained to a simple filter that inspects the event magnitude and requires it to exceed a certain threshold (cf. Fig. 6.20).

While, in practice, the thresholds used in a filter are application-specific parameters provided by domain experts, we identified a range of values for experimentation by profiling the temperature on both sites. Figure 6.21 presents a 24-hour temperature profile for both sites[4]. The upper plot shows a quick increase of temperature at the TIZ site, from 24 °C to 28 °C, in only 3 hours starting at 6:00 am. This is because the TIZ site is formed by offices with large windows that face the east side, which is hit by the sun directly in the morning. The bottom plot shows a candlestick chart where lines correspond to the [min., max.] temperature range, and the boxes span a ± 1 standard deviation, in hourly intervals. Here we observed that the absolute values were much more variable in Piloty than in TIZ. This resulted from nodes in the TIZ deployment being placed in the ceiling, where hot air accumulated and was more uniform than in the Piloty site. From these figures, we chose to use 3 different thresholds at 22, 24 and 26 °C.

[4] The results shown here include a calibration step as reported by others in the literature, e.g., [116].

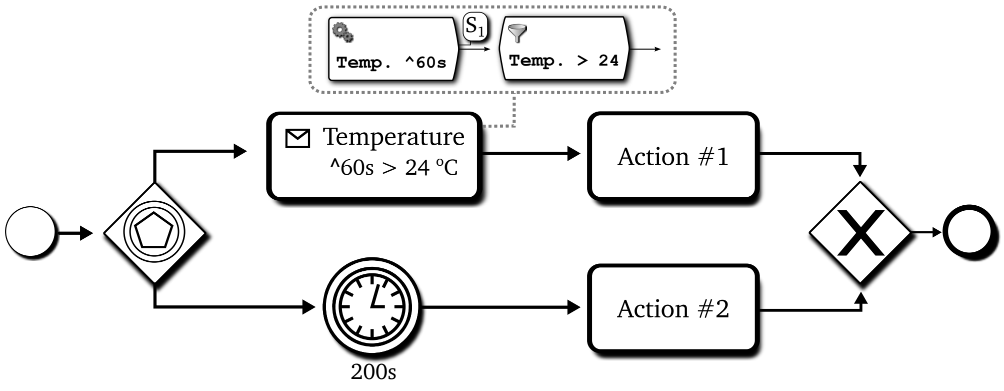


Figure 6.20.: Test workflow with a periodic event generator and a simple filter

We evaluated the performance of this workflow with each of the three thresholds, sequentially; the results are presented in Fig. 6.22. Since we run these tests on TUDμNet, we can centrally log the temperature value generated by each node, regardless of whether that event reached the sink or not. From this information we plot the average, minimum and maximum temperature values read by the nodes in the scope during the event creation by the periodic event generator. In the plot, the three rectangles with a light-blue background highlight the range of values that each simple filter accepts; a

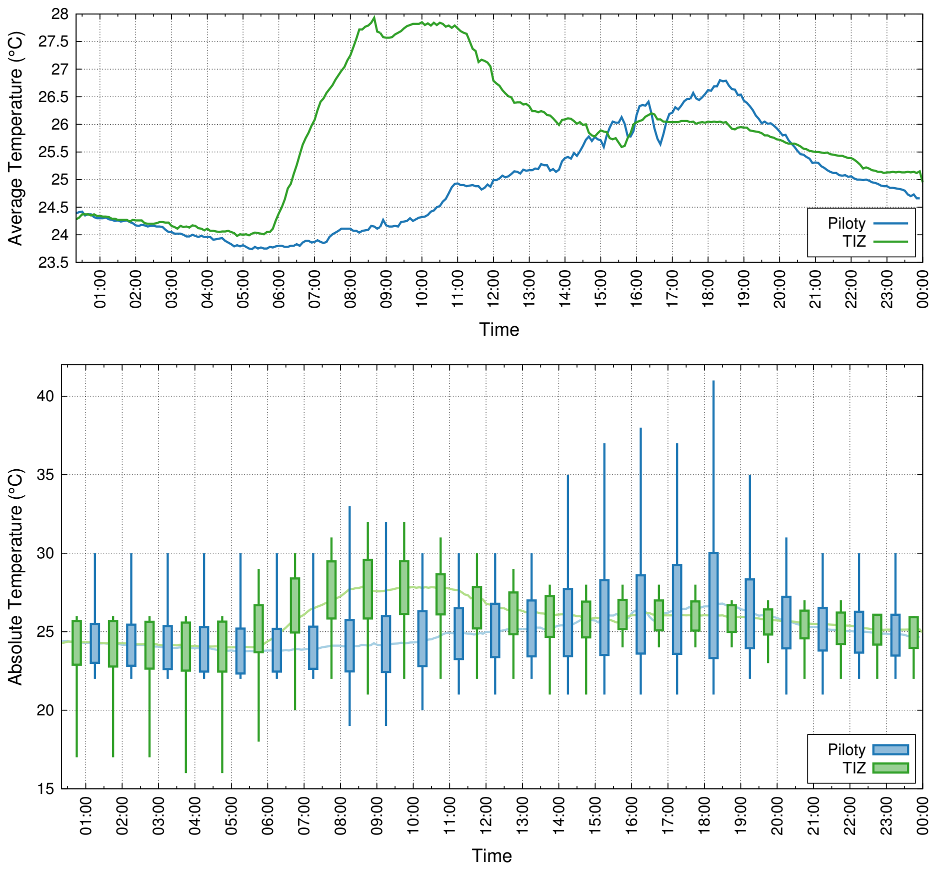


Figure 6.21.: 24-hour temperature profile for Piloty and TIZ sites. Upper plot: average temperature across all nodes, in 15-second intervals. Bottom plot: min, max and dispersion ranges for temperature in hourly intervals.

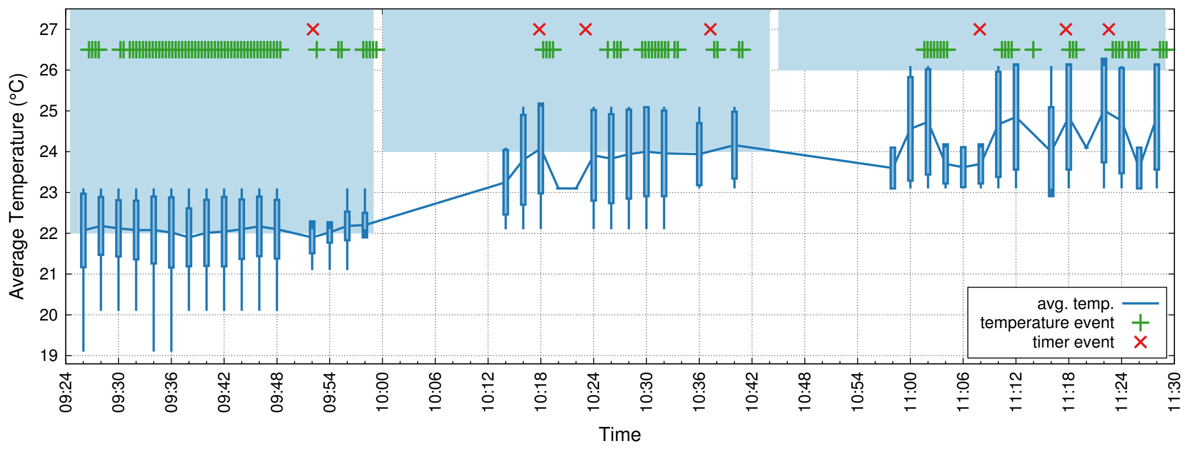


Figure 6.22.: Evaluation of a simple event filter. Green plus symbols represent correctly detected events, while red cross symbols represent time-out events.

box overlapping the rectangle thus represents a situation where the generated events should be reported at the base station. The green plus symbols indicate points in time where events from the simple filter indeed arrived first at the workflow manager node, while red cross symbols indicate that the timer event was detected first. There were three situations where the timer event kicked in first:

- when nodes did not receive the scope specification or the event expression, thus they did not generate the input events (e.g., the first and third red crosses at ∼9:54 and ∼10:23, resp.),
- when nodes generated events, but these did not pass the filter (e.g., at ∼11:08), and
- when events passed the filter, but these were not received at the base station (e.g., at ∼10:18 and ∼10:38).

Despite these special cases, the event detection through simple filters yielded the correct result at least 94% of the time. The net energy savings achieved ultimately depend on the selectivity of the filtering criteria over the input events. In the tested scenario, for instance, 653 events out of 968 (67%) were filtered at the originating node for not passing the filter, and thus did not generate further traffic.

6.9 Complex Event Detection

With the ability to compose individual events into higher level, more meaningful ones, the possible applications increase substantially. Consider the scenario where a user wants to find out how many rooms are there throughout the day with room temperature exceeding the acceptable comfort levels, i.e., above 26 °C. This logic can be addressed by the count event processing composer, configured to consume events from a corresponding simple filter (as discussed in the previous section). In a deployment where only this application would be active, it would be sufficient and necessary to deploy one node per room. In this section we evaluate the performance of this operator.

For our evaluation, we defined a workflow similar to that of Fig. 6.20, and extended it with an event-based gateway using a count event operator as shown in Fig. 6.23. Since there are multiple nodes per room in the employed testbeds, one way to deal with this task is to define a scope with one node per room. In order to span a larger number of nodes in the experiment, however, we used a scope specification with more than one node per room. The time windows of the count event operator and the event generator were configured to act in a 60-second interval. We let this experiment run for 2 hours; in order to understand the behavior at scale, we repeated it with different network sizes.

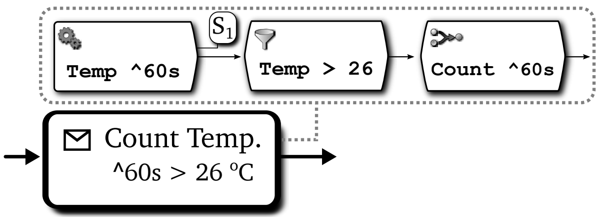
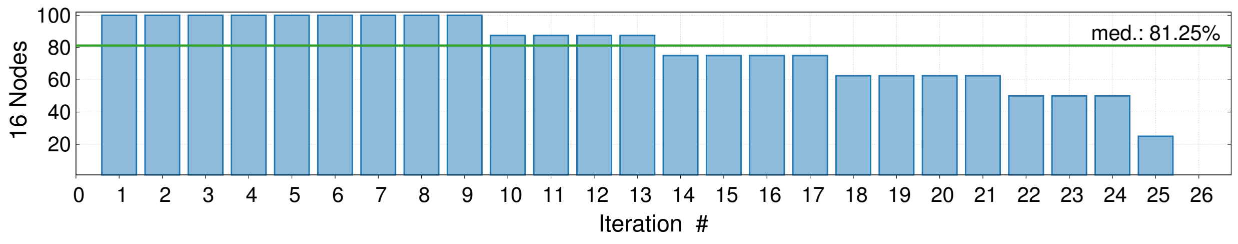


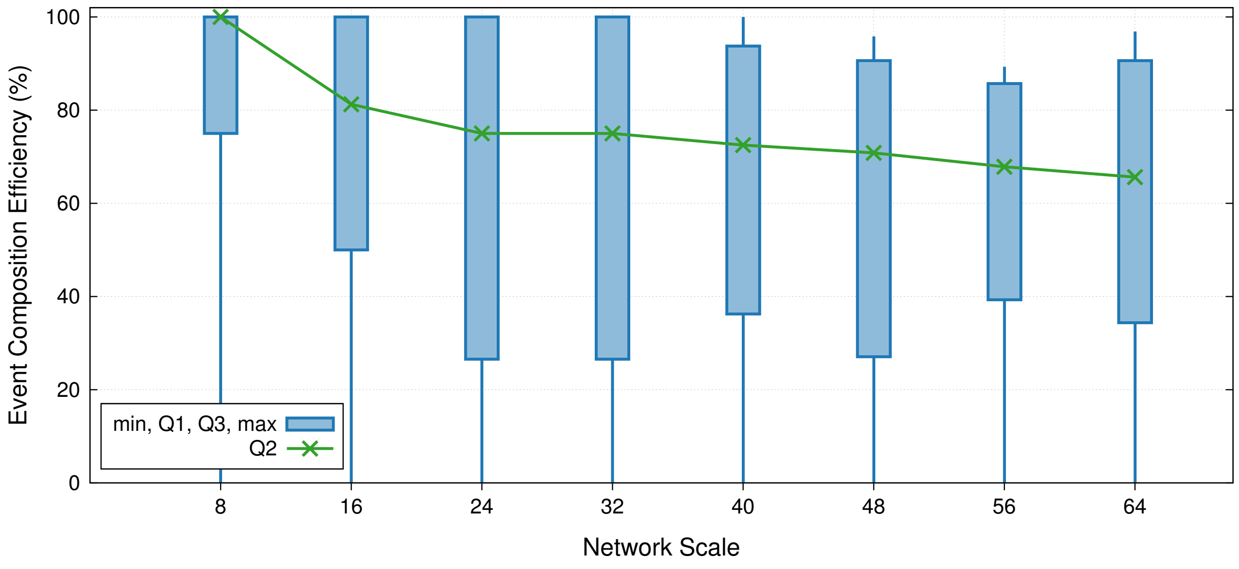
Figure 6.23.: Event expression using the count composite event operator

The overall event expression used in this scenario, composed by the three operators, had a length of 27 bytes. This is considerably shorter than the available payload length of an 802.15.4 message, leaving sufficient space for chaining or nesting the expression to further event expressions.

In Fig. 6.24 we present the results of this evaluation, carried out at the Piloty deployment. In the upper plot, each box represents the percentage of events aggregated and reported correctly at the sink (out of the set of events that should have been generated), sorted in descending order. The 2-hour duration of each test allowed for exactly 26 iterations of the workflow execution. This upper plot corresponds to the particular experiment with 16 nodes, where a median of 81.25% events was observed; a detailed presentation of the results for each network scale is presented in Appendix B.



(a) Test with 16 nodes



(b) Performance summary of all network scales

Figure 6.24.: Event composition performance results. Upper plot: number of events aggregated in the network and reported correctly back at the event manager node in the particular network size of 16 nodes. Bottom plot: summary of all the network sizes evaluated.

The bottom plot is a candlestick chart summarizing the results for each network size, in steps of 8 nodes. Each candlestick's line spans from the absolute minimum and maximum number of events aggregated, the box corresponds to the first and third quartile. Finally, the green curve shows the median for each network size. In general, the results were very good. However, the absolute results had a high variance, ranging from excellent to poor. Indeed, for all network scales there was one iteration out of the 26 where no events were aggregated at all. A closer look at the testbed logs showed that this was due to the event subscription message not being received correctly at any node in the network, which suggests a transient problem at the transmitter side. Although the event manager in ukuFlow accounts for this case by retransmitting the subscription at a regular interval, the particular implementation of the `count` event operator (which mimics the SQL `count` aggregate function) returns a zero count before the expression can be re-disseminated. This situation can be ameliorated by tuning the frequency of subscription retransmission in the ukuFlow event manager, but is left as future work.

The green curve (median) decays with larger network scales, from a value of 100% for a small scale setup of 8 nodes, down to 65% for 64 nodes. This behavior is due to the fact that at larger network sizes, where the network tree is deeper, a single packet lost causes the immediate loss of a higher number of aggregated events. This result matches with the observations made for member-to-root traffic in Section 6.6.3. Depending on the characteristics and requirements of the target application, the performance obtained at the larger network size might be acceptable, or require more advanced and reliable strategies.

6.10 Summary

From the outset, for this work we targeted the understanding of the system in a realistic setup. In this chapter we dived into the performance of the main components that enable ukuFlow as high-level workflow macroprogramming approach running entirely in-network. We have shown its potential to run multiple workflows in parallel, looking at the dynamic memory utilization and time complexity of a CPU-bound process. Following this, we investigated the performance of the network protocols in and underneath Scopes, to better understand the behavior of ukuFlow's actuation and event detection capabilities.

By means of the two main TUDμNet deployments, the realization and evaluation of the platform provided valuable insights into the behavior of the proposed workflow mechanisms. The TIZ deployment was identified as a tractable topology due to its higher node density and shorter network diameter, while Piloty was a very challenging environment characterized by low link instability and large number of hops. Our work with this range of parameters lead us to ensure that ukuFlow offers efficient WfMS algorithms in very disparate setups.

Lastly, it should be noted that certain suboptimal results are due to the wireless technology not being any longer state-of-the art (the hardware design of the radio is over 10 years old), and hence improvements can be expected with newer sensor platforms.

7 Conclusions

In theory, theory and practice are the same. In practice, they are not.

Albert Einstein (1879-1955)

We started this thesis by raising the problem that the development of applications for low-power WSANs requires advanced programming skills, typically available only to the sensor network specialist. The research efforts in this direction had either addressed the low-level developer, or did not match the expectations of domain experts such as volcanologists, civil engineers, biologists or the like.

7.1 Contributions

The largest contribution of this work is the design and evaluation of a macroprogramming system that simplifies the development of application logic for the domain expert, while shielding them from the vagaries of sensor/actuator networks. The approach is based on using **workflows** as first-class citizens to define application logic, which we view as a generic instrument that experts across all domains can understand and use. In addition, this work makes the proposition that the entire logic of the application can be pushed into the network. Despite certain similarities with other projects, to our knowledge, this is the first attempt to offer such a holistic platform.

To this end, we have architected, implemented and evaluated ukuFlow, a holistic workflow management system for low-power embedded sensor and actuator networks. The work includes models, tools and mechanisms to support the end-to-end workflow development and execution lifecycle. Although we initially designed a new workflow model trimmed for WSANs, we soon opted for not reinventing the wheel and adopted an industrial-strength standard, namely BPMN. This brought the issue of simplification since, in its version 2.0, the standard included over a 100 constructs. We have carefully picked out a subset, called uWDL, that enables a wide range of applications, and illustrated it with a complex air traffic application such as surface management operations for handling aircraft stops.

In uWDL we have incorporated three key aspects that enable WSAN applications: 1) scopes, as a means to refine sets of nodes that need to participate in an interaction; 2) statements, as mechanism to execute control logic on the sensor/actuator nodes; and 3) events, as construct to precisely specify the high-level situations that determine which course of action to take. The definition of workflow logic using these constructs is enabled by BPMN2uku, a graphical editor integrated into Eclipse for the specification of both workflow and event models. The control of these building blocks is achieved with an in-network architecture that enables the users to *upload* the workflow specification to a network node, and takes over the execution in an autonomous fashion.

In order to evaluate the performance of this approach, we digressed from the main topic and devoted ourselves to the construction of an entire metropolitan-scale testbed federation, TUDμNet, that spans 4 different realistic environments. This thesis presented the rationale behind the individual testbeds targeted, and discussed key challenges and solutions. We elaborate on the problem of designing a scalable USB backchannel that exhibits a reliable sensor node operation. Today, this testbed federation has been used by over 100 users/10 power users, amounting collectively to around 8,000 hours of experimentation.

The previous chapter presented the empirical evaluation of ukuFlow. The results remind a comparison between a *station wagon* and a *sport coupé*: ukuFlow's strengths lie in offering a flexible palette of services that can be run in-network, and not in a high efficiency in only one of these services. The evaluation contains a detailed analysis of the CPU and memory utilization at a single-node level, showing the suitability of the prototype implementation for mote-class nodes. The macro-view of the system behavior focused on the two more mature sites of the federation: the Piloty and TIZ buildings. Each of these sites include around 60 nodes distributed over a few hundred square meters. Our network characterization shed light on differences between the two sites, identifying the former as the more challenging setup due to artifacts such as thick walls and various sources of interference.

The performance results of the in-network actuation and event detection mechanisms showed the practicability of the autonomous execution of workflows inside a WSAN. The tests showed that the system can be perfectly used for small-scale networks (i.e., fewer than 20 nodes), but also achieve high accuracy at larger scales (60 nodes or more). At scale, the harsh conditions observed in the Piloty site led to a degradation of the efficiency in the detection of complex events, which might not be acceptable in certain applications.

7.2 Future Work

Despite the large effort that this work required, much work is left to be done. Part of this work involves improving the actual hardware platforms in all its dimensions. A critical aspect that our work highlighted was the need to extend the program space of the microcontroller. Although we have shown that the prototypical implementation of ukuFlow's modules fits in a mote-class node such as a TelosB with less than 60 KB of flash, a full-featured implementation would greatly benefit from a program space of 128 KB or more. For instance, a larger flash module could enable validating the workflow's well-formedness dynamically inside the network, instead of relying on having a backend that carries this out only once. Likewise, improved radio transceivers will increase the scoping performance, and in turn the event detection and actuation accuracy.

Certainly, a qualitative assessment of the approach is necessary to understand its benefits and shortcomings. This could be addressed by the creation of domain-specific reference applications which can then be used to more objectively benchmark the development effort across different approaches. Here, the simplified metric of lines of code will likely not suffice, in particular in our approach where a mixed graphical/declarative/imperative macroprogramming model is used. Subsequently, a longitudinal evaluation of the learning curve of uWDL, including the identification of best practices, would be advantageous for its improvement.

Another approach that could enable energy savings as well as increased reliability consists in extending the semi-distributed architecture with a mechanism to automate the workflow distribution among a other nodes. In this way, these *failover* nodes can resume the execution when a workflow manager fails (as discussed in Section 4.8). Additionally, by strategically choosing nodes that are placed at the optimal location in the network for the workflow being run, the communication routes can be shortened and thus the workflow execution times reduced.

The subsystem to run commands could also be extended with further transactional capabilities. This includes: 1) informing the command runner module whether the current number of member nodes in a scope exceeds a specified minimum that should be expected; 2) returning some sort of feedback about the result of the execution of an action, to further control the following execution flow; and 3) include a lightweight atomic commit protocol to coordinate the execution of actions at multiple nodes.

Finally, during the course of this work, two aspects were identified in the area of experimentation and testbeds. First, there is no systematic approach to build WSAN testbeds, which leads to systems that are costly, unreliable, or don't offer the level of reproducibility required in research. Second, current abilities to precisely emulate realistic physical conditions to which WSANs under test are exposed are very limited. Initial efforts target the manipulation of the RF environment [10] and also temperature [11], but much work is left to provide a comprehensive, truly controlled experimentation platform.

Bibliography

[1] Kemal Akkaya and Mohamed Younis. A Survey on Routing Protocols for Wireless Sensor Networks. *Ad Hoc Networks*, 3(3):325–349, May 2005.

[2] Ian F. Akyildiz and Ismail H. Kasimoglu. Wireless Sensor and Actor Networks: Research Challenges. *Ad Hoc Networks*, 2(4):351–367, October 2004.

[3] Gustavo Alonso, Roger Günthör, Mohan Kamath, Divy Agrawal, Amr El Abbadi, and C Mohan. Exotica/FMDC: a Workflow Management System for Mobile and Disconnected Clients. In *Databases and Mobile Computing*, pages 27–45. Springer, 1996.

[4] Wolfgang Apolinarski, Marcus Handte, and Pedro J. Marrón. An Approach for Secure Role Assignment. In *8th Intl. Conference on Intelligent Environments*, pages 34–41, June 2012.

[5] Anish Arora, Emre Ertin, Rajiv Ramnath, Mikhail Nesterenko, and William Leal. Kansei: A High-Fidelity Sensing Testbed. *IEEE Internet Computing*, 10:35–47, 2006.

[6] Asad Awan, Suresh Jagannathan, and Ananth Grama. Macroprogramming Heterogeneous Sensor Networks Using COSMOS. In *2nd ACM SIGOPS European Conference on Computer Systems*, EuroSys'07, pages 159–172, New York, NY, USA, 2007. ACM.

[7] Magdalena Balazinska, Amol Deshpande, Michael J. Franklin, Philipp B. Gibbons, Jim Gray, Suman Nath, Mark Hansen, Michael Liebhold, Alexander Szalay, and Vincent Tao. Data Management in the Worldwide Sensor Web. *IEEE Pervasive Computing*, 6(2):30–40, April 2007.

[8] Gordon Bell. Bell's Law for the Birth and Death of Computer Classes. *Communications of the ACM*, 51(1):86–94, January 2008.

[9] Shah Bhatti, James Carlson, Hui Dai, Jing Deng, Jeff Rose, Anmol Sheth, Brian Shucker, Charles Gruenwald, Adam Torgerson, and Richard Han. Mantis OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms. *ACM/Kluwer Mobile Networks & Applications (MONET), Special Issue on Wireless Sensor Networks*, 10(4):563–579, 2005.

[10] Carlo Alberto Boano, Thiemo Voigt, Claro Noda, Kay Romer, and Marco Zuniga. JamLab: Augmenting Sensornet Testbeds with Realistic and Controlled Interference Generation. In *10th Intl. Conference on Information Processing in Sensor Networks*, IPSN'11, pages 175–186, April 2011.

[11] Carlo Alberto Boano, Marco Zuniga, James Brown, Utz Roedig, Chamath Keppitiyagama, and Kay Römer. TempLab: a Testbed Infrastructure to Study the Impact of Temperature on Wireless Sensor Networks. In Adam Wolisz, Jie Liu, and Lin Zhong, editors, *13th Intl. Conference on Information Processing in Sensor Networks*, IPSN'14, pages 95–106. IEEE/ACM, 2014.

[12] Philipp Bonnet, Johannes Gehrke, and Praveen Seshadri. Querying the Physical World. *IEEE Personal Communications*, 7(5):10–15, October 2000.

[13] Holger Branding, Alejandro P. Buchmann, Thomas Kudrass, and Jürgen Zimmermann. Rules in an Open System: The REACH Rule System. In Norman W. Paton and M. Howard Williams, editors, *Rules in Database Systems*, Workshops in Computing, pages 111–126. Springer London, 1994.

[14] José Ignacio Isaía Brasca. Towards Floating Managers in Scopes – Autonomous Scopes Maintenance. Research Exchange Student Final Presentation, www.dvs.tu-darmstadt.de/staff/guerrero/docs/brasca09floating.pdf, April 2009.

[15] Alejandro P. Buchmann, Stefan Appel, Tobias Freudenreich, Sebastian Frischbier, and Pablo E. Guerrero. From Calls to Events: Architecting Future BPM Systems. In *10th Intl. Conference on Business Process Management*, BPM'12, Tallinn, Estonia, September 2012. Springer.

[16] Jenna Burrell, Tim Brooke, and Richard Beckwith. Vineyard Computing: Sensor Networks in Agricultural Production. *IEEE Pervasive Computing*, 3(1):38–45, January 2004.

[17] Alexandru Caracaş and Alexander Bernauer. Compiling Business Process Models for Sensor Networks. In *Intl. Conference on Distributed Computing in Sensor Systems and Workshops*, DCOSS'11, pages 1–8, June 2011.

[18] Alexandru Caracaş, Thorsten Kramp, Michael Baentsch, Marcus Oestreicher, Thomas Eirich, and Ivan Romanov. Mote Runner: A Multi-language Virtual Machine for Small Embedded Devices. In *3rd. Intl. Conference on Sensor Technologies and Applications*, SENSORCOMM'09, pages 117–125, June 2009.

[19] Matteo Ceriotti, Michele Corra, Leandro D'Orazio, Roberto Doriguzzi, Daniele Facchin, Stefan Guna, Gian Jesi, Renato Cigno, Luca Mottola, Amy L. Murphy, Massimo Pescalli, Gian Picco, Denis Pregnolato, and Carloalberto Torghele. Is There Light at the Ends of the Tunnel? Wireless Sensor Networks for Adaptive Lighting in Road Tunnels. In *10th Int. Conference on Information Processing in Sensor Networks*, IPSN'11, pages 187–198, April 2011.

[20] Sharma Chakravarthy and Qingchun Jiang. *Stream Data Processing: A Quality of Service Perspective: Modeling, Scheduling, Load Shedding, and Complex Event Processing*. Advances in Database Systems. Springer, 2009.

[21] Sharma Chakravarthy, Vidhya Krishnaprasad, Eman Anwar, and Seung-Kyum Kim. Composite Events for Active Databases: Semantics, Contexts and Detection. In *20th Intl. Conference on Very Large Data Bases*, VLDB'94, pages 606–617, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[22] Sharma Chakravarthy and D. Mishra. Snoop: An Expressive Event Specification Language for Active Databases. *Data and Knowledge Engineering*, 14:1–26, November 1994.

[23] K. Mani Chandy. Event-driven Applications: Costs, Benefits and Design Approaches. In *Gartner Application Integration and Web Services Summit*, San Diego, CA, June 2006.

[24] K. Mani Chandy. A Web That Senses and Responds. In Kai Sachs, Ilia Petrov, and Pablo E. Guerrero, editors, *From Active Data Management to Event-Based Systems and More*, volume 6462 of *Lecture Notes in Computer Science*, pages 78–84. Springer Berlin Heidelberg, 2010.

[25] Ioannis Chatzigiannakis, Stefan Fischer, Christos Koninis, Georgios Mylonas, and Dennis Pfisterer. WISEBED: an Open Large-Scale Wireless Sensor Network Testbed. In *1st Intl. Conference on Sensor Networks Applications, Experimentation and Logistics*, volume 29 of *SENSAPPEAL'09*, pages 68–87, ICST, September 2009. Springer-Verlag.

[26] George F. Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design*. International Computer Science Series. Addison-Wesley, 2001.

[27] Riccardo Crepaldi, Simone Friso, Albert Harris III, Michele Mastrogiovanni, Chiara Petrioli, Michele Rossi, Andrea Zanella, and Michele Zorzi. The Design, Deployment, and Analysis of SignetLab: A Sensor Network Testbed and Interactive Management Tool. In *3rd Intl. Conference on Testbeds and Research Infrastructure for the Development of Networks and Communities*, TridentCom'07, pages 1–10, May 2007.

[28] Umeshwar Dayal, Alejandro P. Buchmann, and Dennis R. McCarthy. Rules are Objects Too: A Knowledge Model for an Active, Object-Oriented Database System. In Klaus R. Dittrich, editor, *Advances in Object-Oriented Database Systems*, volume 334 of *Lecture Notes in Computer Science*, pages 129–143. Springer Berlin Heidelberg, 1988.

[29] Luisella Dazzi, Clara Fassino, Roberta Saracco, Silvana Quaglini, and Mario Stefanelli. A Patient Workflow Management System Built on Guidelines. In *American Medical Informatics Association Annual Fall Symposium*, AMIA'97, pages 146–150, Nashville, 1997.

[30] Manjunath Doddavenkatappa, Mun Choon Chan, and Akhihebbal L. Ananda. Indriya: A Low-Cost, 3D Wireless Sensor Network Testbed. In *7th Intl. Conference on Testbeds and Research Infrastructure for the Development of Networks and Communities*, TridentCom'11, pages 302–316, Shanghai, China, April 2011.

[31] Marlon Dumas, Luciano García-Bañuelos, and Artem Polyvyanyy. Unraveling Unstructured Process Models. In Jan Mendling, Matthias Weidlich, and Mathias Weske, editors, *Business Process Modeling Notation*, volume 67 of *Lecture Notes in Business Information Processing*, pages 1–7. Springer Berlin Heidelberg, 2011.

[32] Adam Dunkels. Full TCP/IP for 8-bit Architectures. In *1st Intl. Conference on Mobile Systems, Applications and Services*, MobiSys '03, pages 85–98, New York, NY, USA, 2003. ACM.

[33] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors. In *29th Annual IEEE Intl. Conference on Local Computer Networks*, LCN'04, pages 455–462, November 2004.

[34] Adam Dunkels, Fredrik Österlind, and Zhitao He. An Adaptive Communication Architecture for Wireless Sensor Networks. In *5th Intl. Conference on Embedded Networked Sensor Systems*, SenSys'07, pages 335–349, New York, NY, USA, 2007. ACM.

[35] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: Simplifying Event-driven Programming of Memory-Constrained Embedded Systems. In *4th Intl. Conference on Embedded Networked Sensor Systems*, SenSys'06, pages 29–42, New York, NY, USA, 2006. ACM.

[36] Opher Etzion and Peter Niblett. *Event Processing in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2010.

[37] Federico Ferrari, Marco Zimmerling, Lothar Thiele, and Olga Saukh. Efficient Network Flooding and Time Synchronization with Glossy. In *10th Intl. Conference on Information Processing in Sensor Networks*, IPSN'11, pages 73–84, April 2011.

[38] Ludger Fiege, Mira Mezini, Gero Mühl, and Alejandro P. Buchmann. Engineering Event-based Systems with Scopes. In B. Magnusson, editor, *European Conference on Object-Oriented Programming*, volume 2374 of *ECOOP'02*, pages 309–333. Springer-Verlag, June 2002.

[39] Chien-Liang Fok, Gruia-Catalin Roman, and Chenyang Lu. Mobile Agent Middleware for Sensor Networks: an Application Case Study. In *4th Intl. Symposium on Information Processing in Sensor Networks*, IPSN'05, Piscataway, NJ, USA, April 2005. IEEE Press.

[40] Chien-Liang Fok, Gruia-Catalin Roman, and Chenyang Lu. Rapid Development and Flexible Deployment of Adaptive Wireless Sensor Network Applications. In *25th IEEE Intl. Conference on Distributed Computing Systems*, ICDCS'05, pages 653–662, Washington, DC, USA, June 2005. IEEE Computer Society.

[41] Chien-Liang Fok, Gruia-Catalin Roman, and Chenyang Lu. Agilla: A Mobile Agent Middleware for Sensor Networks. Technical Report WUCSE-2006-16, Washington University in St. Louis, Department of Computer Science and Engineering, March 2006.

[42] Christian Frank and Kay Römer. Algorithms for Generic Role Assignment in Wireless Sensor Networks. In *3rd Intl. Conference on Embedded Networked Sensor Systems*, SenSys'05, pages 230–242, New York, NY, USA, November 2005. ACM.

[43] Gerhard Fuchs and Reinhard German. UML2 Activity Diagram Based Programming of Wireless Sensor Networks. In *Workshop on Software Engineering for Sensor Network Applications*, SESENA'10, pages 8–13, New York, NY, USA, May 2010. ACM.

[44] Antony Galton and Juan Carlos Augusto. Two Approaches to Event Definition. In *13th Intl. Conference on Database and Expert Systems Applications*, DEXA'02, pages 547–556. Springer, September 2002.

[45] Saurabh Ganeriwal, Ram Kumar, and Mani B. Srivastava. Timing-sync Protocol for Sensor Networks. In *1st Intl. Conference on Embedded Networked Sensor Systems*, SenSys'03, pages 138–149, New York, NY, USA, November 2003. ACM.

[46] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A Holistic Approach to Networked Embedded Systems. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 38 (5) of *PLDI'03*, pages 1–11, New York, NY, USA, June 2003. ACM.

[47] Johannes Gehrke and Samuel Madden. Query Processing in Sensor Networks. *IEEE Pervasive Computing*, 3(1):46–55, March 2004.

[48] Omprakash Gnawali, Rodrigo Fonseca, Kyle Jamieson, David Moss, and Philip Levis. Collection Tree Protocol. In *7th Int. Conference on Embedded Networked Sensor Systems*, SenSys'09, pages 1–14, New York, NY, USA, November 2009. ACM.

[49] Saul Greenberg and Chester Fitchett. Phidgets: Easy Development of Physical Interfaces Through Physical Widgets. In *14th Annual ACM Symposium on User Interface Software and Technology*, UIST'01, pages 209–218, New York, NY, USA, November 2001. ACM.

[50] Rachid Guerraoui and André Schiper. Software-Based Replication for Fault Tolerance. *Computer*, 30(4):68–74, April 1997.

[51] Pablo E. Guerrero, Alejandro Buchmann, Kristof Van Laerhoven, Immanuel Schweizer, Max Mühlhäuser, Thorsten Strufe, Stefan Schneckenburger, Manfred Hegger, and Birgitt Kretzschmar. A Metropolitan-Scale Testbed for Heterogeneous Wireless Sensor Networks to Support CO_2 Reduction. In *2nd. Inttl. Conference on Green Communications and Networking*, GreeNets'12. ICST, October 2012.

[52] Pablo E. Guerrero, Alejandro P. Buchmann, Abdelmajid Khelil, and Kristof Van Laerhoven. Poster Abstract: TUDμNet, a Metropolitan-Scale Federation of Wireless Sensor Network Testbeds. In *9th European Conference on Wireless Sensor Networks*, EWSN'12, February 2012.

[53] Pablo E. Guerrero, Daniel Jacobi, and Alejandro P. Buchmann. Workflow Support for Wireless Sensor and Actor Networks. In *4th Intl. Workshop on Data Management for Sensor Networks*, DMSN'07, pages 31–36, New York, NY, USA, September 2007. ACM.

[54] Pablo E. Guerrero, Kai Sachs, Mariano Cilia, Christof Bornhövd, and Alejandro P. Buchmann. Pushing Business Data Processing Towards the Periphery. In *23rd Intl. Conference on Data Engineering*, ICDE'07, pages 1485–1486. IEEE Computer Society, April 2007.

[55] Ramakrishna Gummadi, Omprakash Gnawali, and Ramesh Govindan. Macro-programming Wireless Sensor Networks Using Kairos. In *Intl. Conference on Distributed Computing in Sensor Systems*, DCOSS'05, pages 126–140, June 2005.

[56] Iliya Gurov. Design and Deployment of a Wireless Sensor Network Testbed for Forest Monitoring. Master's thesis, Technische Universität Darmstadt, February 2013.

[57] Iliya Gurov, Pablo E. Guerrero, Martina Brachmann, Silvia Santini, Kristof Van Laerhoven, and Alejandro P. Buchmann. A Site Properties Assessment Framework for Wireless Sensor Networks. In *11th Intl. Conference on Embedded Networked Sensor Systems*, SenSys'13, pages 32–33. ACM, November 2013.

[58] Gregory Hackmann, Christopher Gill, and Gruia-Catalin Roman. Extending BPEL for Interoperable Pervasive Computing. In *Intl. Conference on Pervasive Services*, ICPS'07, pages 204–213. IEEE, July 2007.

[59] Chih-Chieh Han, Ram Kumar Rengaswamy, Roy Shea, Eddie Kohler, and Mani B. Srivastava. A Dynamic Operating System for Sensor Networks. In *3rd Intl. Conf. on Mobile Systems, Applications, and Services*, pages 163–176, New York, NY, USA, June 2005.

[60] Vlado Handziski, Andreas Köpke, Andreas Willig, and Adam Wolisz. TWIST: A Scalable and Reconfigurable Testbed for Wireless Indoor Experiments with Sensor Networks. In *2nd Intl. Workshop on Multi-hop Ad hoc Networks: from Theory to Reality*, REALMAN'06, pages 63–70, New York, NY, USA, May 2006. ACM.

[61] Matti Hannus, Abdul S. Kazi, and Alain Zarli, editors. *ICT Supported Energy Efficiency in Construction*. REEB Project Consortium, 2010.

[62] Wendy B. Heinzelman, Amy L. Murphy, Hervaldo S. Carvalho, and Mark A. Perillo. Middleware to Support Sensor Network Applications. *IEEE Network*, 18(1):6–14, January 2004.

[63] Dang Quoc Hien. BPMN2uku - An Eclipse Plugin for Generating ukuFlow's Code from Business Process Model Notation. Bsc. thesis, Technische Universität Darmstadt, November 2012.

[64] Jason Hill, Mike Horton, Ralph Kling, and Lakshman Krishnamurthy. The Platforms Enabling Wireless Sensor Networks. *Communications of the ACM*, 47(6):41–46, June 2004.

[65] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System Architecture Directions for Networked Sensors. In *Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, pages 93–104, New York, NY, USA, November 2000. ACM.

[66] Annika Hinze, Kai Sachs, and Alejandro P. Buchmann. Event-based Applications and Enabling Technologies. In *3rd Intl. Conference on Distributed Event-Based Systems*, DEBS'09, pages 1–15, New York, NY, USA, July 2009. ACM.

[67] Bianca Hochberger and Jörg Zentgraf. Entwurf eines Workflow-Management-Systems zur Modellierung und Systemtechnischen Unterstützung der Arbeitsabläufe der Flugsicherung. Master's thesis, Techische Universität Darmstadt, May 2000.

[68] David Hollingsworth. The Workflow Reference Model. Document TC00-1003, Workflow Management Coalition, January 1995.

[69] Bosch Software Innovations. Capitalizing on the Internet of Things – How to Succeed in a Connected World. *White Paper Series*, February 2014.

[70] Daniel Jacobi, Marc Fischlin, and Alejandro P. Buchmann. *Security for Multihop Wireless Networks*, chapter Secure Multipurpose Wireless Sensor Networks. CRC Press. Taylor and Francis Group, April 2014.

[71] Daniel Jacobi, Pablo E. Guerrero, Ilia Petrov, and Alejandro P. Buchmann. Structuring Sensor Networks with Scopes. In *3rd IEEE European Conference on Smart Sensing and Context*, EuroSSC'08, Zurich, Switzerland, October 2008. IEEE Communications Society.

[72] Daniel Jacobi, Pablo E. Guerrero, Ilia Petrov, and Alejandro P. Buchmann. Distributed Network Structuring with Scopes. Technical Report 2741, Technische Universität Darmstadt, Darmstadt, Germany, October 2009.

[73] Holger Karl and Andreas Willig. *Protocols and Architectures for Wireless Sensor Networks*. Wiley & Sons, 1st. edition, June 2005.

[74] Steffen Kilb. Design und Implementierung von Scopes für Contiki. BSc. Thesis, Technische Universität Darmstadt, January 2009.

[75] Sukun Kim, Shamim Pakzad, David E. Culler, James Demmel, Gregory Fenves, Steven Glaser, and Martin Turon. Health Monitoring of Civil Infrastructures Using Wireless Sensor Networks. In *6th Intl. Conference on Information Processing in Sensor Networks*, IPSN'07, pages 254–263, New York, NY, USA, 2007. ACM Press.

[76] JeongGil Ko, Kevin Klues, Christian Richter, Wanja Hofer, Branislav Kusy, Michael Bruenig, Thomas Schmid, Qiang Wang, Prabal Dutta, and Andreas Terzis. Low Power or High Performance? A Tradeoff Whose Time Has Come (and Nearly Gone). In *9th European Conference on Wireless Sensor Networks*, EWSN'12, pages 98–114, Berlin, Heidelberg, February 2012. Springer-Verlag.

[77] John Koenig. JBoss jBPM, November 2004.

[78] Olaf Landsiedel, Federico Ferrari, and Marco Zimmerling. Chaos: Versatile and Efficient All-to-All Data Sharing and In-Network Processing at Scale. In *11th Int. Conference on Networked Sensing Systems*, SenSys'13, pages 1–14, New York, NY, USA, November 2013. ACM.

[79] Koen Langendoen, Aline Baggio, and Otto Visser. Murphy Loves Potatoes: Experiences from a Pilot Sensor Network Deployment in Precision Agriculture. In *14th Intl. Workshop on Parallel and Distributed Real-Time Systems*, WPDRTS'06, April 2006.

[80] Philip Levis. Experiences from a Decade of TinyOS Development. In *10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 207–220, Berkeley, CA, USA, October 2012. USENIX Association.

[81] Philip Levis and David E. Culler. Maté: A Tiny Virtual Machine for Sensor Networks. In *Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pages 85–95, New York, NY, USA, October 2002. ACM Press.

[82] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. In *1st Intl. Conference on Embedded Networked Sensor Systems*, SenSys'03, pages 126–137, New York, NY, USA, November 2003. ACM.

[83] Philip Levis, Samuel Madden, David Gay, Joseph Polastre, Robert Szewczyk, Alec Woo, Eric A. Brewer, and David E. Culler. The Emergence of Networking Abstractions and Techniques in TinyOS. In *1st Symposium on Networked Systems Design and Implementation*, NSDI'04, pages 1–14, Berkeley, CA, USA, March 2004. USENIX Association.

[84] Philip Levis, Neil Patel, David Culler, and Scott Shenker. Trickle: a Self-regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks. In *1st Symposium on Networked Systems Design and Implementation*, NSDI'04, pages 15–28, Berkeley, CA, USA, March 2004. USENIX Association.

[85] Frank Leymann and Dieter Roller. Business Process Management with FlowMark. In *Compcon Spring '94, Digest of Papers.*, pages 230–234, 1994.

[86] Christoph Liebig, Bianca Boesling, and Alejandro P. Buchmann. A Notification Service for Next-Generation IT Systems in Air Traffic Control. In *GI-Workshop: Multicast-Protokolle und Anwendungen*, Braunschweig, Germany, May 1999.

[87] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TAG: a Tiny AGgregation Service for Ad-hoc Sensor Networks. *5th Symposium on Operating Systems Design and Implementation*, 36(SI):131–146, December 2002.

[88] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. The Design of an Acquisitional Query Processor for Sensor Networks. In *ACM SIGMOD Intl. Conference on Management of Data*, SIGMOD'03, pages 491–502, New York, NY, USA, June 2003. ACM.

[89] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TinyDB: an Acquisitional Query Processing System for Sensor Networks. *ACM Trans. Database Systems*, 30(1):122–173, March 2005.

[90] Geoffrey Mainland, Greg Morrisett, and Matt Welsh. Flask: Staged Functional Programming for Sensor Networks. In *13th ACM SIGPLAN Intl. Conference on Functional Programming*, ICFP'08, pages 335–346, New York, NY, USA, September 2008. ACM.

[91] Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk, and John Anderson. Wireless Sensor Networks for Habitat Monitoring. In *1st ACM Int. Workshop on Wireless Sensor Networks and Applications*, WSNA'02, pages 88–97, New York, NY, USA, September 2002. ACM.

[92] Colette Maloney. Foreword. In *2nd. EEB Data Models Community Workshop*, page 6, Sophia Antipolis, France, October 2011.

[93] Kirk Martinez, Paritosh Padhy, Ahmed Elsaify, Gang Zou, A. Riddoch, Jane K. Hart, and Henry L. R. Ong. Deploying a Sensor Network in an Extreme Environment. In *IEEE Intl. Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing*, volume 1, pages 186–193, June 2006.

[94] Friedemann Mattern, editor. *Total vernetzt – Szenarien einer informatisierten Welt*. Xpert.press. Springer-Verlag, 2003.

[95] Dennis R. McCarthy and Umeshwar Dayal. The Architecture of an Active Data Base Management System. In James Clifford, Bruce G. Lindsay, and David Maier, editors, *Intl. Conference on Management of Data*, SIGMOD'89, pages 215–224, Portland, Oregon, May 1989. ACM Press.

[96] MEMSIC. Gateways' Datasheets. `www.memsic.com/wireless-sensor-networks/index.cfm`, July 2012. (last visited October 2014).

[97] MEMSIC. Mica2 Data Sheet. `www.memsic.com`, July 2012. (last visited February 2013).

[98] Mohammad M. Molla and Sheikh Iqbal Ahamed. A Survey of Middleware for Sensor Network and Challenges. In *Workshops at the Intl. Conference on Parallel Processing*, ICPPW'06, pages 223–228, Washington, DC, USA, August 2006. IEEE Computer Society.

[99] Gabriel Montenegro, Nandakishore Kushalnagar, Jonathan Hui, and David Culler. Transmission of IPv6 Packets over IEEE 802.15.4 Networks. RFC 4944 (Proposed Standard), September 2007. Updated by RFCs 6282, 6775.

[100] Luca Mottola and Gian Pietro Picco. Programming Wireless Sensor Networks: Fundamental Concepts and State of the Art. *ACM Computing Surveys*, 43(3):1–51, April 2011.

[101] Luca Mottola and Gian Pietro Picco. Middleware for Wireless Sensor Networks: an Outlook. *Journal of Internet Services and Applications*, 3(1):31–39, May 2012.

[102] René Müller, Gustavo Alonso, and Donald Kossmann. A Virtual Machine for Sensor Networks. In *2nd ACM SIGOPS European Conference on Computer Systems*, EuroSys'07, pages 145–158, New York, NY, USA, March 2007. ACM.

[103] Lama Nachman, Jonathan Huang, Junaith Shahabdeen, Robert Adler, and Ralph Kling. Imote2: Serious Computation at the Edge. In *Intl. Wireless Communications and Mobile Computing Conference*, IWCMC'08, pages 1118–1123, August 2008.

[104] Object Management Group, Inc. *Business Process Model and Notation (BPMN) Version 2.0*, 2011.

[105] US Dept. of Energy. Solar Decathlon. http://www.solardecathlon.org, 2009. (last visited October 2014).

[106] National Institute of Standards and Technology. RoboCup Rescue Arena Assembly Guide. `www.nist.gov/el/isd/upload/2011_Assembly_Guide.pdf`, July 2011. (last visited October 2014).

[107] Frederik Osterlind, Adam Dunkels, Joakim Eriksson, Niclas Finne, and Thiemo Voigt. Cross-Level Sensor Network Simulation with COOJA. In *31st IEEE Conference on Local Computer Networks*, LCN'06, pages 641–648, November 2006.

[108] PlatformX. Stargate Project Page. `platformx.sourceforge.net`, July 2005. (last visited October 2014).

[109] Joseph Polastre, Robert Szewczyk, and David Culler. Telos: Enabling Ultra-low Power Wireless Research. In *4th Int. Symposium on Information Processing in Sensor Networks*, IPSN'05, Piscataway, NJ, USA, April 2005. IEEE Press.

[110] Gregory J. Pottie and William J. Kaiser. Wireless Integrated Network Sensors. *Communications of the ACM*, 43(5):51–58, May 2000.

[111] Viktor K. Prasanna, James Reich, Amol Bakshi, and Daniel Larner. The Abstract Task Graph: A Methodology for Architecture-Independent Programming of Networked Sensor Systems. *Workshop on End-to-End, Sense-and-Respond Systems, Applications and Services*, pages 19–24, June 2005.

[112] Mirko Presser, Srdjan Krco, Tobias Kowatsch, Wolfgang Maass, Sebastian Lange, Francois Carrez, Bernard Hunt, Richard Egan, Jan Höller, Alessandro Bassi, Stephan Haller, and Gunter Woysch. *Inspiring the Internet of Things: The Internet of Things Comic Book*. Alexandra Institute, Aarhus, Denmark, October 2011.

[113] Red Hat, Inc. BPMN2 Modeler Web Site. `www.eclipse.org/bpmn2-modeler`, November 2013. (last visited October 2014).

[114] Kay Römer. Programming Paradigms and Middleware for Sensor Networks. In *GI/ITG Workshop on Sensor Networks*, pages 49–54, February 2004.

[115] Kay Römer, Christian Frank, Pedro J. Marrón, and Christian Becker. Generic Role Assignment for Wireless Sensor Networks. In *11th ACM SIGOPS European Workshop*, pages 7–12, Leuven, Belgium, September 2004.

[116] Luis Ruiz-Garcia, Pilar Barreiro, Jose Ignacio Robla, and Loredana Lunadei. Testing ZigBee Motes for Monitoring Refrigerated Vegetable Transportation Under Real Conditions. *Sensors*, 10(5):4968–4982, 2010.

[117] Nick Russell, Arthur H. M. ter Hofstede, Wil M. P. van der Aalst, and Nataliya Mulyar. Workflow Control-Flow Patterns: A Revised View. Report BPM-06-22, BPM Center, `bpmcenter.org`, 2006.

[118] Nick Russell, Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, and Petia Wohed. On the Suitability of UML 2.0 Activity Diagrams for Business Process Modelling. In *3rd Asia-Pacific Conference on Conceptual Modelling*, volume 53 of *APCCM'06*, pages 95–104, Darlinghurst, Australia, Australia, January 2006. Australian Computer Society, Inc.

[119] Christopher M. Sadler and Margaret Martonosi. Data Compression Algorithms for Energy-constrained Devices in Delay Tolerant Networks. In *4th Intl. Conference on Embedded Networked Sensor Systems*, SenSys'06, pages 265–278, New York, NY, USA, November 2006. ACM.

[120] Philipp M. Scholl, Kristof Van Laerhoven, Dawud Gordon, Markus Scholz, and Matthias Berning. jNode: a Sensor Network Platform that Supports Distributed Inertial Kinematic Monitoring. In *9th Int. Conference on Networked Sensing Systems*, INSS '12, pages 1–4, Antwerp, Belgium, June 2012.

[121] Alec Sharp and Patrick McDermott. *Workflow Modeling: Tools for Process Improvement and Application Development*. Artec House Publishers, 2001.

[122] Randall B. Smith. SPOTWorld and the Sun SPOT. In *6th Intl. Conference on Information Processing in Sensor Networks*, IPSN'07, pages 565–566, New York, NY, USA, April 2007. ACM.

[123] Patrik Spiess, Stamatis Karnouskos, Luciana Souza, Domnic Savio, Dominique Guinard, Vlad Trifa, Oliver Baecker, and Moritz Koehler. Reliable Execution of Business Processes on Dynamic Networks of Service-Enabled Devices. In *7th IEEE Intl. Conference on Industrial Informatics*, INDIN'09, pages 533–538, Cardiff, UK, June 2009.

[124] Mukundan Sridharan, Wenjie Zeng, William Leal, Xi Ju, Rajiv Ramnath, Hongwei Zhang, and Anish Arora. From Kansei to KanseiGenie: Architecture of Federated, Programmable Wireless Sensor Fabrics. In *7th Intl. Conference on Testbeds and Research Infrastructure for the Development of Networks and Communities*, TridentCom'11, pages 155–165. Springer, April 2011.

[125] John A. Stankovic. Research Challenges for Wireless Sensor Networks. *SIGBED Rev.*, 1(2):9–12, July 2004.

[126] Jan Steffan, Ludger Fiege, Mariano Cilia, and Alejandro P. Buchmann. Towards Multi-Purpose Wireless Sensor Networks. In *Intl. Conference on Sensor Networks*, SENET'05. IEEE Computer Society, August 2005.

[127] Martin Strohbach, Gerd Kortuem, and Hans Gellersen. Cooperative Artefacts - A Framework for Embedding Knowledge in Real World Objects. In *Smart Object Systems Workshop at UbiComp*, September 2005.

[128] Ryo Sugihara and Rajesh K. Gupta. Programming Models for Sensor Networks: A Survey. *ACM Trans. Sen. Netw.*, 4(2):8:1–8:29, April 2008.

[129] Michael Sweet. Mini-XML Project. www.msweet.org, December 2011. (last visited October 2014).

[130] Robert Szewczyk, Joseph Polastre, Alan Mainwaring, and David Culler. Lessons from a Sensor Network Expedition. In Holger Karl, Andreas Willig, and Adam Wolisz, editors, *1st European Workshop on Wireless Sensor Networks*, volume 2920 of *EWSN'04*, pages 307–322. Springer, January 2004.

[131] David Tennenhouse. Proactive Computing. *Communications of the ACM*, 43(5):43–50, May 2000.

[132] Stefano Tranquillini, Patrik Spieß, Florian Daniel, Stamatis Karnouskos, Fabio Casati, Nina Oertel, Luca Mottola, Felix Jonathan Oppermann, Gian Pietro Picco, Kay Römer, and Thiemo Voigt. Process-based Design and Integration of Wireless Sensor Network Applications. In *10th Intl. Conference on Business Process Management*, BPM'12, pages 134–149, Berlin, Heidelberg, September 2012. Springer-Verlag.

[133] Nicolas Tsiftes and Adam Dunkels. A Database in Every Sensor. In *9th ACM Conference on Embedded Networked Sensor Systems*, SenSys'11, pages 316–332, New York, NY, USA, November 2011. ACM.

[134] U.S. Department of Transportation. *Instrument Flying Handbook*. Federal Aviation Administration, 2012.

[135] USB Implementers Forum. *Universal Serial Bus Specification Revision 2.0*, April 2000.

[136] Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, Bartek Kiepuszewski, and Alistair P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, July 2003.

[137] Pascal A. Vicaire, Zhiheng Xie, Enamul Hoque, and John A. Stankovic. Physicalnet: A Generic Framework for Managing and Programming Across Pervasive Computing Networks. In *16th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS'10, pages 269–278, April 2010.

[138] Mark Weiser. The Computer for the 21st Century. *SIGMOBILE Mobile Computing Communications Review*, 3(3):3–11, July 1999.

[139] Matt Welsh and Geoff Mainland. Programming Sensor Networks Using Abstract Regions. In *1st USENIX/ACM Symposium on Networked Systems Design and Implementation*, NSDI'04, pages 29–42, Berkeley, CA, USA, March 2004. USENIX Association.

[140] Chen Wenjie, Chen Lifeng, Chen Zhanglong, and Tu Shiliang. A Realtime Dynamic Traffic Control System Based on Wireless Sensor Network. In *Intl. Conference Workshops on Parallel Processing*, ICPP'05 Workshops, pages 258–264, June 2005.

[141] Geoffrey Werner-Allen, Konrad Lorincz, Matt Welsh, Omar Marcillo, Jeff Johnson, Mario Ruiz, and Jonathan Lees. Deploying a Wireless Sensor Network on an Active Volcano. *IEEE Internet Computing*, 10(2):18–25, 2006.

[142] Geoffrey Werner-Allen, Patrick Swieskowski, and Matt Welsh. MoteLab: a Wireless Sensor Network Testbed. In *4th Intl. Symposium on Information Processing in Sensor Networks*, IPSN'05, Piscataway, NJ, USA, April 2005. IEEE Press.

[143] Kamin Whitehouse, Cory Sharp, Eric Brewer, and David E. Culler. Hood: A Neighborhood Abstraction for Sensor Networks. In *2nd Intl. Conference on Mobile Systems, Applications and Services*, MobiSys'04, pages 99–110, Boston, MA, USA, June 2004. ACM Press.

[144] Petia Wohed, Wil M. P. van der Aalst, Marlon Dumas, Arthur H. M. ter Hofstede, and Nick Russell. On the Suitability of BPMN for Business Process Modelling. In *Business Process Management*, BPM'06, pages 161–176, Vienna, Austria, September 2006.

[145] Ning Xu, Sumit Rangwala, Krishna Kant Chintalapudi, Deepak Ganesan, Alan Broad, Ramesh Govindan, and Deborah Estrin. A Wireless Sensor Network for Structural Monitoring. In *2nd Intl. Conference on Embedded Networked Sensor Systems*, SenSys'04, pages 13–24, New York, NY, USA, November 2004. ACM Press.

[146] Yong Yao and Johannes Gehrke. The Cougar Approach to In-Network Query Processing in Sensor Networks. *SIGMOD Record*, 31(3):9–18, September 2002.

[147] Yong Yao and Johannes Gehrke. Query Processing in Sensor Networks. In *Biennial Conference on Innovative Data Systems Research*, CIDR '03, January 2003.

[148] Xiaoliang Zhao, Tao Qian, Gang Mei, Chiman Kwan, Regan Zane, Christi Walsh, Thurein Paing, and Zoya Popovic. Active Health Monitoring of an Aircraft Wing with an Embedded Piezoelectric Sensor/Actuator Network: II. Wireless Approaches. *Smart Materials and Structures*, 16(4), August 2007.

A The ukuFlow Bytecode

We present the primary blocks of the ukuFlow bytecode, aligned at 16 bits, as is the case in the MSP430 line of microcontrollers. Capitalized elements occupy 1 byte, and have predefined values, while elements surrounded by angular brackets are of variable width, and are defined subsequently.

The description includes the bytecode specification for the general workflow data, events, script tasks, gateways and event operators.

General Workflow Data

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
workflow id								# wf. elements							
# of scopes								min. # of instances required							
max. # of instances required								looping info							
<workflow elements> ...															
<scope definitions> ...															

Events

Start event

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
wf. element id								START_EVENT							
id of following wf. element															

End event

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
wf. element id								END_EVENT							

Script Task

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
wf. element id								EXECUTE_TASK							
id of following wf. element								# of statements							
<statements> ...															

The statements take one of the following formats:

Computation Statement

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
COMPUTATION_STATEMENT								id of variable that receives value							
expression length															
<expression>															

Local Function Statement

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
LOCAL_FUNCTION_STATEMENT								id of variable that receives value							
command length								# of parameters							
<command string>															
<list of parameters>															

Scoped Function Statement

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
SCOPED_FUNCTION_STATEMENT								id of scope to use							
command length								# of parameters							
<command string>															
<list of parameters>															

A parameter has one of the following structures:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
REPOSITORY_VALUE								id of repository entry							

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
UINT8_VALUE								value							

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
UINT16_VALUE								value’s lsb							
value’s msb															

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
STRING_VALUE								string length							
<string>															

Gateways

Fork Gateway

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
wf. element id								FORK_GATEWAY							
# of outgoing seq. flows															
<list of workflow ids to which to fork>															

Join Gateway

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
wf. element id								JOIN_GATEWAY							
id of following wf. element								# of incoming seq. flows							
<list of workflow ids from where to expect tokens>															

Inclusive Decision Gateway

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
wf. element id								INCLUSIVE_DECISION_GATEWAY							
# of outgoing seq. flows															
<list of outgoing sequence flows>															

These outgoing sequence flows have the following structure:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
id of following wf. element								condition expression length							
<condition expression>															

In the case of there is a default sequence flow, this must be the last one inside of a gateway, and must contain no expression, as follows:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
id of following wf. element								0							

Inclusive Join Gateway

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
wf. element id								INCLUSIVE_JOIN_GATEWAY							
id of following wf. element								# of incoming seq. flows							
<list of workflow ids from where to expect tokens>															

Exclusive Decision Gateway

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
wf. element id								EXCLUSIVE_DECISION_GATEWAY							
# of outgoing seq. flows															
<list of outgoing sequence flows>															

Exclusive Merge Gateway

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
wf. element id								EXCLUSIVE_MERGE_GATEWAY							
id of following wf. element								# of incoming seq. flows							
<list of workflow ids from where to expect tokens>															

Event-based Exclusive Decision Gateway

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
wf. element id								EVENT_BASED_EXCLUSIVE DECISION_GATEWAY							
# of outgoing seq. flows															
<list of outgoing sequence flows>															

These outgoing sequence flows have the following structure:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
id of following wf. element								event operator expression len.							
<event operator expression>															

Event Generators

All event generators follow the syntax below:

0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15
event generator type	event operator id
channel id	source
scope id	
<event generator-specific parameters>	

We present two event generators next:

0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15
ABSOLUTE_EG	event operator id
channel id	source
scope id	absolute node time, in seconds (4 bytes)
absolute node time, in seconds (4 bytes)	

0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15
PATTERN_EG	event operator id
channel id	source
scope id	# of repetitions
period length	
pattern length	<actual pattern>

Event Filters

Simple Event Filter:

0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15
SIMPLE_EF	event operator id
channel id	# of expressions
<list of expressions>	

where the expressions follow the format:

0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15
expression length	<expression>

Composite Event Filters

All event composers follow the syntax below:

0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15
event composer type	event operator id
channel id	window size’s lsb
window size’s msb	

An example event composer of type `COUNT_EC`:

0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15
COUNT_EC	event operator id
channel id	window size
(in seconds)	

B Detailed Event Composition Performance

The following figure expands on the data presented in the evaluation of the event composition operator (Section 6.9). Each plot corresponds to a different network scale (in steps of 8 nodes); boxes represent the amount of events used for the composition in the respective iteration, sorted in descending order.

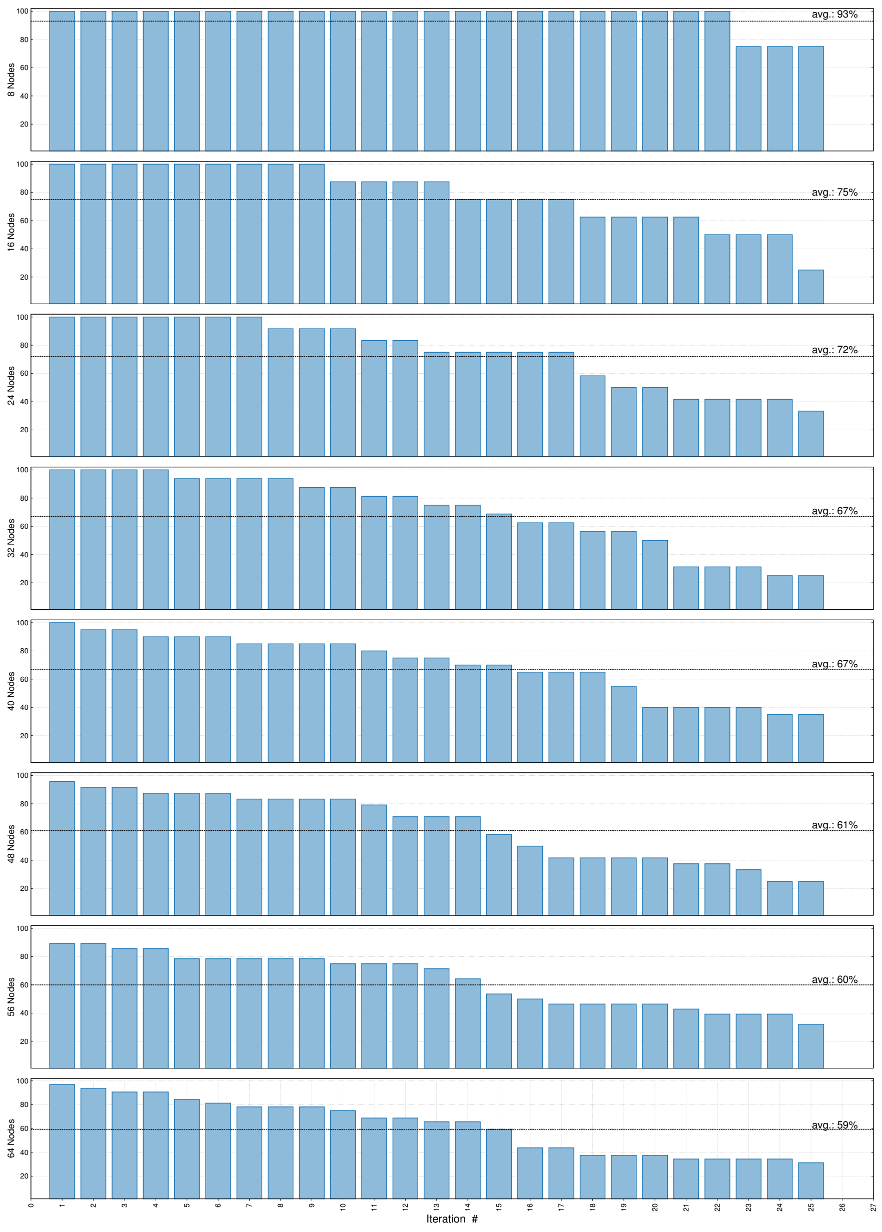


Figure B.1.: Evaluation of event composition performance, measured with networks of 8, 16, 32, 40, 48, 56 and 64 nodes.

In the figure it can be observed that the average amount of events composed decreased with the network scale, from 93% for 8 nodes, down to 59% with 64 nodes. Also, for all network scales, the range of values for the percentage of events composed ranged from 100% to 0% (the lowest value only occurred once).