Politecnico di Milano
Dipartimento di Elettronica e Informazione
Dottorato di Ricerca in Ingegneria dell'Informazione

Programming Wireless Sensor Networks: From Physical to Logical Neighborhoods

Ph.D. Dissertation of:
Luca Mottola

Advisor:
Prof. Gian Pietro Picco
Tutor:
Prof. Letizia Tanca
Supervisor of the Doctoral Program:
Prof. Patrizio Colaneri

2008 - XX

To my family
Alla mia famiglia

Abstract

Recent technological advances and progresses in hardware miniaturization have made wireless sensor networks (WSNs) a viable solution to gather information from the environment and make it available to scientists [1], or to use sensed data to decide on actions to take on the environment [2]. A recent report from market research firm ONWorld [3] indicates that the market for WSNs is expected to grow tenfold by 2011. The same report identifies *ease of programming* as the major barrier to the adoption of WSNs. To address this issue, this thesis presents a set of WSN programming abstractions whose aim is to simplify application development in a range of settings, particularly, in *sense-and-react* scenarios where the system is used to close the control loop by taking actions on the environment.

The first part of the thesis describes the conceptual path to the above objective. We present a detailed taxonomy of existing WSN programming solutions, and cast available approaches in our classification. In doing so, we realize how most WSN programming frameworks available to date fall short of expectation in dealing with sense-and-react scenarios. In these settings, programmers are indeed to cope with several challenges stemming from *node heterogeneity*, *changing requirements*, and *group-based interactions*, that are only partially addressed in existing approaches. The programming solutions presented in this thesis address each of these challenges by providing solutions to program *individual devices*, *group of nodes*, and the *network as a whole* (the latter class of solutions being usually termed "macroprogramming"). In addition, to take into account the characteristics of the WSN hardware, we *co-design* the programming abstractions with the underlying, distributed algorithms and protocols required. This enables unprecedented degrees of interplay between the programming and the system layer, which ultimately yield better performance than tackling the two problems separately, as done traditionally.

In the second part of the work, we look at how to program *individual nodes*. Firstly, we investigate how to reconfigure the single node behavior depending on unanticipated situations. To address these needs, we present the RUNES [4,5] and FIGARO [6] programming models. Both are component-based approaches allowing programmers to precisely identify the functionality that is to be reconfigured. In addition, FIGARO automatically handles the whole reconfiguration process. In parallel, we study

how to blend reactive and proactive interactions in a single programming framework, as needed in sense-and-react scenarios. Our answer to this issue is the TeenyLIME middleware [7, 8], a programming model in which we revisit the tuple space paradigm to account for the distinctive traits of WSN applications.

The third part of the thesis investigates programming *groups* or *subsets* of nodes. We tackle this problem with the Logical Neighborhoods [9,10] abstraction. Differently from the physical neighborhood of a node—implicitly defined by the location of nodes and their radio ranges—Logical Neighborhoods empower programmers with a higher-level notion of proximity determined by application information. Using a declarative language we devised, programmers identify the nodes part of a Logical Neighborhood based on application-level characteristics, and interact with them using simple message-passing primitives. Instead of the nodes within radio range, the message recipients are now the nodes matching a given neighborhood specification, regardless of their physical position. This way, we provide a basic building block to build more sophisticated functionality as well as higher-level abstractions on top.

In the fourth part of this thesis, we explore the coupling of Logical Neighborhoods with mechanisms other than message passing. We illustrate how Logical Neighborhoods provide a natural complement to the FIGARO component model, by offering support to direct code updates towards a specific part of the system [6]. Next, we describe Virtual Nodes [11]: a programming abstraction whereby application-defined groups of nodes can be abstracted into a single, logical one. Finally, we describe how Logical Neighborhoods can be embedded within an existing macroprogramming language [12, 13]. This allows to precisely identify the specific portions of the system that need to be involved in achieving a given goal.

To substantiate our claims regarding the superior effectiveness of our co-design approach, we always analyze our solutions from two complementary perspectives. On one hand, we quantify the *programming effort* in developing non-trivial reference applications both using our solutions and with mainstream programming tools. On the other hand, we study the *system performance* w.r.t. metrics such as network overhead and system lifetime compared to traditional communication schemes.

To conclude, we maintain that future work in programming WSNs must focus on providing stronger semantics and guarantees, as well as on verifying the correctness of the resulting implementations.

Riassunto

I recenti sviluppi tecnologici e i progressi nella miniaturizzazione hanno reso possibile utilizzare le reti di sensori wireless (Wireless Sensor Networks - WSNs) per ricavare informazioni dall'ambiente e renderla disponibile agli scienziati del dominio [1], o per utilizzare i dati sensoriali per guidare azioni sull'ambiente [2]. Una recente indagine dell'agenzia ONWorld [3] prevede che il mercato delle WSNs cresca di un ordine di grandezza entro il 2011. La stessa indagine identifica la facilità di programmazione come la maggior barriera all'adozione di WSNs. Per affrontare questa problematica, questa tesi presenta un insieme di astrazioni di programmazione per WSNs il cui scopo è di semplificare lo sviluppo di applicazioni in scenari diversi, in particolare, in scenari di tipo *sense-and-react* dove il sistema è usato per chiudere l'anello di controllo effettuando azioni sull'ambiente.

La prima parte di questa tesi descrive il cammino concettuale per raggiungere tale scopo. Presentiamo una tassonomia dettagliata delle soluzioni esistenti e classifichiamo gli approcci disponibili. Nel fare ciò, realizziamo come la maggior parte dei sistemi di programmazione oggi disponibili non sono applicabili a scenari sense-and-react. In tali situazioni, il programmatore deve affrontare diverse problematiche originanti dall'*eterogeneità* dei nodi, da *requisiti mutevoli*, e da *interazioni basate su gruppi* di nodi che sono solo parzialmente supportate dagli approcci esistenti. Le soluzioni presentate in questa tesi affrontano queste problematiche fornendo soluzioni per la programmazione di *nodi individuali*, di *gruppi di nodi*, e della *intera rete* (l'ultima classe di soluzioni normalmente chiamata "macroprogramming"). In aggiunta, in luce delle caratteristiche dell'hardware dei nodi WSNs, adottiamo un approccio di *co-design* nel progetto delle astrazioni di programmazione e dei protocolli distribuiti a supporto. Ciò permette una stretta collaborazione tra gli strati di programmazione e di sistema, producendo migliori performance che affrontando i problemi separatamente.

Nella seconda parte del lavoro ci concentriamo sulla programmazione dei *singoli nodi*. Dapprima investighiamo come riconfigurare il comportamento del nodo a fronte di mutevoli requisiti. Per rispondere a questo bisogno, presentiamo i modelli di programmazione RUNES [4, 5] e FiGaRo [6]. Entrambi sono approcci a componenti che permettono al programmatore di identificare con precisione la funzionalità da riconfigurare. In aggiunta, FiGaRo gestisce automaticamente l'intero processo di riconfigurazione. In

parallelo, studiamo come fondere interazioni reattive e proattive in un solo modello di programmazione, come richiesto in scenari sense-and-react. La nostra risposta a questo problema è il middleware TeenyLime, un modello di programmazione dove rivisitiamo il paradigma a spazi di tuple per tenere in considerazioni i tratti distintivi delle applicazioni WSNs.

La terza parte della tesi investiga la programmazione di *gruppi* o *sottoinsiemi* di nodi. Affrontiamo questo problema con l'astrazione Logical Neighborhoods [9, 10]. Diversamente dalla neighborhood fisica di un nodo—implicitamente definita dalla posizione dei nodi e dal loro raggio di comunicazione—Logical Neighborhoods fornisce una nozione di prossimità di più alto livello determinata da informazioni applicative. I programmatori identificano i nodi in una neighborhood logica in base a caratteristiche applicative ed interagiscono con essi usando un paradigma a scambio di messaggi. I destinatari dei messaggi sono ora i nodi rispondenti ad una definizione di neighborhood indipendentemente dalla loro posizione fisica. Forniamo quindi un blocchetto fondamentale per la costruzione di funzionalità più sofisticate e di astrazioni di più alto livello.

Nella quarta parte della tesi esploriamo l'uso di Logical Neighborhoods con meccanismi diversi dallo scambio di messaggi. Illustriamo come Logical Neighborhoods fornisce un complemento naturale al modello a componenti FiGaRo offrendo supporto per dirigere gli aggiornamenti di codice verso una specifica parte del sistema [6]. Successivamente, descriviamo Virtual Nodes [11]: un'astrazione di programmazione con cui gruppi di nodi identificati dall'applicazione possono essere incapsulati in un unico nodo logico. Da ultimo, descriviamo come Logical Neighborhoods può essere incorporato in un linguaggio di macroprogramming esistente [12, 13]. Ciò permette di identificare con precisione le porzioni di sistema che devono essere coinvolte nel raggiungimento di uno specifico obiettivo.

Per affermare la superiore efficacia del nostro approccio co-design, analizziamo le nostre soluzioni sempre da due prospettive complementari. Da un lato, quantifichiamo lo *sforzo di programmazione* nello sviluppo di applicazioni di riferimento sia usando le nostre soluzioni sia comuni strumenti di programmazione. Dall'altro lato, studiamo la *performance del sistema* rispetto a metriche quali il traffico di rete e il tempo di vita del sistema confrontandole con meccanismi di comunicazione tradizionali.

Per concludere, argomentiamo come ulteriori sforzi di ricerca sono richiesti nel fornire garanzie e semantiche più forti e nella verifica della correttezza delle implementazioni risultanti.

Acknowledgments

It is anything but easy to thank all the people who contributed to this thesis and, most importantly, to my personal and professional development during my Ph.D. studies.

More than anyone else, however, my Advisor Gian Pietro Picco has been the one who made me think research was what I really wanted to do. His enthusiasm has always been contagious, regardless of whether we were at a research meeting, on some long-haul flight, or in front of the vending machine. I learned a lot, not only on a professional plane, and I owe to him most of what I know about doing research.

Other people I met over these years also contributed in different ways. Amy L. Murphy, Cecilia Mascolo, and Giampaolo Cugola have been great collaborators as well as good friends. Carlo Ghezzi and Luciano Baresi have always been supportive and proved to care about me well beyond professional life. Viktor Prasanna allowed me to be exposed to a stimulating research environment during my stay at the University of Southern California, Los Angeles (CA, USA).

Besides the above, my colleagues Paolo, Davide, Animesh, and Matteo C. have also been good friends and wonderful people to work with. By the same token, I am grateful to friends inside and outside the professional environment, people I simply shared good moments and laughter with: Alessandro B., Sabrina, Irene, Roberta, Anna M., Francesco, Alessandro L., and Daniele. A special thank goes to Pincy, who first whispered the word "research".

More than anyone else, however, my family has been fundamental. Even when we had to face problems way more important than anything I could write in this thesis, I never missed your unconditional support to my professional development. Eventually, I hope to have a chance to return what you gave me.

Thank you.

Luca

Ringraziamenti

Non è facile ringraziare tutte le persone che hanno contribuito a questa tesi e, soprattutto, al mio sviluppo personale e professionale durante i miei studi di dottorato.

Ciò nonostante, il mio Advisor Gian Pietro Picco è stato colui che, più di chiunque altro, ha sviluppato in me la convinzione di voler fare ricerca. Il suo entusiasmo è sempre stato contagioso, sia che fossimo ad una riunione, su qualche volo intercontinentale, o davanti alla macchinetta del caffè. Ho imparato molto, non solo sul piano professionale, e gli devo gran parte di ciò che so sul fare ricerca.

Altre persone hanno contribuito in diverse maniere. Amy L. Murphy, Cecilia Mascolo, e Giampaolo Cugola sono stati collaboratori eccezionali e buoni amici. Luciano Baresi e Carlo Ghezzi non hanno mai fatto mancare il loro supporto e hanno dimostrato di tenere a me ben al di là dei rapporti professionali. Viktor Prasanna mi ha permesso di essere esposto ad un ambiente di ricerca stimolante durante la mia permanenza presso University of Southern California, Los Angeles (CA, USA).

Oltre alle persone di cui sopra, i miei colleghi Paolo, Davide, Animesh, e Matteo C. sono stati buoni amici e splendide persone con cui lavorare. Alla stessa maniera, sono grato agli amici dentro e fuori l'ambiente professionale, persone con cui ho semplicemente condiviso risate e bei momenti: Alessandro B., Sabrina, Irene, Roberta, Anna M., Francesco, Alessandro L., e Daniele. Un grazie speciale è per il Pincy, con cui per primo ho parlato di "ricerca".

Più di chiunque altro però, la mia famiglia è stata fondamentale. Anche quando abbiamo dovuto affrontare problemi ben più importanti di qualunque cosa possa scrivere in questa tesi, non avete mai fatto mancare il vostro supporto incondizionato al mio percorso professionale. Spero, prima o poi, di avere la possibilità di ritornare ciò che mi avete dato.

Grazie.

Luca

Contents

List of Figures

1. Introduction

Wireless Sensor Networks (WSNs) are distributed systems composed of a high number of tiny devices, each equipped with a low-power processing unit, a wireless communication interface, and sensing or acting functionality. By embedding processing and communication within the physical world, WSNs can be used as a tool to bridge real and virtual environments. Harmonizing these two dimensions, however, can be achieved along different directions, as illustrated in Figure 1.1.

WSNs can be used to gather information from the environment and make it available to *scientists* for studying the corresponding dynamics. Seminal research in WSNs mostly tackled this type of *sense-only* scenarios. Initial deployments indeed concentrated on applications such as environmental monitoring [22], wildlife tracking [23], and supervised agriculture [24]. In these scenarios, WSNs unveiled their potential to academic and industrial researchers, though they remained far from the actual domain-experts, i.e., their intended end-users. Early players in the WSN arena immediately acknowledged ease of programming as the main obstacle to a widespread adoption of this technology [25].

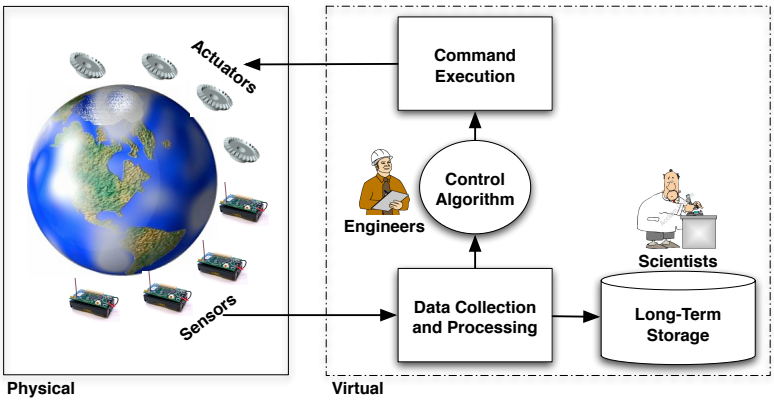


Figure 1.1.: Bridging virtual and physical environments using WSNs.

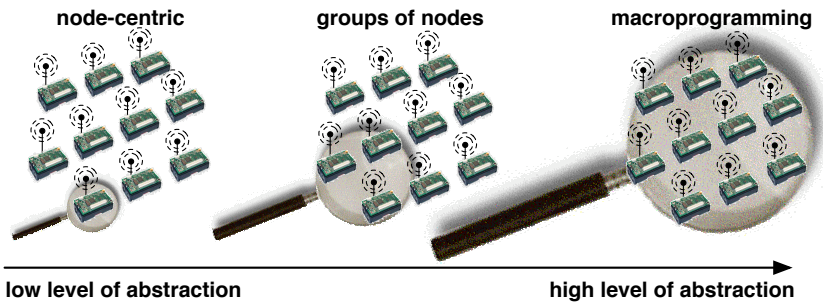


Figure 1.2.: WSN programming approaches.

Besides the above scenarios, sensed data can be used by *control engineers* to decide on actions to take on the environment [2]. Similar *sense-and-react* scenarios hold the potential for enabling unattended control systems that blend transparently with the physical world. The ability to react to external stimuli indeed enables novel applications such as home automation [26], building monitoring and control [27], and emergency response [4]. However, driving actuation based on sensed data further complicates the programmer's life, due to decentralization of the application logic. Therefore, providing the appropriate *programming abstractions* to domain-experts acquires even greater importance.

The **first part** of this thesis describes the conceptual path to the above objective. As intuitively illustrated in Figure 1.2, existing programming solutions for WSNs have been hitherto classified depending on the level of abstraction provided. On one end of the spectrum, *node-centric* programming frameworks focus on individual nodes. Differently, *macroprogramming* approaches give developers the ability to treat the WSN as a whole, and express the application processing regardless of the single devices. A few proposal also exist to find a trade-off between these two extremes, by providing abstractions to program *groups* of nodes. As discussed in Chapter 2, we maintain that this classification fails in capturing the essence of existing WSN programming abstractions. Therefore, in the same chapter we present a detailed taxonomy of existing WSN programming solutions, and cast available approaches in our classification. In doing so, it is easy to realize how most of the current state of the art is devised for sense-only scenarios. Although this kind of programming solutions served as a stepping stone for WSNs, they fall short of expectation in dealing with sense-and-react scenarios, being essentially ill-suited to express the corre-

sponding requirements. Sense-and-react scenarios indeed exhibit unique characteristics and traits, which require programmers to cope with several challenges:

C1: *Heterogeneous* devices must *coordinate* so that each of them, with its own capabilities and features, contributes to accomplish a higher-level goal. Coordination usually occurs in the form of localized interactions, to keep processing close to where actuation is to be performed, thus saving on latency and resource consumption.

C2: The application demands are no longer carved in stone. When the high-level application goals change, the system must adapt its behavior accordingly. Therefore, programmers are to deal with a *fluid set of requirements* that dynamically evolve depending on a priori unknown conditions.

C3: Multiple tasks must run concurrently, each having a different set of sensors as input, and controlling disparate classes of actuators. Consequently, different groups of devices must interact among themselves. The focus is therefore on *subsets* of nodes, as opposed to individual devices or the whole network.

As we describe in Chapter 3, in this thesis we aim at providing flexible programming abstractions to address the three challenges above. In doing so, we explore the whole spectrum of WSN programming approaches, by providing solutions to program individual devices, group of nodes, and the sensor networks as a whole. At the same time, this is only our minimal goal. Our solutions may indeed prove useless if the underlying implementations do not account for the very characteristics of WSN hardware. Therefore, in this work we *co-design* the programming abstractions with the underlying, distributed support required. This enables unprecedented degrees of interplay between the language constructs and the mechanisms providing the corresponding semantics, which ultimately yield better performance than tackling the two problems separately, as done traditionally.

Figure 1.3 graphically illustrates the thesis organization, and how our contributions meet the three challenges above. In the **second part** of this work, we start addressing some of those challenges by looking at how to program *individual nodes*. Facing a changing set of requirements (C2) requires programmers to reconfigure the single node behavior depending on unanticipated situations. Mainstream solutions for single node programming

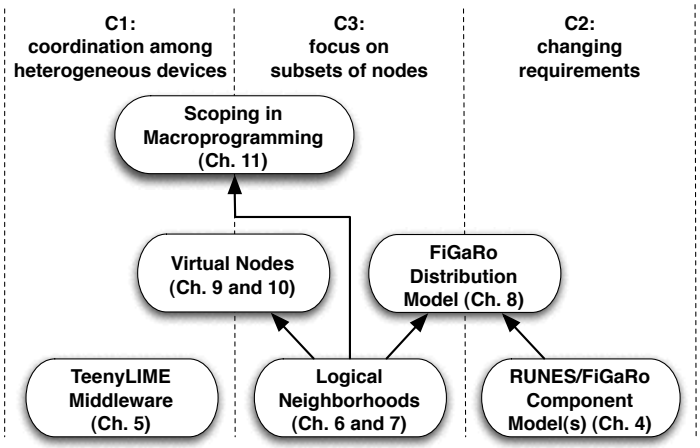


Figure 1.3.: Thesis organization.

(e.g., [28]), however, provide little or no support to identify what functionality need to be reconfigured, and how to carry out the reconfiguration process. In Chapter 4, we report on our work on two component models explicitly devised to support fine-grained software reconfiguration in WSNs. We first describe the RUNES programming model [4,5], a component-based solution aiming at an extreme form of heterogeneity where not only WSN devices are involved in the application processing, but also more powerful nodes such as PDAs. RUNES allows us to identify the minimal set of notions and operations to enable reconfiguration of single functionality. Based on our experience with RUNES, we identify a further set of open issues, this time specific to the WSN domain. We address them with the FiGaRo component model [6]. Compared to RUNES, FiGaRo provides an enriched programming model where component dependencies and versions become first-class citizens and, differently from RUNES, it automatically handles the whole reconfiguration process.

To achieve coordination among heterogeneous devices (C1), programmers must be empowered with appropriate abstractions enabling both reactive and proactive interactions. The former are needed to take actions based on external conditions, whereas the latter are required to gather the measures of interests (e.g., a sensor reading) and drive actuation. Unfortunately, existing programming frameworks only provide basic communication facilities, usually based on message-passing. We instead provide both

types of language constructs in the TeenyLIME middleware [7,8], described in Chapter 5. TeenyLIME considerably raises the level of abstraction by giving programmers the high-level abstraction of a tuple space [29]. Nonetheless, this cannot be re-applied as is in WSNs. Therefore, we revisit the tuple space paradigm to account for the distinctive traits of WSN applications, and augment the programming model with sensor-specific features to further simplify the programming activity.

The **third part** of the thesis departs from programming individual devices by providing a foundation to deal with *subsets of nodes* (C3). Key to this goal is how to identify the relevant subsets, and how to describe simple interactions among them. Still, directly applicable solutions are mostly missing in the current state of the art. We tackle this problem by introducing the notion of Logical Neighborhood [9,10], described in Chapter 6. Differently from the physical neighborhood of a node—implicitly defined by the location of nodes and their radio ranges—Logical Neighborhoods empower programmers with a higher-level notion of proximity determined by application information. Using a declarative language we devised, programmers identify the nodes part of a Logical Neighborhood based on application-level characteristics. For instance, a node controlling a fan in a tunnel can define a logical neighborhood including all smoke sensors on a given lane that are currently reading a value above a safety threshold, and query them to find out about the current situation.

In principle, the notion of scoping provided by Logical Neighborhoods is orthogonal w.r.t. the interactions among nodes in a specific subset. For instance, one could couple Logical Neighborhoods with a query-response paradigm by directing interests towards a specific part of the system. Alternatively, one could combine our solution with a tuple space paradigm by enabling tuple sharing only among nodes in a Logical Neighborhood. To avoid restricting the spectrum of possible usages, programmers leverage off Logical Neighborhoods using a broadcast-based, message passing API—still described in Chapter 6—which mimics the traditional communication facilities of common WSN programming frameworks. The message recipients, however, are the nodes matching a given neighborhood definition, instead of those within radio range. Message delivery to nodes in a logical neighborhood occurs thanks to a dedicated routing protocol, also illustrated in Chapter 7, that we explicitly devised for Logical Neighborhoods. Existing solutions were indeed ill-suited to support the corresponding communication patterns. This way, we provide a full-fledged, basic building

block atop which one can build more sophisticated functionality as well as higher-level abstractions.

The above claim is substantiated in the **fourth part** of this thesis, where we explore the coupling of Logical Neighborhoods with mechanisms other than message passing. Reconfiguring the system behavior to address mutable requirements (C2) not only entails the ability to reconfigure single functionality on individual devices, but also requires to deliver the corresponding implementations only to the interested nodes (C3). In this respect, Chapter 8 illustrates how Logical Neighborhoods provide a natural complement to the FiGaRo component model, by offering support to direct code updates towards a specific part of the system [6]. Indeed, as the definition of Logical Neighborhoods is entirely up to the programmer, subsets of nodes can be identified also depending on their current software configuration, e.g., to target all devices running component `"ABC"` with version less than 3.

Despite the flexibility of Logical Neighborhoods, complex interactions among heterogeneous devices (C1) are still difficult to express using simple message-passing. To better support programmers in addressing this challenge, we leveraged off Logical Neighborhoods to build higher-level language constructs. In Chapter 9, we describe Virtual Nodes [11]: a programming abstraction whereby application-defined groups of nodes (C3) can be abstracted into a single, logical one. Spanning both ends of the control loop, virtual nodes take the form of virtual sensors or virtual actuators. The former abstract the data sensed by real sensors into the reading of a single, fictitious node; whereas the latter provide a single handle to control a distributed set of actuators. Although the distributed support for virtual actuators is readily implemented using the Logical Neighborhoods communication layer illustrated in Chapter 7, existing routing schemes turned out to be highly inefficient for implementing the semantics required by virtual sensors. Based on the demands posed by the corresponding language constructs, we therefore devised a dedicated routing solution [30], described in Chapter 10, that also enjoys wider applicability beyond our own programming framework.

After addressing individual node programming, and investigated solutions geared towards group of devices, the missing tile is therefore to look at the WSN as a whole. Unfortunately, *macroprogramming* approaches usually assume homogeneous scenarios where a single, system-wide task is to be accomplished. Hence, existing solutions are unable to express in-

teractions among heterogeneous devices limited to specific portions of the system (C1, C3). To address this issue, in Chapter 11 we describe how a notion of scoping can be embedded within an existing macroprogramming language [12, 13], and how this affects the corresponding compilation process [31]. This allows to identify the specific portions of the system that need to be involved in achieving a given goal, while retaining the high level of abstraction usually provided by macroprogramming.

To substantiate our claims regarding the superior effectiveness of our co-design approach, we always analyze our solutions from two complementary perspectives. On one hand, we quantify the *programming effort* in developing non-trivial reference applications both using our solutions and with mainstream programming tools. This gives a measure of the generality, flexibility, and easy-of-use of the abstractions we propose. On the other hand, we study the *system performance* w.r.t. metrics such as network overhead and system lifetime compared to traditional communication schemes. This way, we evaluate the advantages brought by our integrated approach to the final system performance.

After addressing the three challenges above, we maintain that future work in programming WSNs must focus on providing stronger semantics and guarantees, as well as on verifying the correctness of the resulting implementations. Our thoughts in this regard are illustrated in Chapter 12, which concludes this thesis.

Part I.

Programming Wireless Sensor Networks

2. Background

Ease of programming has long been recognized as a major hurdle to the adoption of WSN technology. In response to this need, several programming solutions have been hitherto developed. A well-established characterization of the available approaches is, however, largely missing. As a result, researchers are unable to orient themselves in this diverse field, and developers struggle in identifying the solutions most appropriate to their application requirements.

To address this issue, in this chapter we describe a taxonomy of WSN programming abstractions based on multiple characterizing dimensions. In doing so, our objective is twofold: i) we define a framework to put in context the contribution of this thesis, and ii) we aim at providing the WSN community at large with a foundation to classify, compare, and evaluate existing and future approaches.

2.1. Introduction

Leveraging off WSNs to bridge the gap between virtual and real environments, as outlined in Chapter 1, requires domain experts to develop WSN applications without being knowledgeable in embedded system programming. In view of these needs, the research community put increasing effort in devising programming solutions for WSNs. To date, however, none of the proposed approaches emerged as a clear winner in the WSN programming arena. Indeed, WSNs applications are still mostly developed at a very low level of abstraction, essentially atop the bare operating system. As a result, WSN software is typically unreliable, inefficient, and almost non-reusable. This greatly complicates the achievement of the above vision.

The issues causing the aforementioned situation are essentially twofold:

- WSNs are a truly interdisciplinary field, with applications ranging

from environmental monitoring in remote locations (e.g., [22]) to control of emergency situations in road tunnels (e.g., [4]). These typically exhibit the most disparate requirements, thus preventing the use of "one size fits all" solutions.

- A systematic characterization of the existing solutions in the field is largely missing. As a result, researchers struggle in comparing different approaches on a common ground. Likewise, the end-users are unable to relate the characteristics of a given programming approach to their application requirements.

To address the above issues, this chapter describes a classification of existing WSN programming abstractions based on multiple dimensions. To organize our effort, in Chapter 2.2 we define a conceptual architecture we use as a term of reference throughout the rest of the chapter. Chapter 2.3 examines paradigmatic sensor network applications, highlighting the requirements developers must meet when choosing the most appropriate programming platform. This serves to better investigate the characteristics of existing WSN applications at the origin of the first issue above. Chapter 2.4 gives a bird's eye view on our taxonomy, highlighting its overall organization and the two major directions we follow for classification. The first direction looks at the nature of the language constructs provided to programmers. In this respect, the different dimensions for classification we identify are discussed in Section 2.5. Differently, the second, complementary line of investigation considers the architectural aspects tied to a given programming framework. The corresponding dimensions for classification are described in Section 2.6. In both cases, for each characterizing dimension we intertwine the discussion with a description of distinguished solutions falling in the corresponding category. Moreover, Section 2.7 complements the description of relevant examples from the current state of the art by surveying further programming solutions. Next, Section 2.8 illustrates a mapping of existing WSN programming solutions to our taxonomy, so to give the reader a broad, yet well-organized perspective on available approaches. Finally, in the same section we draw several observation and discuss common trends emerged from our analysis.

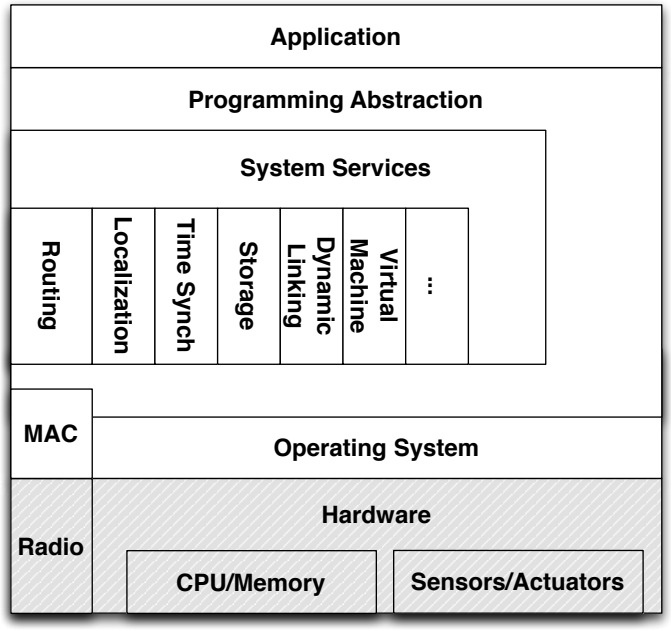


Figure 2.1.: Reference architecture.

2.2. Reference Architecture

Traditional distributed systems have been designed with a strict layered approach. Full-fledged operating systems providing lots of functionality have been used, along with network stacks comprising several levels, from the physical access layer up to the application level.

A similar approach is ill-suited to WSNs, because of the characteristics of the devices employed. Indeed, resource scarcity demands for cross-layer, lightweight approaches where the functionality are optimized to take advantage of feedback information at different levels. The operating system is limited to providing basic mechanisms, e.g., scheduling and thin hardware abstraction layers, while the network stack is most often tailored to the specific application and usually comprises only the MAC and routing layers. Given the wide range of possible configurations enabled by this design approach, it is useful to define a conceptual architecture to cast our taxonomy in the context of WSN research at large.

A pictorial representation of our reference architecture is shown in Fig-

ure 2.1. Its core constituents are described in the following, along with brief mentions of relevant examples in the current state of the art.

Application. Interestingly, in the current literature the term "sensor network application" is used in a very broad sense. For instance, in [21] mechanisms such as node localization and time synchronization are also termed as "applications". We explicitly separate similar functionality from the application level to avoid confusion, and term them as *system services*. Specifically, in our reference architecture an application provides data the end-user can make direct use of. Differently, system services are mechanisms which do not provide any useful information by themselves, yet they are needed in support of specific applications.

Notably, an application's distinctive traits dictate the features that the most appropriate programming abstraction must have to express the corresponding requirements. Therefore, in Section 2.3 we provide a classification of the characteristics of relevant WSN applications in light of the influence they bear on the possible programming solutions of choice.

Programming abstraction. Solutions at this level constitute the major focus of our taxonomy. They empower programmers with higher-level constructs to express various forms of processing, from the application itself to system services, e.g., routing protocols. In this field, the only characterizing dimension that received some attention so far is the one of *node-centric* programming vs. *macroprogramming* [32]. The former generally refers to programming abstractions to express the processing of individual nodes. Consequently, the overall system behavior must be described in terms of pair-wise interactions among nodes within radio range. Differently, macroprogramming solutions are usually characterized by higher-level abstractions used to program the system as a whole, regardless of the single devices.

Nonetheless, the above distinction falls short of expectation in capturing the essence of currently available programming abstractions. For instance, both TinyDB [17] and Kairos [32] are commonly regarded as macroprogramming solutions. However, the former provides an SQL-like interface where no mention of individual nodes is made. The latter, instead, revolves around an imperative programming language where dedicated constructs are provided to iterate trough the neighbors of a given node, and communication occurs by reading or writing shared variables at specific nodes. Therefore, the notion of individual device does not disappear in Kairos' programming model.

To gain a deeper understanding of existing WSN programming approaches, we pursue our investigation along two orthogonal lines:

- In Section 2.5, we study the nature of the *language constructs* provided to programmers. These constitute the entry point to the system functionality, and are therefore pivotal for describing the application processing.

- In Section 2.6, we analyze the *architectural aspects* of existing WSN programming solutions. In a sense, we explode the programming abstraction layer in Figure 2.1. This allows us to have a closer look at how the different approaches relate to each other and to the other conceptual blocks in our reference architecture.

System services. As already mentioned, these include additional mechanisms built atop the core functionality provided by the operating system. Based on the requirements of the application at hand, only a subset (or none) of these mechanisms may be employed. For instance, generic *routing* mechanisms [33] fall in this category, alongside with *localization* mechanisms [34] and *time synchronization* protocols [35, 36]. Notably, some approaches in routing play at border between system services and programming abstractions by providing simple language constructs to express the data flows required, e.g., as in Directed Diffusion [37]. Solutions providing *storage* services have also been proposed, e.g., [38,39]. Mechanisms to enable *dynamic linking* of binary code are present either at this layer, or at the operating system level. As for the former approach, two examples are the Impala [40] middleware and FlexCup [41], the latter empowering the TinyOS [42] operating system with the ability to update single components at run-time. Similar techniques are often coupled with mechanisms allowing for differential patching [43,44], and dedicated dissemination protocols [45–49]. Reprogramming is brought to an extreme using *virtual machines* for WSN nodes. These usually sit atop the operating system and enable on-the-fly execution of arbitrary, scripting-like code. Representatives of this class are Maté [50], ASVM [51], VMStar [52], and SwissQM [53]. The majority of these solutions use a dedicated instruction set, whereas SwissQM [53] enables the execution of custom SQL queries. As for layering aspects, instead, a notable exception is SquackVM [54], which sits directly atop the bare hardware and enables execution of a dedicated subset of the Java language.

Operating system. It is in charge of providing functionality such as scheduling and basic concurrency mechanisms. In addition, it usually provides a thin layer of abstraction to access the underlying hardware, e.g., [55]. Several operating systems for WSN nodes have been proposed so far. The most widespread solution is the TinyOS [42] operating system. Alternatives are the Contiki OS [56], SOS [57], Mantis [58], RETOS [59], and NANO-rk [60]. Among them, SOS, Mantis, and Contiki feature dynamic linking functionality. The concurrency model employed varies from event-driven solutions [42] to preemptive, time-sliced multi-threading [58,59] and asynchronous message passing [57]. These approaches have also been further augmented in several works. Examples are Fibers [14], TinyThreads [61], and Y-Threads [62]. These approaches add a threading model to TinyOS by enabling different forms of blocking calls. Fiber allows a single blocking context, whereas TinyThreads enables cooperative multi-threading by giving programmers a way to yield explicitly the current execution context. Y-Threads, instead, are similar to Fiber, but they provide preemptive multi-threading. Protothreads [63] enables a form of cooperative multi-threading in Contiki. However, they do not store the execution contexts, thus requiring the programmer to save and restore the relevant state by hand. The Object State Model [64] extends the event model with state machines. It also provides the ability to compose different state machines to build hierarchies.

MAC. A plethora of MAC-layer protocols have been hitherto proposed [65, 66]. These essentially fall in two categories. *Randomized* protocols such as [67–69] regulate the access to the physical layer opportunistically, based on the current transmission requests. Conversely, *time-slotted* protocols assign the nodes with predefined time-slots to schedule their transmissions over time, e.g., as in [70, 71]. While the former class of protocols is easier to implement and better tolerates nodes joining or leaving, the latter class enables better reliability and greater energy savings. The radio can indeed be switched to a low-power mode based on the current transmission schedule. Nonetheless, the latter protocols generally require tight time synchronization among the nodes in some k-hop neighborhood.

Hardware layer. Comprising the micro-controller and data/program memory, the radio device, and the possible sensors or actuator attached, it defines the capabilities of the chosen platform in terms of communication, processing, and interactions with the environment. A plethora of WSN hardware platforms exist both as commercial products and in the form of

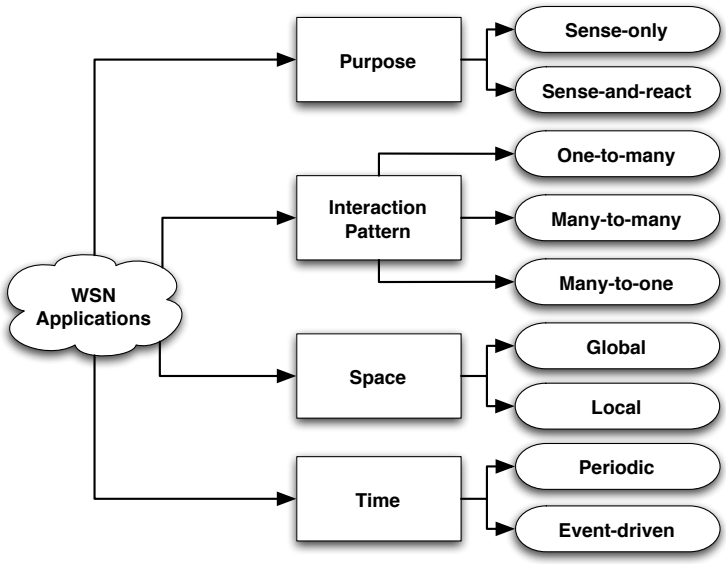


Figure 2.2.: Dimensions for classifying WSN applications.

research prototypes, e.g., [72–80]. However, the individual hardware components used for processing and communication do not differ sensibly. Most platforms use a 16-bit Texas Instruments MSP430 processor or a 8/16-bit micro-controller of the Atmel ATMega family. Notable exceptions are the IMote 2 and SunSPOT platforms, using the more powerful Intel PXA and ARM920T cores, respectively. The amount of volatile memory ranges from 2 Kb to 512 Kb, whereas external memory support varies from 128 Kb to 4 Mb. As for radio hardware, most platforms work in the 2.4Ghz ISM band, and feature IEEE 802.15.4 [81]-compliant radio chips, e.g., the ChipCon 2420. Alternative solutions operate in the 868/916 Mhz band, e.g., using the ChipCon 1000 transceiver. Various types of sensors and actuators may be attached, also using expansion boards like [82]. The specific type of sensing/actuator device is largely application-specific.

2.3. Sensor Network Applications

WSNs are being employed in a variety of settings. As we previously observed, these usually differ in the requirements developers must satisfy. Nonetheless, it is possible to identify common characteristics that can fa-

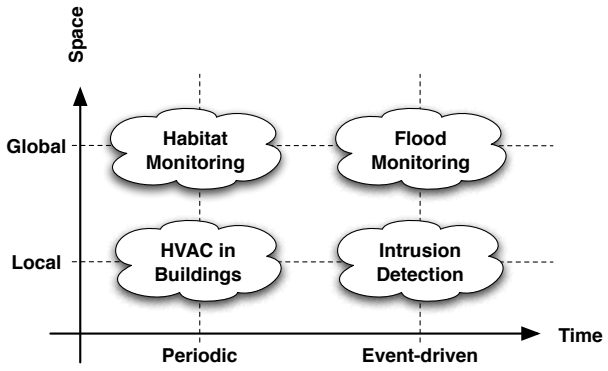


Figure 2.3.: Example WSN applications in space and time.

cilitate the mapping to existing programming abstractions. Figure 2.2 provides a high-level view on the various dimensions we deem relevant to this goal. These are described next.

Purpose. As pointed out in Chapter 1, WSNs can be used to *sense* from the environment and collect data for later, off-line analysis. Nevertheless, the system can also react to sensed data, and use them to drive *actuation* on the environment under control, thus closing the control loop. Notably, introducing actuators drastically change the nature of the application: the system becomes highly heterogeneous, as opposed to the mostly homogeneous architectures employed in sense-only scenarios; and multiple activities must be carried out simultaneously, e.g., to control actuators installed in different parts of the system.

Interaction pattern. Depending on the application goals, the nodes in the network collaborate differently. Early WSN deployments mostly featured a *many-to-one* interaction pattern: data is funneled from all nodes in the network to a central collection point. Conversely, more recent applications are characterized by *many-to-many* or *one-to-many* interactions. For instance, the former is germane to scenarios where multiple actuators need to gather data from overlapping sets of sensing devices, as in [27].

Space and time. WSN applications deal with interacting with the surrounding environment in *space* and *time*. Therefore, a distinction can be made along these two orthogonal dimensions, as illustrated in Figure 2.3. The application behavior over time can be classified as either:

- **Periodic:** applications like HVAC in buildings [27] are designed to continuously process information. The application therefore performs periodic tasks to gather data, coordinate with other part of the system, and perform the actuation needed.
- **Event-driven:** applications like flood monitoring [83, 84] perform some relevant processing only when specific environmental conditions are met, e.g., a sensor reading raises above a thresholds. Until then, the system is mostly quiescent.

Dually, two classes of applications can be identified depending on their behavior in space:

- **Global:** applications such as habitat monitoring [22] sense and process data throughout the whole system, as the phenomena observed usually spans the whole geographical area where the system is deployed.
- **Local:** applications like intrusion detection [85], on the other hand, limit their processing in some vicinity of the phenomena observed. This is usually limited in space, so that data coming from global observations are not needed.

Interestingly, the range of existing WSN application covers all combinations of the above dimensions. For instance, flood monitoring [83, 84] is representative of applications where the phenomena of interest is sporadic, yet data must be sensed and processed throughout the system. This is needed to give domain-experts the ability to understand a given phenomena based on environmental conditions possibly distant from the phenomena itself. Dually, HVAC in buildings exemplifies applications where the processing is continuous and limited to specific regions of the system. For instance, controlling an air conditioner in a room usually requires sensing temperature and humidity only in the same room [27].

Figure 2.4 illustrates a mapping from WSN applications found in the current literature to the above characterization. Several observation can be drawn:

- Sense-only applications are mostly characterized by many-to-one interactions. The few examples requiring many-to-many interactions are due to the possible co-existence of multiple users willing to gather the same data.

Application	SO/SR	Interactions	Space	Time
Habitat Monitoring [22]	SO	Many-to-one	Global	Periodic
Wildlife Monitoring [23]	SO	Many-to-one	Global	Periodic
Glacier Monitoring [86, 87]	SO	Many-to-one	Global	Periodic
Grape Monitoring [24]	SO	Many-to-one	Global	Periodic
Landslide Detection [88]	SO	Many-to-one	Global	Periodic
Volcano Monitoring [89]	SO	Many-to-one	Global	Periodic
Passive Structural Monitoring [90]	SO	Many-to-one	Global	Periodic
Fence Monitoring [91]	SO	Many-to-one	Local	Event-driven
Sniper Localization [92]	SO	Many-to-one	Local	Event-driven
Intrusion Detection [85]	SO	Many-to-one	Local	Event-driven
Forest Fire Detection [93]	SO	Many-to-one	Global	Event-driven
Flood Detection [83, 84]	SO	Many-to-one	Global	Event-driven
Health Emergency Response [94]	SO	Many-to-one	Local	Periodic
Avalanche Victims Rescue [95]	SO	Many-to-many	Local	Periodic
Smart Tool Box [96]	SO	Many-to-many	Global	Event-driven
Vital Sign Monitoring [97]	SO	Many-to-many	Global	Event-driven
Vehicular Traffic Control [98]	SR	Many-to-many	Local	Periodic
Smart Homes [26]	SR	Many-to-many	Local	Periodic
Assisted Living [99]	SR	Many-to-one/ One-to-many	Local	Periodic
Building Control and Monitoring [100]	SR	Many-to-one/ One-to-many	Local	Periodic
Active Structural Monitoring [90]	SR	Many-to-many	Local	Periodic
Heating Ventilation and Air Conditioning Control [27]	SR	Many-to-many/ One-to-many	Local	Periodic
Tunnel Control and Monitoring [4]	SR	Many-to-many/ One-to-many	Local	Periodic

Figure 2.4.: Mapping example WSN applications to the classification in Figure 2.2.

- The behavior in space and time of sense-only applications varies sensibly. Although applications periodically gathering data on a global scale constitute a sizable portion of the domain space, several examples exist featuring different combinations of space/time behavior.
- The space behavior of sense-and-react applications is mostly local. Actuators are indeed limited in the extent by which they can influence the environment. Therefore, they do not need to gather sensor readings outside their range of actuation [2].
- Similarly, the behavior in time of sense-and-react application is mostly periodic. Classical control mechanisms indeed require continuous interactions with the environment to adapt the actions taken to the

physical world dynamics.

Based on the above observations, in the following we survey currently available WSN programming abstractions in light of the requirements programmers must meet.

2.4. Taxonomy Overview

The overall organization of our taxonomy of WSN programming abstractions is illustrated in Figure 2.5. Following the two major lines of investigation previously introduced, we investigate in Section 2.5 the distinctive traits of the language constructs provided to the programmer. Firstly, we look at how developers describe *computation* vs. *communication*. Indeed, in the current state of the art both aspects are described to different extents, by focusing on single nodes, group of nodes, or the system as a whole. Next, we analyze the *programming idioms* employed in the different solutions, and the corresponding *distribution models* used by programmers to express the necessary coordination among nodes.

In Section 2.6 we study how the various solutions interact with each other, and with the rest of our reference architecture. We begin by investigating the extent to which a solution can be *composed* with other. Indeed, approaches exist that are designed to be building blocks employed in collaboration with other programming solutions, as opposed to holistic approaches providing complete support for writing full-fledged applications without the ability to coordinate with different abstractions. Next, we investigate the *reach* of a given WSN programming solution in our reference architecture. Remarkably, some of the existing approaches are not limited to writing applications. Rather, they are able to reach into the lower level of the architecture, e.g., by providing support to the implementation of system services. Moreover, we study whether currently available approaches allow the application requirements to *penetrate* the stack, e.g., investigating the presence of features giving programmers control over low-level functionality. Finally, we assess the range of *supported platforms* to evaluate the actual viability of the corresponding solutions.

Throughout the discussion, we provide relevant examples to illustrate the various dimensions for classification. In every case, we first provide an overview on the specific solution, describing the abstractions at the core of the programming model. Then, we illustrate an example taken from the

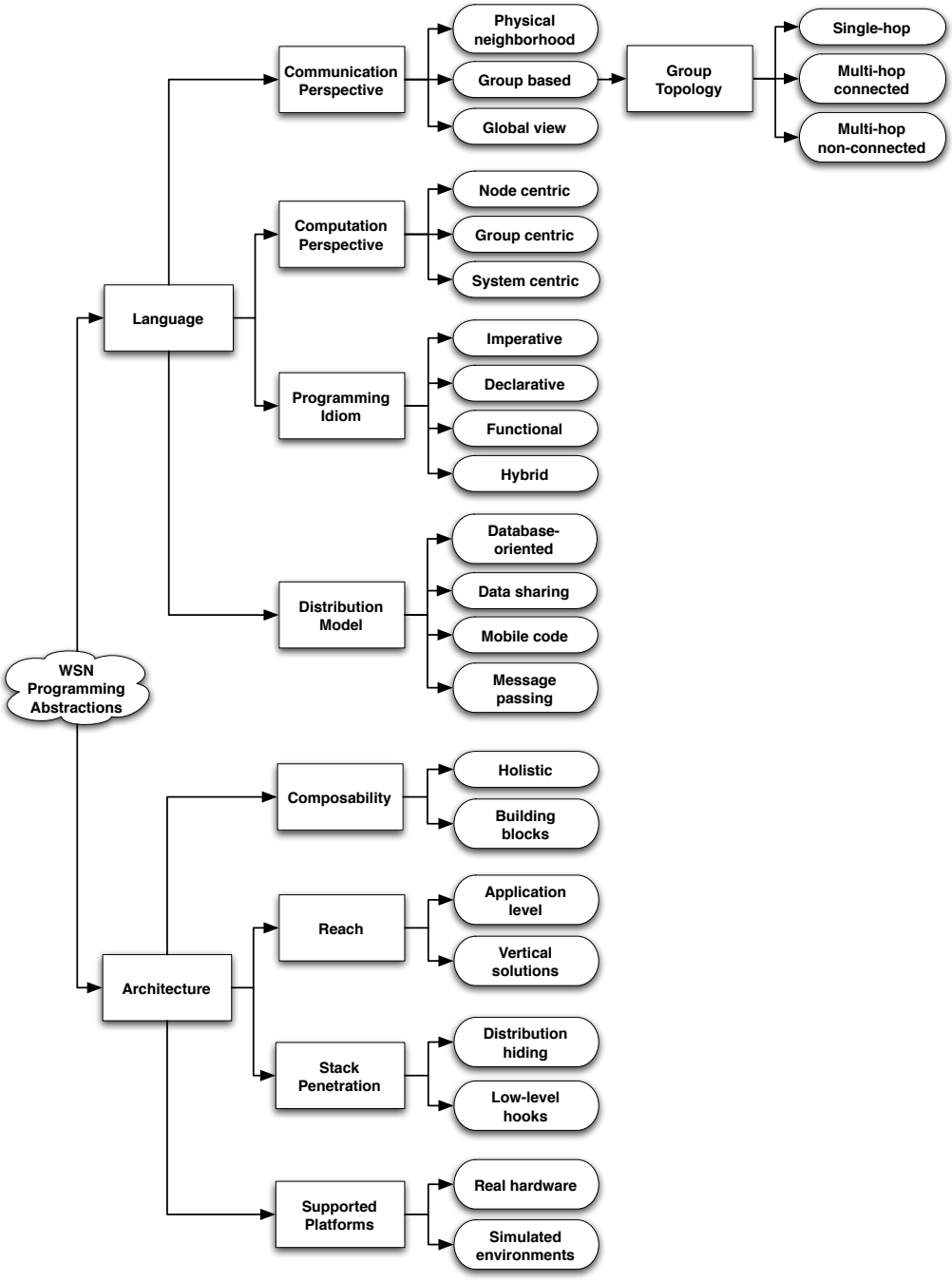


Figure 2.5.: Taxonomy organization.

corresponding literature, and conclude by providing some highlights on the underlying implementation of the programming model at hand.

2.5. Characterizing the Language

In this section, we study the nature of the language constructs made available to WSN programmers through a series of paradigmatic examples. In a sense, here we play at the border between the application and programming abstraction layers in Figure 2.1, to study how developers can leverage off the functionality available at lower levels.

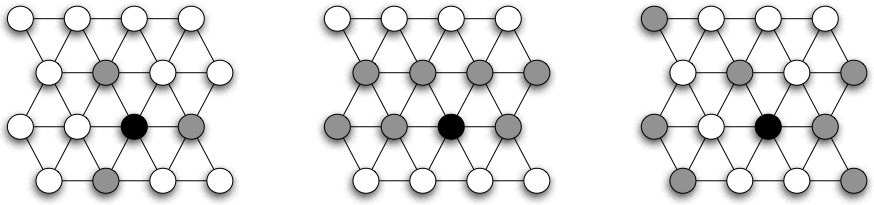
2.5.1. Communication Perspective

WSN nodes can hardly perform any useful task if left alone. It is indeed the overall collaboration of a large number of such devices that allows the system to accomplish a higher-level goal. Therefore, communication among different nodes plays a pivotal role in describing the application processing.

Classification

Existing WSN programming abstractions radically differ in the way they let developers express inter-node communication. Three approaches primarily emerge in the current state of the art:

- **Physical neighborhood:** approaches such as NesC [28] and FACTS [101] give programmers basic communication facilities to exchange messages with nodes within radio range. Message recipients can either be all reachable nodes (*broadcast* communication) or a specific device (*unicast* communication). The addressing scheme leverages off statically assigned node identifiers.
- **Group based:** solutions like Hood [21], Abstract Regions [14], and EnviroSuite [102] provide APIs to target subsets of nodes depending on application-level information. The addressing scheme is mainly based on programmer-provided information. Furthermore, the topological characteristics of the group may also differ, as depicted in Figure 2.6:
  - **Single-hop groups:** the target nodes are assumed to be reachable from the sender with a single message transmission, as in Hood [21]. An example is depicted in Figure 2.6(a).



(a) Single-hop group. (b) Multi-hop connected group. (c) Multi-hop non-connected group.

Figure 2.6.: Topological characteristics of group based communication. Grey nodes are those addressed by the black node.

  - **Multi-hop connected groups:** the target nodes can be multiple hops away from the sender, yet there must be a path of only target nodes from the sender to the farthest target node, as in EnviroSuite [102]. Figure 2.6(b) reports such an example.
  - **Multi-hop non-connected groups:** no assumptions on the geographical location of the target nodes are made, as in Abstract Regions [14]. An example is illustrated in Figure 2.6(c).

- **Global view:** differently from the aforementioned approaches, solutions such as TinyDB [17] and Pleiades [15] allow programmers to look at communication from a global viewpoint, without necessarily taking the perspective of a single node sending data to a specific set of destinations.

Generally, solutions based on physical neighborhood or group based communication do not radically differ from traditional distributed computing. Indeed, communication is still played from the perspective of an individual node willing to exchange data with some remote hosts. In a sense, this places the programmer *inside* the network. Approaches based on global views, instead, provide a node-independent standpoint. Programmers play the role of an external entity, describing communication from *outside* the network.

As paradigmatic examples of the above classification, here we illustrate the nesC [28] language and Active Messages [103] for communication in the physical neighborhood, the Abstract Regions [14] programming model for

```
interface AMSend {
  command error_t send(am_addr_t addr, message_t* msg, uint8_t len);
  command error_t cancel(message_t* msg);
  event void sendDone(message_t* msg, error_t error);
  command uint8_t maxPayloadLength();
  command void* getPayload(message_t* msg, uint8_t len);
}

interface Receive {
  event message_t* receive(message_t* msg,void* payload,uint8_t len);
}
```

Figure 2.7.: nesC Active Message interfaces for sending and receiving messages.

group-based communication, and the Pleiades [15] system for communication based on global views.

Physical neighborhood communication: nesC

Overview. nesC is an event-driven programming language for WSNs derived from C. Its main goal is to provide basic programming abstractions for the TinyOS [42] operating system. Applications in nesC are built by interconnecting *components* that interact by *providing* or *using* interfaces. An interface lists one or more functions, tagged as *commands* or *events*. Commands are used to execute actions, while events are used to collect the results asynchronously. A component providing an interface implements the commands it declares, whereas the one using the interface implements its events. Therefore, data flows both ways between components connected through the same interface.

Although higher-level abstractions have been developed atop nesC [104], the communication mechanisms are mainly based on Active Messages: a scheme whereby messages are tagged with an identifier that specifies what component must process a given message upon reception. Components leverage off Active Messages mainly using two simple interfaces, shown in Figure 2.7. Additional interfaces are also provided to deal directly with the radio hardware, e.g., by setting the transmission power level on a per-packet basis.

Example. A fragment of code implementing a component that queries the sensing device and sends the reading in broadcast is reported in Figure 2.8.

```
module Sampler {
  uses interface Boot;
  uses interface TemperatureSensor;
  uses interface AMSend;
}
implementation {

  bool transmitLock;
  message_t msgBuffer;

  event void Boot.booted {
    call TemperatureSensor.getData();
  }
  event void TemperatureSensor.dataReady(uint16_t v){

    uint16_t* msg_payload = (uint16_t*) msgBuffer->payload;
    *msg_payload = v;
    if (!transmitLock) {
      transmitLock = TRUE;
      if (!call AMSend.send(TOS_BCAST_ADDR, &msgBuffer,
                            sizeof(message_t))) {
        transmitLock = FALSE;
      }
    }
  }
  event void AMSend.sendDone(message_t* msg, result_t success) {
    if(transmitLock && msg == msgBuffer ) {
      transmitLock = FALSE;
    } else {
      // Error...
    }
  }
}
```

Figure 2.8.: Sense and broadcast component in nesC.

The `booted()` function in the `Boot` interface is called at system start-up. Inside it, we call the `getData()` command in the `TemperatureSensor` interface, whose providing component is directly bound to the sensing device. This is a typical *split-phase* operation [28]: the command returns immediately, and the caller component is asynchronously notified when the device completes its operation, in our case using the `dataReady` event. In the corresponding event handler, we pack the sensed value in a message and call the `AMSend.send` command. To make sure we do not try to send another message while the transmission is in progress, we set the `transmitLock` flag. This is unset inside `AMSend.sendDone`, that is automatically called when the ongoing transmission completes. The level of abstraction pro-

vided is therefore quite low, as the programmer is forced to deal directly with radio-related aspects such as scheduling multiple transmissions over time.

Implementation highlights. The mechanisms implementing the Active Message interfaces are normally bound the specific MAC-level mechanisms employed, e.g., [67], or directly to radio hardware. As a result, part of them is often platform-specific. Generally, the implementations provide (unreliable) 1-hop unicast or broadcast transmissions. Specific MAC layers, nonetheless, can offer some form of reliability when coupled with specific radio chips [68, 105]. Moreover, there is no support for packet buffering. Instead, the application provides its own storage for sending and receiving messages.

Group based communication: Abstract Regions

Overview. Abstract Regions is a set of general-purpose communication primitives providing addressing, data sharing, and aggregation among a given subset of nodes. A *region* defines a neighborhood relationship between a specific node and other nodes in the system. For instance, a region can be defined to include all nodes within a given number of hops or within distance d. Data sharing is accomplished using a tuple space-like paradigm, while dedicated constructs are provided to aggregate information stored at different nodes in a region. Although Abstract Regions are built atop nesC/TinyOS, they also employ a lightweight thread-like concurrency model called *Fibers* to provide blocking operations. The Abstract Regions API is depicted in Figure 2.9. By their nature, Abstract Regions target applications exhibiting some form of spatial locality, e.g., tracking moving objects, or identifying the contours of physical regions [106].

Example. To demonstrate the use of Abstract Regions, we illustrate how to write an object tracking application using the API in Figure 2.9. Similar applications involve taking periodic measures from relevant sensor devices (e.g., magnetometers), and compare them against a threshold. Nodes sensing a value above the threshold coordinate to elect a leader, e.g., the node with the highest reading. The leader computes the centroid of all readings, and transmits the result back to the user.

Figure 2.10 shows the Abstract Regions code for object tracking. All distributed processing required is performed through the region API. Initially, every node sets up the region to include the 8 geographically closest nodes

```
// Discover region
result_t Region.formRegion(<region specific args>, int timeout);
// Wait for region discovery
result_t Region.sync(int timeout);

// Set and get shared variables
result_t SharedVar.put(sv_key_t key, sv_value_t val);
result_t SharedVar.get(sv_key_t key, addr_t node,
                       sv_value_t *val, int timeout);
// Wait for shared variable gets
result_t SharedVar.sync(int timeout);

// Reduce 'value' to 'result' with given 'operator'
// 'yield' returns the percentage of nodes responding
result_t Reduce.reduceToOne(op_t operator, sv_key_t value,
                            sv_key_t result, float *yield,
                            int timeout);
// Reduce and set result in all nodes
result_t Reduce.reduceToOne(op_t operator, sv_key_t value,
                            sv_key_t result, float *yield,
                            int timeout);
// Wait for reductions to complete
result_t Reduce.sync(int timeout);
```

Figure 2.9.: Abstract Regions API (adapted from [14]).

(`k_nearest_region.create()`). In the main loop, each node periodically queries the sensor, and makes the output available to other nodes in the regions, along with its physical location. This is achieved by leveraging off different shared variables, and using `region.putvar()` to set their value. If the sensor reading is above the threshold, every node first determines the highest reading in the region by using `region.reduce()` with operation `MAX`. If the local node is indeed the one with the highest reading, a series of sum-reductions is performed over the shared variable in the region to compute the centroid, and the result is sent to the base station.

Implementation highlights. Abstract Regions leverages off nesC to produce executable code. As for implementation of the mechanisms behind the Abstract Regions API, these depend on the particular region employed. For instance, the region used in the example is implemented using a form of geographically-limited flooding. Differently, a planar-mesh region used in a contour-finding application can be implemented based on Yao graphs [107]. Generally, different regions require different implementations.

```
location = get_location();

// Region setup to include 8 nearest neighbors
region = k_nearest_region.create(8);

while (true) {
  reading = get_sensor_reading();

  // Store data as shared variables
  region.putvar(reading_key, reading);
  region.putvar(reg_x_key, reading * location.x);
  region.putvar(reg_y_key, reading * location.y);

  if (reading > threshold) {
     // Retrieve the id of the node with max reading
     max_id = region.reduce(OP_MAXID, reading_key);

     // If this node is leader
     if (max_id == my_id) {
       // Compute centroid
       sum = region.reduce(OP_SUM, reading_key);
       sum_x = region.reduce(OP_SUM, reg_x_key);
       sum_y = region.reduce(OP_SUM, reg_y_key);
       centroid.x = sum_x / sum;
       centroid.y = sum_y / sum;
       send_to_basestation(centroid);
     }
  }
  sleep(periodic_delay);
}
```

Figure 2.10.: Object tracking in Abstract Regions (adapted from [14]).

Global view communication: Pleiades

Overview. Pleiades is a programming language providing a global view on the entire sensor network. To achieve this, Pleiades extends the C syntax with constructs for addressing the nodes in a network and accessing the local state of individual nodes. A Pleiades program normally features a sequential thread of control, i.e., execution unfolds with only one node in the system executing any Pleaides instruction at any point in time. Nonetheless, a dedicated language construct called `cfor` is provided to introduce concurrent executions. Using `cfor`s, multiple nodes may be executing different Pleaides instructions simultaneously. If deemed required, the underlying run-time can guarantee serializable execution of `cfor`s.

Example. Figure 2.11 depicts an example Pleiades program to imple-

```
#include "pleiades.h"
boolean nodelocal isFree=TRUE;
nodeset nodelocal neighbors;
node nodelocal neighborIter;

void reserve (pos dst) {
  boolean reserved = FALSE;
  node nodeIter, reservedNode = NULL;
  node n=closest_node(dst);
  nodeset loose nToExaming = add_node(n, empty_nodeset());
  nodeset loose nExamined = empty_nodeset();

  if (isfree@n) {
    reserved = TREE; reservedNode = n;
    isfree@n = FALSE;
    return;
  }

  while (!reserved && !empty(nToExamine)) {
    cfor (nodeIter=get_first(nToExamine);
          nodeIter!=NULL;
          nodeIter = get_next(nToExamine)) {
      neighbors@nodeIter=get:neighbors(nodeIter);
      for (neighborIter@nodeIter=get_first(enighors@nodeIter);
           neighborIter@nodeIter|=NULL;
           neighborIter@nodeIter=get_next(neighbors@nodeIter)) {
        if (!member(neighborIter@nodeIter,nExamined))
          add_node(neighborIter@nodeIter,nToExamine);
      }
      if (isfree@nodeIter) {
        if (!reserved) {
          reserved=TRUE; reservedNode=nodeIter;
          isfree@nodeIter=FALSE;
          break;
        }
      }
      remove_node(nodeIter,nToExamine);
      add_node(nodeIter,nExamined);
    }
  }
}
```

Figure 2.11.: A street-parking application in Pleiades (adapted from [15]).

ment a street-parking application. Sensors are deployed in parking spots to monitor whether they are currently occupied. The application goal is to identify the node with a free spot closest to the driver's destination. This is achieved by iterating among the nodes in the network in search of the first device installed in an available spot, starting from the node closest to the desired destination. The program makes use of most of Pleiades language

features:

- The `node` data type provides an abstraction of a single WSN device, whereas `nodeset` gives programmers a way to iterate in a collection of nodes. Various helper functions are provided to obtain node collections. For instance, `get_network_nodes()` returns a `nodeset` containing the entire set of currently available nodes. Differently, `get_neighbors(n)` returns `n`'s one-hop neighbors.

- Variables are normally shared across all nodes in the system. When the processing requires nodes to have private variables, these are declared using the `nodelocal` attribute, e.g., `isfree` in Figure 2.11. Node-local variables are accessed using `var@e`, where `var` is a `nodelocal` variable and `e` is a `node`.

- The `cfor` construct allows for concurrent execution of the loop body on all nodes in a `nodeset`. If needed, the Pleiades run-time can ensure that the effect of a `cfor` corresponds to some sequential execution of the loop. Here, this is required to make sure that only one free node is reserved for the car arriving. Differently, access to `loose` variables is not synchronized inside `cfor` loops.

Implementation highlights. The Pleiades compiler performs data-flow analysis to partition the program in a set of independent execution units called *nodecut*s. Each nodecut is designed to run on a single node. The compiler assigns nodecuts to single nodes based on the expected communication cost for accessing variables at remote nodes inside a given nodecut. When the execution flow transitions from one nodecut to the following, the control flow is migrated across nodes. A dedicated locking mechanism is provided to implement serializable execution of `cfor`s. A coordinator is elected among the nodes possibly involved, which manages the locks on shared variables depending on the current state of execution. As this may lead to deadlocks, e.g., in the case of nested `cfor`s, the coordinator also monitors the execution state of other nodes involved to determine the presence of deadlocks.

2.5.2. Computation Perspective

Computation constitutes an orthogonal dimension w.r.t. communication. Nonetheless, the provision of expressive language constructs to describe the

processing required is key to expressing the application behavior. Furthermore, the interplay between communication and computation in WSNs plays an even more important role, e.g., in minimizing communication thanks to local aggregation and fusion of sensed values.

Classification

Similarly to communication, existing WSN programming solutions give developers different ways to express the application processing:

- **Node centric:** solutions such as nesC [28] and ATaG [108] revolve around constructs whose execution is guaranteed to alter the state of individual nodes.
- **Group centric:** differently from node-centric solutions, approaches like Regiment [16] and Pieces [109] allow the application processing to alter the state of some subset of nodes at once.
- **System centric:** group-centric approaches are brought to an extreme when the effects of a single instruction span the whole network, e.g., as in TinyDB [17].

To make this illustration more concrete, in the following we describe the ATaG [108] framework as example of node-centric approaches, the Regiment [16, 110] system for group-centric processing, and TinyDB [17] as example of a system-centric solution.

Node-centric computation: Abstract Task Graph (ATaG)

Overview. The Abstract Task Graph (ATaG) is a programming framework providing a mixed declarative-imperative approach. The notions of *abstract task* and *abstract data item* are at the core of ATaG's programming model. A task is a logical entity encapsulating the processing of one or more data items, which represent the information. Different copies of the same task run on different nodes, yet they are independent of each other. The flow of information between tasks is defined in terms of their input/output relations. To achieve this, *abstract channels* are used to connect each data item to the tasks that produce or consume it.

The code within a task is written using an imperative language. To express data exchange between tasks, programmers are provided with the

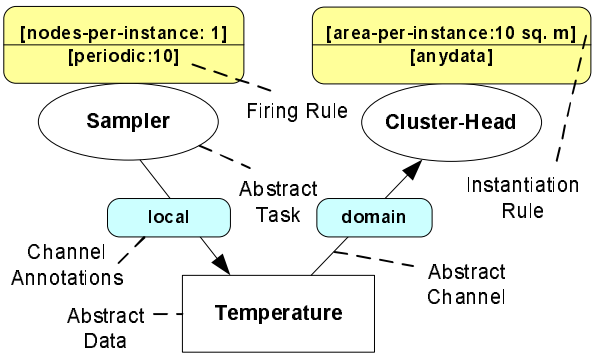


Figure 2.12.: A sample ATaG program.

abstraction of a shared data pool, where each task can output data, or be notified when some data of interest is available. Dedicated APIs are provided for this. To support the former aspect, a single `putData(DataItem)` operation is made available. The second aspect is handled by providing the programmer with an automatically generated template for each task, that lists an empty `handleDataItem()` function for each incoming channel. The programmer fills these functions implementing the processing associated to each input data item.

Example. Figure 2.12 illustrates a sample ATaG program specifying a cluster-based, data gathering application. Sensors within a cluster take periodic temperature readings, which are then collected by the corresponding cluster-head. The former behavior is encoded in the *Sampler* task, while the latter is represented by *Cluster-Head*. The *Temperature* data item is connected to both tasks using a channel originating from *Sampler*, and a channel directed to *Cluster-Head*.

Tasks are annotated with firing and instantiation rules. The former specify when the processing in a task must be triggered. In our example, the *Sampler* task is triggered every 10 seconds according to the `periodic` rule. The *Cluster-Head* fires whenever at least one data item is available on *any* of its incoming channels, in accordance with its `any-data` firing rule. Tasks run on the individual nodes according to the instantiation rules specified by programmers. The `nodes-per-instance:1` construct requires the task to be instantiated once on every node. On the other hand, the `area-per-instance` construct used for *Cluster-Head* entails partitioning the geographical space according to the given parameter, and deploying

one instance of the task per partition.

Abstract channels are annotated to express the interest of a task in a data item. In our example, the *Sampler* task generates data items of type *Temperature* kept `local` to the node where they have been generated. The *Cluster-Head* uses the `domain` annotation to gather data from the temperature sensors in its cluster, which binds to the system partitioning obtained by `area-per-instance` and connects tasks running in the same partition.

Implementation highlights. The ATaG compiler takes as input the description of tasks and channels, examines the corresponding flow of data, and decides on some allocation of tasks to nodes depending on information on the target environment, e.g., the location of nodes. The output of the compiler targets a dedicated node-centric run-time layer, designed to be highly-modular [111]. Different concerns are therefore isolated in different modules. Some of the mechanisms are, however, not provided beforehand. For instance, the programmer must provide the most appropriate routing scheme depending on the specific application and target environment.

Group-centric computation: Regiment

Overview. Regiment is a functional language geared towards applications exhibiting spatial locality. In Regiment, the programmer manipulates sets of data streams called *signals*. These may represent readings on individual nodes, the outcome of a node's local computation, or an aggregate value obtained by processing multiple signals. In addition, Regiment features a notion of *region* as a set of spatially correlated signals, e.g., the set of sensor readings generated by nodes in a given geographic area. The processing is expressed by applying programmer-provided functions to the data streams in a region. In addition, a `Node` construct is provided to access the state of individual devices, e.g., to gather the local node identifier or to query the sensing device.

Example. Consider a system for plume monitoring. Sensors are deployed over a large geographical area for early detection of plumes. Key to the correctness of the application is avoiding false positives due to noisy readings from individual sensors. To meet this requirement, we must make sure the sum of the readings gathered by nodes around the phenomena exceeds a pre-specified threshold.

Figure 2.13 depicts an example Regiment program to implement the above processing. The programmer first defines a set of functions used in

```
fun abovethreshold(t) { t > CHEM_THRESHOLD }
fun read(n) { sense("concentration", n) }
fun sum(r) { rfold((+), 0, r) }

readings = rmap(read, world);
detects = rfilter(abovethreshold, readings);

hoods = rmap( fun(t, nd){ khood(1,nd) }, detects);
sums = rmap (sum, hoods);
base <- rfilter( fun(t){ t > CLUSTER_THRESHOLD }, sums);
```

Figure 2.13.: Plume monitoring using Regiment (adapted from [16]).

the rest of the program, e.g., to filter sensed data (`abovethreshold()`), to gather the reading from the sensor (`read()`), or to sum all values sensed in a region (`sum()`). In the latter, `rfold` is used to aggregate all values in region `r` into a single signal, using the `+` operator and `0` as initial value. Next, the programmer must identify a region of nodes that surpass their local thresholds. This is accomplished by first gathering the local readings at all nodes in the system, and then performing the necessary filtering. The former operation is expressed as the application of `read()` to all the nodes in the system, using `rmap`. This takes as input a function and a region, and applies the function to all values in the region. The `world` region in the example represents all nodes in the system. The filtering part is accomplished using `rfilter`, which takes a boolean function and a region as inputs, and returns a region including values for which the input function yields true.

In the example, `hoods` is instead a *nested* region. It consists of the set of nodes in the one-hop neighborhood of every node in `detected`. This is obtained by applying a region formation function (`khood()`) to all nodes in `detects`. Regions can be formed similarly to Abstract Regions, described in Section 2.5.1. The remaining instructions are used to sum the readings in the nested regions created earlier, and to send a notification to the base station in case any of these sums turns out to be above the safety threshold.

Implementation highlights. The Regiment compiler leverages off multiple steps of compilation to generate the final, node-centric running code. A Regiment program is first translated into an intermediate language called RQuery. Subsequently, the region streams are translated into local streams. The output of the compiler is event-driven code written in an intermediate language called token machine language [112]. This language does not

```
SELECT AVG(light), AVG(temp), location
FROM sensors AS s
SAMPLE PERIOD 2 s FOR 30 s
```

Figure 2.14.: Monitoring bird nests using TinyDB (adapted from [17]).

assume any thread-like concurrency model. It is therefore suited for implementation on top of event-driven operating systems for WSNs, such as TinyOS. As for communication, nodes in a given region exchange data using spanning trees. These are created and maintained by the Regiment support layer on every node.

System-centric computation: TinyDB

Overview. TinyDB is a query processing system for WSNs whose focus is to optimize energy consumption by controlling where, when, and how often data is sampled from the sensing devices. In TinyDB the user submits queries at the base station. These are parsed, optimized depending on the data requested, and injected into the network. Upon reception of a query, a node process the corresponding requests, gathers some reading if needed, and funnels the results to the base station. The language used to express queries is a dialect of SQL. The data model revolves around a single `sensors` table, that logically contains one row per node per instant in time, and one column per possible data type the node can produce (e.g., temperature or light). The data in this table is materialized only on request, unless *materialization points* are created in the network to proactively gather the data.

Example. Consider an application to monitor the presence of birds in nests. The user would like to gather the average light and temperature readings close to the nest. In addition, she wants to do so once every 2 seconds for 30 seconds total.

The above processing can be encoded in a TinyDB query as illustrated in Figure 2.14. The `SELECT`, `FROM` and `WHERE` clauses have the same semantics as in standard SQL. The `location` attribute is assumed to be obtained from some external localization mechanism. The `SAMPLE PERIOD` construct is used instead for specifying the rate and lifetime of the query.

Implementation highlights. When the query is injected from the base-

station, a routing tree is built spanning all the nodes in the network. The routes are then decorated with meta-data to provide information on the type and nature of data sensed by nodes in a specific portion of the tree. While executing the query at each node, the TinyDB engine performs several optimizations to reduce the amount of data flowing upwards towards the base-station. These mechanisms result in a query-specific execution plan. For instance, data sampling and transmissions are interleaved so to reduce the overall power consumption without affecting the quality of data reported. A dedicated transmission scheme is also employed to schedule the transmissions at nodes at different levels of the tree. The goal is to make data flow upward starting from the leaves, so that intermediate nodes can aggregate information before sending their own readings to the base station.

2.5.3. Programming Idioms

The example solutions described so far already highlight the variety of programming idioms offered to sensor network developers, ranging from fully declarative to purely imperative approaches. As the nature of the constructs provided partially dictates the learning curve for new programmers, the specific idiom employed bears some influences on the acceptance of the language itself.

Classification

Looking at the current state of the art, four major categories can be identified:

- **Imperative approaches:** by far the most widespread, this class of programming solutions leverages off purely imperative languages with *sequential* or *event-driven* semantics. Examples of the former class are Abstract Regions [14] and Pleaides [15], whereas a solution adopting the latter is nesC [28].
- **Declarative approaches:** purely declarative solutions are usually very concise in describing the system behavior, yet they are often intended for very specific application domains. Existing solutions in this category follow either *database-style*, or *rule-oriented* approaches. Representative example of the former class are TinyDB [17] and Cougar [113], whereas FACTS [101] is an example of rule-based solution.

- **Functional approaches:** approaches like Regiment [16] as well as snBench [114] are built upon functional languages. Developers express the application processing by applying one or more functions to data sensed in some part of the system. Similarly to declarative approaches, leveraging off this programming idiom often yields very concise implementations.

- **Hybrid approaches:** combining more than a single idiom to address different aspects can help in achieving separation of concerns, as in ATaG [108].

To better illustrate each of the above classes, here we resort to examples described earlier. Specifically, the Pleiades [15] system, described in Section 2.5.1, provides a natural illustration for imperative approaches featuring sequential semantics. Indeed, Pleiades programs closely resemble standard C code. Furthermore, serializable executions of `cfor`s guarantees sequential semantics across different nodes. The Regiment system [16,110], illustrated in Section 2.5.2, is an example of functional language. Applications needing to process sensed data through multiple stages are naturally expressed this way. The TinyDB [17] query processor we described in Section 2.5.2 features a fully declarative approach. By leveraging off a SQL-like interface, programmers are able specify constraints on the data of interest without specifying the exact procedure to gather the data themselves. Finally, the ATaG framework illustrated in [108], features a hybrid approach. Communication among nodes is described in a declarative manner, whereas computation is expressed using an imperative language.

2.5.4. Distribution Models

Existing solutions provide different abstractions to deal with the data exchanged among nodes. Depending on the requirements posed by the application at hand, specific approaches may be more suited than others. At an extreme, specific solutions may not be applicable to a given application domain.

Classification

Four approaches emerge in the current WSN literature:

- **Database oriented:** systems like TinyDB [17] tend to regard the

sensor network as a relational database, and give programmers constructs to pose SQL-like queries.

- **Data sharing:** approaches such as Kairos [32] and Abstract Regions [14] are based on sharing data in the form of remotely accessible variables or tuples. Nodes can read or write data in the shared memory space using dedicated constructs.
- **Mobile code:** code migration is exploited in SensorWare [115, 116] and Agilla [18]. However, a small fraction of existing system provides *strong mobility* [117]: a mechanism whereby the code migrates along with the current status of execution, including the program counter. Often, a data sharing mechanism is employed under the covers for coordination.
- **Message passing:** solutions such as nesC [28] and DSWare [19] are based on exchanging messages among the nodes. Messages may represent raw sensed data or higher-level, logical events obtained from various stages of processing.

A natural illustration of database-oriented approaches is the TinyDB system, described in Section 2.5.2. In TinyDB, sensed data are made available as entries of a `sensors` table. The user accesses the table using SQL-like queries. Abstract Regions, illustrated in Section 2.5.1, is an example of data-sharing model. Programmers use dedicated constructs to export local data as shared variables, and to read other nodes' data by addressing the information of interest based on unique keys. To describe the remaining classes of distribution models, here we illustrate Agilla [18] as an example of mobile code system, and DSWare [19] as representative of message passing approaches.

Mobile code: Agilla

Overview. Agilla is a middleware system for WSNs adopting a mobile code paradigm. Programs are composed of one or more software agents able to migrate across nodes. Differently from alternative mobile code systems for WSNs, Agilla provides strong mobility. In this sense, an Agilla agent is similar to a virtual machine with its own instruction set and dedicated data/instruction memory. Coordination among agents is accomplished using a Linda-like tuple space [29]. Agents can insert data in a

```
BEGIN   pushn fir
        pusht LOCATION
        pushc 2
        pushc FIRE
        regrxn  // Register fire alert reaction
        wait    // Wait for reaction to fire
FIRE    pop
        sclone  // Strong clone to the node detecting fire
        ...     // Fire tracking code
```

Figure 2.15.: Fire tracking with Agilla (adapted from [18]).

local data pool to be read by different agents at later times, using a pattern matching mechanism to identify the data of interest. The use of tuple spaces allows to decouple the application logic residing in the agents from their coordination and communication. They also provide a way for agents to discover the surrounding context.

Example. Consider a fire monitoring application in a forest. Lightweight fire-detection agents are deployed to monitor the temperature in various regions. When a rise in temperature is detected, these spawn fire-tracking agents that swarm around to collect information about the exact location of the fire.

To implement a similar application, Agilla provides APIs to interact with the tuple space at each node, and to clone agents. As for the former aspect, Agilla provides operations to insert, read, and remove tuples. In addition, it gives programmers the ability to add *reactions* to its tuple space. These express the interest of an agent in a specific piece of data. When a tuple matching the reaction is inserted in the tuple space, the agent that previously installed the reaction is asynchronously notified. Operations to clone agents are named `smove`, `wmove`, `sclone`, and `wclone`. The `w` and `s` in front of the operation name specifies whether strong or weak mobility is required. Weak operations only migrate code, and the execution resumes from the beginning. Moving entails migrating state and code, and resuming execution on a new different node. Cloning instead resumes execution on both the old and new node.

Figure 2.15 shows how a fire tracking agent is notified about the presence of a rise in temperature. When such an agent is injected into a node, it registers a reaction for `FireAlert` tuples and waits for it to fire. This occurs when a fire detection agent outputs the corresponding tuple in the tuple space. Upon triggering of the reaction, the agent immediately clones itself

```
INSERT INTO EVENT_LIST
 (EVENT_ID, RANGE_TYPE, DETECTING_RANGE,
  SUBEVENT_SET, REGISTRANT_SET, REPORT_DEADLINE,
  DETECTION_DURATION [, SPATIAL_RESOLUTION ])
```

Figure 2.16.: Subscription format in DSWare (adapted from [19]).

to a different node node. Once there, it will continue to clone to gather information in regions around the phenomena.

Implementation highlights. Agilla is implemented on top of TinyOS. The instruction set and the mechanisms enabling on-the-fly execution of code are based on Maté. On every node, an agent manager maintains each agent's context, allocates memory when the agent arrives, and deallocates the same memory space when the agent leaves or dies. The latter aspects are dealt with using a lightweight implementation of dynamic memory, as this functionality is not originally available in TinyOS. The context manager determines the node location and maintains the list of reachable neighbors, whereas the tuple space manager implements the operations to read/write from/to the tuple space, and registers/triggers reactions when required. Most importantly, migrating agents requires reliable transmissions. This is achieved using a hop-by-hop retransmission scheme whereby messages not yet acknowledged are re-sent upon expiration of a timeout.

Message passing: DSWare

Overview. DSWare is a message passing middleware system whose focus is the real-time detection of sporadic events. It employs a form of Publish/Subscribe [118] in which users specify subscriptions based on the characteristics of the phenomena of interest. A higher-level notion of event is provided that programmers can use to infer the occurrence of specific phenomena from the raw sensor observations. For instance, an event can be defined as the composition of two physical sub-events occurring within a specific time interval one from the other. Confidence levels can also be defined to specify the relationships among sub-events with other factors that potentially affect the decision on the higher-level observation, e.g., the relative importance of sub-events, or their fitness to a known pattern.

Subscriptions are expressed using a dialect of SQL from the user base-station, according to the structure in Figure 2.16. Besides the event iden-

tifier, RANGE_TYPE and DETECTING_RANGE specify the group of sensors responsible for detecting the event. The corresponding notification is reported before the REPORT_DEADLINE to every node in the REGISTRANT_SET. DETECTION_DURATION instead specifies the total duration of this subscription, whereas SPATIAL_RESOLUTION determines the geographical granularity for the event's detection. Finally, the SUBEVENT_SET specifies a group of sub-events that must occur for this event to be observed, their timing constraints and confidence levels.

Example. Consider an application to detect explosions in a given geographical area. Temperature, light, and acoustic sensors are deployed to accomplish the task. We can therefore define a sub-event occurring when the temperature is higher than a safety threshold, a light sub-event corresponding to a sharp change in the light intensity, and an acoustic sub-event representing the occurrence of a sound whose signature bears similarities with those known for explosions. The higher-level event corresponding to an explosion is defined as the combination of the aforementioned sub-events when occurring within a specified time interval and within a given geographical region.

Implementation highlights. Subscriptions are propagated in the network until they reach the area of interest. In doing so, a routing tree is built connecting the base-station issuing the subscription to all sensor nodes in charge of observing the phenomena at hand. Two optimizations are performed in case multiple nodes subscribe for the same information. First, in case the subscriptions are for the same data yet they have different rates, DSWare places copies of the relevant information at intermediate nodes to limit the net amount of information flowing in the network. Second, DSWare tries to merge the paths leading to different base-stations to minimize redundant transmissions [119]. To guarantee real-time delivery of event notifications, an earliest deadline first scheduling mechanism is employed. An alternative, energy-aware scheduling technique is also provided, although it may occasionally fail in meeting the requested deadlines.

2.6. Architectural Aspects

In this section, we describe the relationships among the programming solutions themselves and with the rest of the conceptual blocks we identified in Section 2.2.

2.6.1. Composability

Here we look at whether a given programming solution can work alongside other programming approaches. Indeed, despite most of the solutions in the current state of the art cannot be combined together, researchers recently begun observing that having smaller, composable building-blocks could be a way to tackle the complexity in developing WSN applications [120].

Classification

Based on the above observation, it is possible to make a distinction along the following lines:

- **Holistic approaches** like Kairos [32] and snBench [114] are designed as general-purpose solutions to enable the development of a wide range of applications. By the same token, they are intended to be used as the only programming platform for a given application scenario, and are therefore unable to work in combination with other approaches.
- **Building-blocks** such as Hood [21] and Generic Role Assignment [20], instead, are conceived to solve very specific problems, and to be used in conjunction with additional programming solutions to tackle different issues.

The majority of the existing WSN programming platforms fall in the former class. For instance, the TinyDB [17] system, illustrated in Section 2.5.2, covers a large spectrum of applications, yet it does not leave any room for being used in collaboration with other programming abstractions. Instead, as example of building-block solution, in the following we describe Generic Role Assignment [20, 121].

Building-block: Generic Role Assignment (GRA)

Overview. GRA aims at tackling the specific problem of self-configuring WSN devices depending on programmer-specified requirements, while leaving concerns like data collection and dissemination to other, complementary solutions. To address the configuration issue, the authors present a role specification language and distributed algorithms to implement the required role assignments. A role specification is a list of role-rule pairs. For each possible role, the corresponding rule describes the conditions for

```
ON :: {
  temp-sensor == true &&
  battery >= threshold &&
  count (2 hops) {
    role == ON &&
    dist(super.pos, pos) <= sensing-range
  } <= 1 }
OFF :: else
```

Figure 2.17.: Role specification for the coverage problem (adapted from [20]).

this role to be assigned to the local node. Rules are expressed as boolean predicates that either refer to the properties of the node considered (e.g., the remaining energy or its geographical location), or to the properties of other nodes within a given number of hops (e.g., how many temperature sensors are reachable within 3 hops).

Example. Here we describe how GRA can be used to solve the coverage problem. A certain geographical area is said to be covered if every physical point in this area lies within the observation range of at least one sensor node. If nodes are densely deployed, redundant nodes can be turned off to save energy. However, when active nodes run out of power, we must switch the redundant nodes on again.

The above application essentially requires a proper assignment of the roles `ON` and `OFF` to WSN nodes. Figure 2.17 shows the corresponding role specification. For a given node to take the `ON` role, it must have a temperature sensor, a minimum battery level, and at most another node with role `ON` must be found within the node's sensing range. The latter condition is specified using the `count` operator. This takes as input a number of hops and the returns the number of nodes within such range matching the specification in curly braces.

Implementation highlights. All nodes in the network are provided with all role specifications. Based on those, they evaluate how many hops they need to push their local information so to guarantee other nodes are able to evaluate their rules. To account for changing node properties and network dynamics, the role specifications are periodically re-evaluated. In the former case, a node re-evaluates the specification only if the change may affect its or some other node's role. As for the latter, distributed protocols are provided to recognize when the nodes join or fail.

2.6.2. Reach

Differently from the previous dimension, in this section we look at how deep a programming abstraction can serve into our reference architecture. This usually occurs opposed to an increase in the level of abstraction. For instance, solutions presenting lower levels of abstraction often provide support for implementing system services besides full-fledged applications.

Classification

The above observation suggests the following classification:

- **Application-level solutions** are those that can be applied only at the level of end-user applications, i.e., the topmost layer in Figure 2.1. Examples are Pleiades [15] and Cougar [113].

- **Vertical solutions**, instead, are able to express the processing required at various levels of our reference architecture, which we illustrated in Section 2.2. Examples in this category are nesC [28] and Hood [21]. Indeed, these approaches can be used, for instance, to implement localization mechanisms and MAC protocols besides end-user applications.

Similarly to the previous dimension, the majority of research efforts fall in the first category. For instance, the Pleiades system, described in Section 2.5.1, does not easily allow the specification of system services due to the high level of abstraction provided. Indeed, the mechanisms germane to lower levels of the reference architecture, e.g., localization mechanisms, are more easily expressed using physical neighborhood communication and node-centric computation. In fact, these approaches implicitly convey to the programmer the notion of geographical co-location of nodes, while giving easy access to low-level facilities in the node hardware. In this category, a notable example is Hood [21], described next.

Vertical solution: Hood

Overview. Hood enables a notion of neighborhood as a programming primitive in nesC. Constructs are provided to identify a subset of a node's physical neighbors based on application criteria, and to share state with

```
generate attribute LightAttribute from int;

generate neighborhood LightHood {
  wire filter LightThrehshold;
  set max_neighbors to 5;
  reflection LightRefl from LightAttribute;
}
```

Figure 2.18.: Reading light values using Hood (adapted from [21]).

them. A node exports information in form of *attributes*, which are defined by the programmer at compile-time. Membership in a programmer-specified neighborhood is determined using *filters*. These are boolean functions that examine a node's exported attributes and determine, based on their values, whether the remote node is to become part of the considered subset. If so, a *mirror* for that particular neighbor is created on the local node. The mirror contains both *reflections*, i.e., local copies of the neighbor's data that can be used to access the shared data, and *scribbles*, which are local annotations about that neighbor. As different functionality may require different criteria to establish membership in a neighborhood, Hood allows for multiple, independent neighborhoods. The complexity of node discovery and data sharing is automatically dealt with by the underlying Hood run-time. Notably, the neighborhood concept in Hood is applicable to a range of mechanisms, from target tracking applications to localization mechanisms and MAC protocols [21].

Example. Consider a simple application where nodes monitor light readings. Figure 2.18 depicts a fragment of Hood code to define a neighborhood including light sensors whose current reading is above a specified threshold.

The `generate` construct is used either to define an `attribute`, or to declare a new `neighborhood`. In the former case, our example simply creates a `LightAttribute` out of an integer value. The `LightHood` neighborhood is instead created by specifying what filter is to be used for establishing membership in this neighborhood (`LightThreshold`), the maximum number of members of this neighborhood (`5`), and the specific attribute mirrored on the local node. The former is supposed to be provided as a nesC module implementing a standard interface.

A simple API is given to programmers to interact with the neighborhood in the application code. For instance, nesC commands are provided to iterate through the current members of a given neighborhood, and access

their local mirrors. Differently, nesC events are defined to fire when the value of a locally mirrored attribute changes.

Implementation highlights. The Hood constructs to define attributes and neighborhoods are given as input to a dedicated pre-processor that outputs plain nesC code. The underlying distributed implementation is simply based on periodic 1-hop broadcasting and filtering on the receiver side. This also servers for neighbor discovery and maintenance. In case the application wants to control the spreading of local information directly, e.g., because of a sudden increase in a sensed value, it can require broadcasting the local attributes on demand.

2.6.3. Stack Penetration

Although a given programming solution may not lend itself to the description of mechanisms below the application level, some of the existing solutions provide hooks into the lowest layers to give applications the ability to tune the system behavior depending on changing requirements.

Classification

Taking the above observation into account, it is worth making the following distinction:

- Solutions such as Cougar [113] and Kairos [122] completely **hide** all low-level concerns related to distribution. By doing so, they relieve programmers from the burden of handling gory details deep in the the stack. The same solutions, however, may miss optimizations obtained by letting the application requirements percolate down through the architecture.

- Approaches like MiLAN [123] and Abstract Regions [14] provide *hooks* into the lowest layers to control the behavior of physical devices such as the radio, while trying to retain a reasonable high level of abstraction. This is dual to the previous class of solutions: programmers can perform fine-grained adaptation on various system aspects, e.g, changing the number of retransmissions per message, yet they are somewhat forced to mix high-level programming constructs with low-level concerns. Similar approaches therefore allow programmers to explore the trade-off between accuracy and resource-consumption,

although the effect of fine-grained tunings on overall system behavior is generally hard to predict.

Almost all available approaches featuring global view communication or system-centric computation fall in the former category. An example to illustrate the latter class of solutions is Abstract Regions, described in Section 2.5.1. It provides a notion of *quality feedback* to give the application information on the accuracy or completeness of a given operation. Based on this information, the programmer can leverage off a *tuning interface* to gain access to the underlying communication protocols, and change their behavior depending on whether the application requirement are currently satisfied. For instance, in case reduction operations have a low yield due to message losses, the application can increase the number of retransmissions per packet until the data received satisfies the user's needs. By the same token, however, this choice increases network traffic, and therefore affects the overall energy consumption.

2.6.4. Supported Platforms

Despite the plethora of available programming approaches, the range of supported platforms, as reported in the current literature, is surprisingly narrow.

Classification

The following trends can be noted in the current state of the art:

- **Real hardware:** at the operating system layer, most higher-level abstractions are built by adding a pre-processing step to nesC, e.g., as in EnviroSuite [102]. Thus, they leverage off TinyOS as operating system, and should therefore support any TinyOS-compliant WSN platform.
- **Simulated environments:** often, simulated environments are used to assess the performance of a given solution. In this case, either custom simulators or TOSSIM [124] are used. Early performance studies, e.g., [19], have also been carried out using ad-hoc network simulators, such as NS-2 [125] or GlomoSim [126].

Nonetheless, experiments using real nodes are seldom reported in the literature. This is essentially due to i) the lack of portability between different

hardware platforms, ii) the difficulty in setting up reproducible experiments using real hardware. As for the first issue, for instance, although nesC code may compile seamlessly for different target nodes, modifications are usually needed to account for the very characteristics of the devices employed, notably including the radio transceiver. Instead, the difficulties in reproducing real-world experiments are mostly due to the characteristics of the wireless medium. For instance, the radiation patterns may change over time because of multiple reasons, ranging from temperature and humidity fluctuations to the presence of people nearby. In this respect, more effort is required from a methodological standpoint.

2.7. Completing the Picture

Before introducing the mapping from currently available solutions to our dimensions for classification, we briefly survey relevant approaches that we did not consider in the previous sections.

Cougar. Similarly to TinyDB, Cougar [113] provides programmers with a SQL-like interface for querying the sensor network. Therefore, they are both geared towards data collection applications, while lacking constructs to express fine-grained control flows. At the system level, they are both based on a routing tree rooted at the user base station. The techniques used to achieve energy efficiency are, however, different. For instance, Cougar tries to push selection operators down the routing tree to reduce the amount of data flowing up, yet it does not consider acquisitional issues, as TinyDB does.

EnviroSuite/EnviroTrack. The programming solution presented in [102, 127] is explicitly conceived for tracking applications. An object-oriented programming model is provided where objects represent physical phenomena. The programmer express the conditions for creating these objects (e.g., an object is created when a magnetometer reports a possible presence of a moving vehicle), and associates operations to process the corresponding data (e.g., inferring the size of the vehicle based on known acoustic patterns). Dedicated protocols are provided and used depending on the nature of the phenomena being observed, e.g., its speed of movements. The compiler infers the nature of the phenomena from the object conditions, or from programmer-provided information.

FACTS. In [101], the authors present a rule-based programming model in-

spired by logical reasoning in expert systems. Data is represented as facts in a dedicated repository. The appearance of new facts trigger the executions of one or more rules, that may generate new facts or remove existing ones. Native C functions can be used to interact with the sensing devices and provide inputs to the creation of new facts. Facts can be shared among different nodes. The basic communication primitives provide 1-hop data sharing, although multi-hop protocols can be employed in collaboration with the basic rule engine.

Flask. A data-flow language is presented in [128], whose programming model revolves around the notion of data-flow operator. This is a computational unit taking multiple inputs, and producing a single output value. Programs in Flask are composed by wiring operators in an acyclic graph. The control flow migrates across operators in a depth-first manner. Different operators can be located on different nodes, and interconnected using a form of Publish/Subscribe communication infrastructure. The underlying routing scheme can be changed by programmers on a per-application basis. Flask programs are completely executed at compile-time, yielding executable nesC code as output. Interestingly, Flask is also designed for building higher-level abstractions in terms of data-flow operators, e.g., FlaskDB [128].

Market-based programming. The authors of [129] present a framework for expressing system-centric computation using a market oriented approach. The objective is to obtain a globally efficient behavior under dynamic conditions. Every node is characterized by the actions it can take, the corresponding cost, and a possible reward the node receives in exchange for performing a given action. To maximize its own profit, nodes autonomously decide which actions to take based on the current rewards, the actions' cost, and the surrounding context. The user induces the desired behavior by dynamically changing the rewards given to nodes for each type of action. Communication is delegated to a specific routing scheme chosen by programmers depending on the application's requirements.

MiLAN. The work in [123] provides an explicit notion of data quality to programmers. The application specifies the expected reliability for each input sensor providing raw data, as well as the minimum requirements that the high-level information must meet in terms of rates and confidence. Based on how the high-level information is derived from the raw data, MiLAN identifies the minimum set of sensors that need to be queried for

satisfying the application needs while minimizing energy consumption. The underlying communication protocols are decoupled from the optimization logic. Therefore, they are expected to offer the appropriate hooks so that the information inferred by MiLAN on the minimal set of sensors needed can be exploited also at the lowest layers.

Pieces. Focusing on collaborative applications where multiple, geographically related data must be processed, the solution in [109] provides a programming abstraction based on a notion of group. Similarly to Abstract Regions, system support must be provided on a per-group basis. Constructs are provided to determine the role of nodes in a group (e.g., to determine when a node is to be elected as the leader), and to share information. The application processing is expressed in discrete steps as input/output operations on state variables. The inputs are governed by sensed data, whereas the outputs are the result of some processing based on the previous values of state variables and the current inputs.

TAG. Along the same lines of Cougar and TinyDB, TAG [130] is based on a dialect of SQL to let programmers query the sensor network. At the implementation level, TAG also leverages off a routing tree. However, it focuses on how to achieve energy efficiency in aggregation queries. To do so, TAG employs information on the mathematical properties of the aggregation function at hand, and adapts message routing accordingly. For instance, in case not all input data are needed to compute the aggregated, some nodes are excluded from the query, or their values are inferred from those sensed at other, nearby nodes.

SensorWare. Similarly to Agilla, SensorWare [115, 116] allows to move TCL-based scripts from node to node, while providing explicit support for multiple applications running concurrently on the same sensor network. Nonetheless, it only provides weak mobility, i.e., the execution state does not move with code, which results in the execution always resuming from the beginning on a different node. Differently from other mobile code solutions, coordination is accomplished using direct communication across different nodes instead of shared memory spaces. When migrating code, policies regarding the energy required by a given script can be specified to determine the acceptance of the same script on a node. The current implementation targets fairly powerful devices, e.g., PDAs like Compaq iPAQs or embedded devices with XScale processors.

SINA. Compared to TinyDB, the work in [131] overcomes the limitation of

SQL dialects using an imperative language called SQTL, that enables the injection of arbitrary code into the network. Support is provided to export the outcome of SQTL-based processing as query results. This enables the description of more sophisticated coordination patterns w.r.t. pure SQL. For instance, cluster-based information aggregation can be implemented using SQTL, and the results gathered using a pure SQL query.

snBench. The work in [114] targets shared, multi-user sensor networks, and exports a strongly-typed, functional language to express the application processing. Partially deviating from the functional idiom, however, repetitions and assignments to local variables are also allowed. To implement the programming model, a central entity keeps track of the current status of each host in the system, and injects single processing units in the network accordingly. In doing so, the individual processing units are optimized to take advantage of programs' shared computation and dependencies, with the ultimate goal of making more efficient use of computation, network, and storage resources. Sensors are named via URI relative to the host they are connected to. Therefore, processing is assumed to occur on fairly powerful nodes controlling several sensors, as opposed to mote-class devices usually equipped with a few sensing devices.

Spatial Programming. In [132], the authors propose a logical addressing scheme coupled with a lightweight form of mobile code technology. In Spatial Programming, the environment is viewed as a single address space, and nodes are accessed using spatial references. This refers to the expected physical location of a node (e.g., `hill:camera[0]`), and may optionally point to some property of the node itself (e.g., whether the node is currently active). A dedicated run-time system maintains the mapping from spatial references to the physical nodes. Smart Messages are used for migrating code and data across nodes. These represent a form of scripting language targeted to a virtual execution environment. A shared memory space is provided for coordination among Smart Messages, and also used to determine how to route a smart message at each hop, thus allowing to change the routing policy dynamically.

2.8. Mapping and Discussion

Figures 2.19 and 2.20 illustrate the overall mapping from the programming solutions hitherto described to our taxonomy.The former concentrates on

Programming Abstraction	Communication	Computation	Programming Idiom	Distribution Model
Abstract Regions [14]	Group based, Multi-hop non-connected	Node centric	Imperative	Data sharing
Abstract Task Graph [108]	Global view	Node centric	Hybrid	Data sharing
Agilla [18]	Routing dependent	Node centric	Imperative	Mobile code
Cougar [113]	Global view	System centric	Declarative	Database oriented
DSWare [19]	Global view	System centric	Declarative	Message passing
EnviroSuite/ EnviroTrack [102, 127]	Group-based, Multi-hop connected	Group centric	Imperative	Data sharing
FACTS [101]	Physical neighborhood	Node centric	Declarative	Data sharing
Flask [128]	Routing dependent	Node centric	Functional	Message passing
Generic Role Assignment [20, 121]	Group-based, Multi-hop connected	Node centric	Declarative	Data sharing
Hood [21]	Group-based, Single-hop	Node centric	Imperative	Data sharing
Kairos [32]	Global view	Node centric	Imperative	Data sharing
Market-based programming [129]	Routing dependent	System centric	Declarative	Message passing
MiLAN [123]	Physical neighborhood	Group centric	Imperative	Message passing
nesC [28]	Physical neighborhood	Node centric	Imperative	Message passing
Pieces [109]	Group based, Multi-hop connected	Group centric	Imperative	Data sharing
Pleiades [15]	Global view	Node centric	Imperative	Data sharing
Regiment [16, 110]	Group based, Multi-hop connected	Group centric	Functional	Data sharing
SensorWare [115, 116]	Physical neighborhood	Node centric	Imperative	Message passing/ Mobile code
Spatial Programming [132]	Group based, Multi-hop non-connected	Group centric	Imperative	Mobile code
SINA [131]	Global view	System centric	Hybrid	Database oriented
snBench [114]	Group based, Multi-hop non-connected	Group centric	Functional	Data sharing
TAG [130]	Global view	System centric	Declarative	Database oriented
TinyDB [17]	Global view	System centric	Declarative	Database oriented

Figure 2.19.: Mapping WSN programming abstractions to the taxonomy in Figure 2.2—language aspects.

Programming Abstraction	Composability	Reach	Stack Penetration	Supported Platforms
Abstract Regions [14]	Holistic	Application level	Tuning interface	TOSSIM
Abstract Task Graph [108]	Holistic	Application level	No	N/A
Agilla [18]	Holistic	Application level	No	Mica2
Cougar [113]	Holistic	Application level	No	NS-2
DSWare [19]	Holistic	Application level	No	GlomoSim
EnviroSuite/ EnviroTrack [102, 127]	Holistic	Application level	No	Mica2, XSM
FACTS [101]	Holistic	Vertical solution	No	ESB, ScatterWeb
Flask [128]	Holistic	Vertical solution	No	TMote Sky
Generic Role Assignment [20, 121]	Building-block	Vertical solution	No	Custom simulator
Hood [21]	Building-block	Vertical solution	On-demand attribute broadcast	Mica2
Kairos [32]	Holistic	Application level	No	Mica2, Mica2Dot
Market-based programming [129]	Holistic	Application level	No	TOSSIM
MiLAN [123]	Holistic	Application level	Data quality guarantees	N/A
nesC [28]	Holistic	Vertical solution	Active Message interfaces	All TinyOS hw platforms
Pieces [109]	Holistic	Application level	No	Custom simulator
Pleiades [15]	Holistic	Application level	No	TelosB
Regiment [16, 110]	Holistic	Application level	No	Custom simulator
Spatial Programming [132]	Holistic	Application level	Node properties	iPAQ
SensorWare [115, 116]	Holistic	Application level	No	iPAQ, XScale
SINA [131]	Holistic	Application level	No	Custom simulator
snBench [114]	Holistic	Application level	No	N/A
TAG [130]	Holistic	Application level	No	Custom simulator
TinyDB [17]	Holistic	Application level	No	Mica2

Figure 2.20.: Mapping WSN programming abstraction to the taxonomy in Figure 2.2—architectural aspects.

the language aspects, highlighting the distinctive traits of the different approaches. Interestingly, the following patterns emerge:

- The way computation is expressed bears some influence on the programming idiom employed. System-centric solutions privilege declarative approaches, whereas approaches featuring group- and node-centric computation mostly leverage off functional and imperative approaches, respectively. Indeed, interactions at the system level are easier to describe in a declarative fashion, taking advantage of several compilation steps to obtain the executable code to be deployed on the individual nodes. Differently, when computation affects a limited portion of the system, functional approaches are suited to describing the processing of information by assuming data sensed in a group as the input, and directing the output to a single node. Finally, imperative approaches are most intuitive when the processing affects individual nodes. Indeed, this programming style closely resembles the traditional way of developing distributed applications.

- Similarly, a relation exists between how communication is described, and the distribution model employed. Generally, message passing and mobile code are employed in conjunction with physical neighborhood or group based communication. The larger is the scope used to describe communication, however, the more data sharing and database approaches are privileged. It would indeed be unreasonable to give programmers high-level views on the communication occurring in the network, and still ask them to reason in terms of messages flowing from a specific sender to a given destination.

Instead, Figure 2.20 provides a high-level view on the architectural aspects of currently available solutions. The following trends can be noted:

- Compared to the whole spectrum of available approaches, there are very few programming abstractions that are designed to work in collaboration with other solutions. The diversity of WSN applications, however, is likely to require an overarching approach where different programming abstractions collaborate into a single, coherent framework to achieve a high-level goal [120]. Although most of the existing approaches are well suited to particular application domains, they lack extensibility as they cannot be composed with additional solutions to tackle different issues.

Programming Abstraction	SO/SR	Interaction Pattern	Space	Time
Abstract Regions [14]	SO	Depending on region	Local	Periodic/ Event-driven
Abstract Task Graph [108]	SR	Many-to-many	Local	Periodic
Agilla [18]	SO	Many-to-many	Local	Event-driven
Cougar [113]	SO	Many-to-one	Global	Periodic
DSWare [19]	SO	Many-to-one	Local	Event-driven
EnviroSuite/ EnviroTrack [102, 127]	SO	Many-to-one	Local	Event-driven
FACTS [101]	SO	Many-to-many	Local	Event-driven
Flask [128]	SO	Many-to-many	Global	Periodic/ Event-driven
Generic Role Assignment [20, 121]	SO	Many-to-many	Global	Periodic
Hood [21]	SO	Many-to-many	Local	Periodic
Kairos [32]	SO	Many-to-many	Global	Periodic
Market-based programming [129]	SO	Many-to-one	Global/ Local	Periodic
MiLAN [123]	SO	Many-to-one	Global	Periodic
nesC [28]	SO	Many-to-many	Local	Periodic/ Event-driven
Pieces [109]	SO	Many-to-one	Local	Event-driven
Pleiades [15]	SO	Many-to-many	Global	Periodic
Regiment [16, 110]	SO	Many-to-one	Local	Event-driven
Spatial Programming [132]	SO	Many-to-one	Local	Event-driven
SensorWare [115, 116]	SO	Many-to-one	Global	Periodic
SINA [131]	SO	Many-to-one	Global	Periodic
snBench [114]	SO	Many-to-one	Local	Event-driven
TAG [130]	SO	Many-to-one	Global	Periodic
TinyDB [17]	SO	Many-to-one	Global	Periodic

Figure 2.21.: Mapping WSN programming abstractions to the characteristics of WSN applications.

- Similarly, most of the existing solutions do not feature hooks into lower layers to adapt the system behavior dynamically. Most often, this choice is motivated by the difficulty by which programmers can predict the effect of some fine-grained tuning on the final system performance. By the same token, however, completely masking communication concerns seems to clash with the increasing demand for cross-layer solutions, in turn motivated by the resource scarcity of common WSN hardware.

Interestingly, we can draw further remarks by comparing Figure 2.19 against Figure 2.20:

- All vertical solutions we surveyed feature node-centric computation.

Indeed, the inherent tension between the level of abstraction used to express computation and the functionality needed to implement lower-level mechanisms, e.g., as in the case of system services, would prevent using group-centric or system-centric computation. Lower-level mechanisms usually need to work close to the hardware layers, which is precisely what group level and system level computation tend to avoid.

- Dually, approaches characterized by system-centric computation rarely feature hooks into the lowest levels of the stack. The level of abstractions provided by the former is usually too high for accommodating similar functionality without affecting the overall programming framework.

Finally, Figure 2.21 provides a mapping from existing programming solutions to the characteristics of WSN applications we identified in Section 2.3. Here, we aim at providing indications about which programming solution is most appropriate for a given application domain, while pointing out open issues and challenges. The fundamental remarks we can draw are summarized as follows:

- Looking at the current state of the art, most solutions support a many-to-one form of interaction. They are therefore appropriate for application such as habitat monitoring [22], that essentially revolve around pure data collection. When many-to-many interactions are supported, this usually occurs only within the boundaries of the physical neighborhood.

- Similarly, localized interactions seem to be better supported for event-driven applications, e.g., detection of moving targets [85]. Dually, a good fraction of available solutions is appropriate for describing periodic tasks when the application processing spans the entire system, as in data collection scenarios.

- As a result of the previous points, none of the existing solutions, with the notable exception of ATaG [108], seems appropriate for sense-and-react applications. As we already pointed out, these usually require many-to-many interactions spanning multiple hops, as well as continuous monitoring limited to specific portions of the system. The current state of the art seems ill-suited to these requirements.

In the following chapter, we further elaborate on the above observations, illustrating how we plan to tackle the corresponding issues in the rest of this thesis.

3. Beyond the State of the Art

Based on the conclusions drawn at the end of the previous chapter, we maintain that most of the research efforts in WSN programming have devoted themselves to sense-only scenarios. Nevertheless, the ability to affect the environment based on sensed data is pivotal for widening the range of possible applications. The contributions discussed in this thesis are mostly geared towards the latter application scenarios. In doing, so we cover multiple and diverse aspects of programming WSNs.

To give a broader look at our contributions, in this chapter we briefly revisit the high-level challenges we illustrated in Chapter 1, and cast our solutions in the taxonomy introduced in the previous chapter. The objective is to provide a common ground for understanding the work described in this thesis.

3.1. Open Problems

Based on the description of the current state of the art outlined in Chapter 2, we maintain the following issues are still to be solved in the context of WSN programming:

- **Software reconfiguration.** In the highly dynamic scenarios we target, changing single functionality running on the individual nodes is of paramount importance. Nonetheless, existing programming solutions featuring *node-centric* computation rarely provide support for identifying what part of the application functionality is to be changed, and how to carry out the reconfiguration process.

- **Coordination.** To save on latency and resource consumption, in sense-and-react applications the processing must be brought into the network, as opposed to residing at the fridges of the system on some powerful base-station. This requires strict coordination among the

devices involved. This issue is further exacerbated by the heterogeneity germane to the scenarios we target, where different types of sensors provide the inputs to control various classes of actuators.

- **Scoping.** In sense-and-react applications, multiple tasks must run concurrently, each affecting only a given part of the system. Therefore, programmers need to identify the relevant subsets of nodes, and express interactions among them. A similar feature is missing in the current state of the art. Available approaches either constrain the topological characteristics of the subset considered, e.g., as in programming solutions featuring *group-based* communication for *1-hop* and *connected* topologies, or do not provide the ability to address arbitrary subsets of nodes.

In this thesis, we tackle the problems above at different levels of abstractions, from programming in the physical neighborhood up to sensor network macroprogramming. The next section provides a high-level view on our contribution.

3.2. Contribution

Software reconfiguration. The RUNES and FiGaRo component models, described in Chapter 4, offer a foundation to enable software reconfiguration on single nodes, therefore providing support for changing the application behavior dynamically. Indeed, the component abstraction makes it easier to limit the scope of reconfiguration to the required functionality. Moreover, the FiGaRo component model promotes component dependencies and versions to first-class citizens in the programming model, giving developers an unprecedented degree of control on how to carry out the reconfiguration process.

Coordination. The TeenyLime middleware, described in Chapter 5, addresses the above challenges by providing a solution featuring *node-centric* computation and *physical neighborhood* computation. The latter is not at at all a limitation in the scenarios targeted by TeenyLime, where actuators are usually co-located with the sensor that trigger them. Moreover, the tuple space abstraction of TeenyLime, coupled with various WSN-specific features, provides a *data-sharing* distribution model that sensibly raises the level of abstraction w.r.t. traditional node-centric solutions.

Scoping. With the Logical Neighborhoods abstraction, described in Chapter 6, we empower programmers with a foundation to enable communication among application-defined subsets of nodes. Our dedicated routing layer, illustrated in Chapter 6, achieves group-based communication regardless of the physical location of the nodes involved, thus giving programmers the ability to define subsets of nodes irrespective of the underlying network topology.

Interestingly, Logical Neighborhoods can be exploited also as a building-block to empower system-level mechanisms or to design higher-level abstractions. In the last part of the thesis we explore this possibility at different levels of abstractions. Our contributions in this respect can be summarized as follows:

- **Fine-grained software reconfiguration.** Besides identifying what is to be reconfigured on the single nodes, in sense-and-react applications programmers must also determine where reconfiguration must occur, i.e., the subset of nodes required to change their behavior. Existing system-level mechanisms for code distribution, however, usually target the whole system by distributing the same binary image to all the nodes. To meet this requirement, we augment the FIGARO component model with a customized version of Logical Neighborhoods, giving FIGARO programmers the ability to target a specific subset of nodes based on their logical characteristics or current software configuration.

- **Virtual nodes.** Although Logical Neighborhoods provide the ability to identify the relevant subsets of nodes, expressing the very application processing still rests on the programmer's shoulders. Nevertheless, existing programming solutions targeting *group-based* communication rarely address sense-and-react scenarios. The virtual node abstraction, described in Chapter 9, is designed as a natural extension of Logical Neighborhoods. With virtual nodes, programmers abstract groups of physical devices into a single, fictitious node. This greatly masks distribution and heterogeneity, drastically simplifying application development. In addition, our dedicated routing layer, illustrated in Chapter 10, achieves significant performance improvements w.r.t. traditional communication schemes.

- **Scoping in macroprogramming.** Applying macroprogramming approaches in sense-and-react systems is usually difficult, as they

mostly target homogeneous scenarios where the application processing is the same on all the nodes. As a result, we miss the opportunity to exploit their high level of abstraction in the context of WSN applications involving actuation. In Chapter 11, we leverage off Logical Neighborhoods as a building block to enable a notion of scoping in an existing macroprogramming framework. The resulting programming model allows programmers to allocate a different processing to different group of nodes, and express complex interactions among these groups. Despite the high level of abstraction provided, the resulting system performance is still reasonable, thanks to a dedicated run-time layer we developed.

Part II.

Programming with Physical Neighborhoods

4. Component Models for Software Reconfiguration

As we discussed in Chapter 1, modern WSN applications require programmers to deal with a dynamic set of requirements that may evolve due to unanticipated needs. Therefore, when the application goals change, the system must be able to adapt its behavior accordingly. In these scenarios, the ability to reconfigure portions of the software running on WSN nodes becomes imperative.

To address this issue, here we describe two contributions in the field of component models for WSN programming. Compared to existing alternatives, our solutions focus on reconfigurability as the chief design criteria. On one hand, we tackle an extreme form of heterogeneity with the RUNES *middleware: a platform-independent programming model able to scale from WSN-class devices to powerful nodes such as PCs or PDAs. On the other hand, we move past RUNES and existing works in the field of WSNs with* FiGaRo*: a component model whereby component dependencies and versions become first-class citizens in the programming framework, and the entire reconfiguration process occurs with no intervention from the programmer. The contributions reported in this chapter appeared in [4–6].*

4.1. Scenario

Modern WSNs must support very challenging application scenarios. For instance, consider an application for control and monitoring in road tunnels [133]. Sensors monitor environmental conditions such as temperature, humidity and air quality. Actuator devices operate tunnel safety systems such as sprinklers, fans, and traffic signs. The system also incorporates larger, more powerful, devices which act as gateways and allow the sensors to report readings both directly to the actuator systems and to a tunnel

control centre.

When an accident occurs, the system's first responsibility is to detect and report the accident and carry out any automated emergency sequences. In addition, some nodes may be damaged and the system must reconfigure itself to compensate for this. As the situation unfolds, a team of firefighters eventually arrives. They carry PDA-class devices capable of interacting directly with the tunnel system. At this point, the tunnel system plays the role of a tool that can be directly manipulated by the firefighters. For instance, it can be selectively queried by the firefighters to help them operate in poor visibility conditions, and the firefighters can directly control the actuator devices.

The distinctive traits of the scenario outlined above can be summarized as follows:

- The system is highly *heterogeneous*. Different devices play different roles in the application, e.g., sensors act as data producers whereas actuators function as data consumers. The nodes may be equipped with different sensing/actuation devices, and be externally powered as opposed to running on batteries. Moreover, the hardware platforms range from tiny sensors to controller PCs and the PDA-class devices carried by the firefighters.

- Devices are *resource poor*. Apart from the obvious issues of power and CPU speed, memory can be a significant limitation that can severely constrain the amount of processing nodes can handle, or their capability to buffer messages.

- Finally, such scenarios are inherently *dynamic* due to changing environmental conditions. For instance, firefighters need to spontaneously create new patterns of interaction as they move around in the tunnel. Such situations require the system to be capable of dynamically loading new functionality onto devices, and offloading functionality as resources dwindle.

4.2. Motivation and Contribution

When it comes to dynamically reconfigure the WSN software in such challenging scenarios, several issues must be addressed. For instance, programmers must identify which functionality is to be reconfigured, and express

the processing to let the new functionality cooperate with those already running on a node. In addition, they must make sure that in case a new piece of functionality does not reach all the interested nodes, backward compatibility can be still be maintained. Finally, programmers must adopt the appropriate mechanisms to identify malicious code being uploaded to the WSN nodes, and force the system not to load the corresponding functionality.

Among the issues mentioned above, programming support is the basic building-block that enables programmers to build highly-reconfigurable systems. In this field, we investigate the use of component-based programming with a focus on the ability of dynamically reconfiguring a node's functionality. Our contribution is twofold:

- The **RUNES** middleware, described in Section 4.3, is based on a two-level architecture: the foundation is a language-independent, component-based programming model that is sufficiently minimal to run on any of the devices typically found in pervasive embedded environments. Above this is a layer of software components that offer the necessary application functionality and can be selectively deployed according to current resource constraints and application needs. In this context, our work specifically concentrated on WSN-class devices. The RUNES middleware, however, lies at the core of an overarching hardware/software architecture [134] whose aim is to tackle an extreme form of heterogeneity, spanning tiny embedded devices as well as powerful nodes such as laptops and PDAs. Section 4.4 describes how this cross-platform approach enables a broader look at the issues arising in our motivating scenario, by giving developers a single programming platform to tackle different concerns at once. Here we also report on the implementation of the RUNES programming model for non-WSN nodes, and provide a unified evaluation of the system performance in Section 4.5.

- The **FiGaRo** component model, illustrated in Section 4.6, moves past the RUNES middleware in the sensor network context. Differently from other component models for WSNs (e.g., [28]), FiGaRo provides dedicated constructs to deal with component dependencies and versions. Our run-time support, described in Section 4.7, makes sure new components are installed only when the corresponding dependencies are satisfied. Moreover, in FiGaRo we simplify the reconfiguration process itself by automatically changing the component

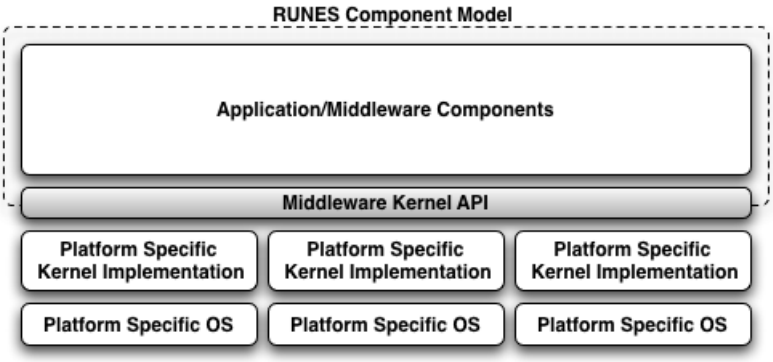


Figure 4.1.: The RUNES software architecture.

interconnections, thus relieving programmers from specifying how reconfiguration must be carried out. Our evaluation of the overhead incurred by the component model shows a very limited overhead w.r.t. native implementations, as illustrated in Section 4.8.

We conclude the chapter in Section 4.9 by comparing our approaches with the current state of the art.

4.3. The RUNES Middleware Foundation

In this section, we first describe the RUNES component model and its associated API. As shown in Figure 4.1, this API is provided at run-time by the middleware kernel. We then discuss the middleware kernel implementation for three different hardware platforms and briefly comment on components we have developed to tackle the issues in our reference application.

4.3.1. Component Model

The RUNES component model comprises the following elements: *components*, *component types*, *interfaces*, *receptacles*, *connectors*, *connector factories*, *attributes* and *capsules*. The API associated with the model is defined in Figure 4.2 in terms of the OMG's Interface Definition Language (IDL). In addition, the relationships between the various elements is shown diagrammatically (using UML) in Figure 4.3.

```
interface Capsule : {
  ComponentType load(in Pattern p);
  void unload(in ComponentType t);
  Component instantiate(in ComponentType t);
  void destroy(in Component c);
  Connector connect(in Interface i, in Receptacle r,
                    in ConnectorFactory cf);
  void setAttribute(in Entity e, in Attribute a);
  sequence<Attribute> getAttributes(in Entity e,
                                    in Pattern p);
  sequence<Entity> getEntities(in Pattern p);
};
```

Figure 4.2.: The Kernel API.

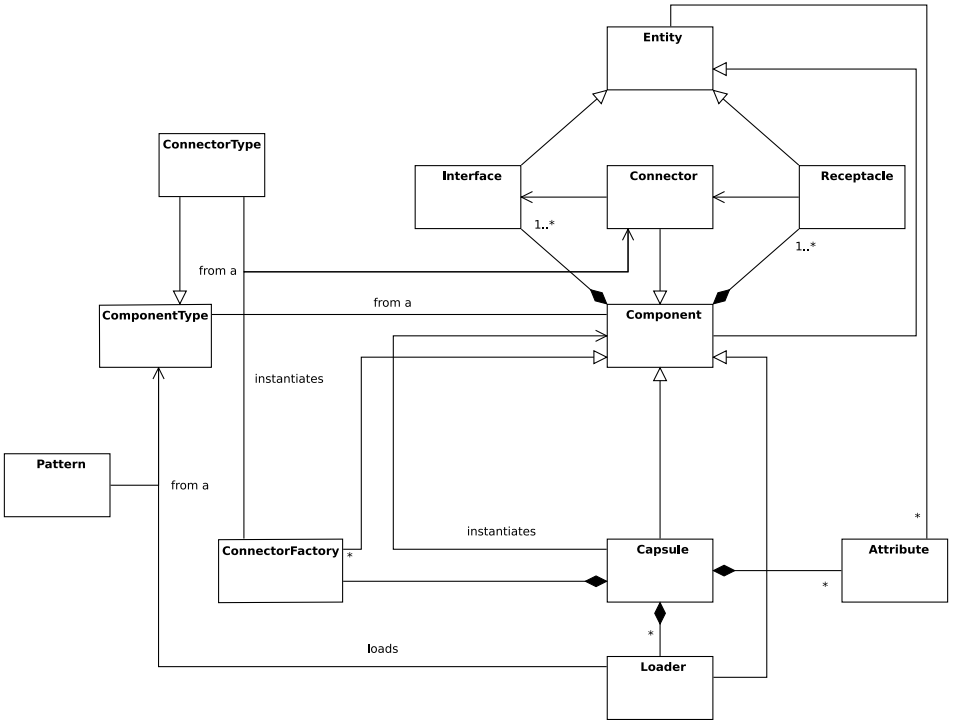


Figure 4.3.: The RUNES component model.

In the model, *components* are the basic run-time units of encapsulation and deployment. They are instantiated at run-time from *component types*,

such that each component type can be used to create multiple component instances at run-time. This is performed using the `instantiate()` operation in Figure 4.2. Components can be deleted as well, using `destroy()`. Component types can themselves be dynamically loaded and unloaded at run-time (see `load()` and `unload()`), which provides the basis for the dynamic nature of the RUNES programming model[1]. The semantics associated to the internal processing and states of a component is generally application-specific. Programmers should only pay attention *not* to share state explicitly across components. This may indeed cause problems in case a component is destroyed while others are trying to access the shared state. Additional support to programmers for dealing with these issues is provided in our FIGARO component model, described in Section 4.6.

Components offer their functionality through one or more *interfaces* each of which is defined in a programming language independent manner as a set of types and operation signatures. In addition, components can access the functionality of other components using one or more *receptacles*. To this end, a component must have each of its receptacles connected (using `connect()`) to a corresponding interface on some external component before it can execute. This connection between a receptacle and an interface is explicitly represented in the model through a so-called *connector*, which is itself a component and therefore can be deleted using `destroy()`.

The model also incorporates the notion of *connector factories*. These are components that create connectors that embody a specific piece of behaviour to be invoked every time a call is made over a given receptacle/interface connection. In this way, connectors may encapsulate arbitrary functionality and can thus be used to perform such functions as monitoring or intercepting communications between their associated receptacle and interface. For instance, a connector factory may create customized connectors to log the communication between two components on permanent storage for debugging purposes. Connector factories are passed as arguments to `connect()`; or, if a null argument is passed, a "default" connector factory is used which binds the receptacle directly to the interface. Note that connector factories are *not* normally used to abstract over network communications; rather, they are intended for "local" use only. Network communication is assumed to be encapsulated within middleware components (see Section 4.4) and is thus transparent to the component model.

[1] The `pattern` argument to `load()` is simply a flexible way of specifying a component type.

All of the above entities (i.e., components, component types, interfaces, receptacles, and connectors) may be annotated with *attributes*. These are key/value pairs that can be used to express arbitrary meta-data. Attributes are managed using `setAttribute()` and `getAttributes()`. Finally, all of the entities reside inside a *capsule* which serves as a run-time component container, providing name space functionality. A capsule can be implemented as an operating system address space although this is not mandatory. All the entities currently inside a capsule can be enumerated using the `getEntities()` operation.

4.3.2. Kernel Implementations

Here we describe concisely how the core abstractions defined by the RUNES component model are realised in three different implementations: *(i)* a Java-virtual-machine-based implementation; *(ii)* a C/Unix-based implementation; and *(iii)* an implementation based on tiny embedded devices running the Contiki [56] operating system.

Component Types and Components. In the Java implementation, component types are straightforwardly represented as classes that inherit from a specific abstract class. This approach allows us to "factor out" the code needed to support component instantiation and destruction. Therefore, components can be realised simply as objects instantiated from a class representing a component type, and the `load()` operation is simply implemented using the default Java class loader. In the C/Unix implementation component types are represented as Unix "shared objects" compiled from source files conforming to a specified structure. The `load()` operation is implemented in terms of the native load/link facilities provided by the operating system, e.g., using `dlopen()`, and instantiation is accomplished by allocating a struct containing per-component state. Each interface operation defined in a component type (realised as a C function) takes as its first argument a pointer to this per-component struct so that the particular component instance being invoked can be determined. In the Contiki implementation, component types are similarly implemented as C source files which map to Contiki "services"; and the Contiki dynamic loading facility is used. Because Contiki supports only a single instance of a given type of "service", the `instantiate()` operation currently only returns a newly instantiated component *once* for each component type. We are currently looking into removing this limitation.

Interfaces, Receptacles and Connectors. In the Java environment, interfaces are trivially implemented as Java interfaces, whereas receptacles are implemented as Java objects. Component types contain initialisation code to create the appropriate receptacles at component instantiation time. In the C/Unix environment both interfaces and receptacles are represented as C structs. Both contain an array of function pointers. In the case of an interface, these pointers point at the target operations (C functions). In the case of a receptacle, they are assigned during `connect()` either directly to the function pointer values in the associated interface, or indirectly via functions within the specified connector that contains some intermediate functionality. In the Contiki environment, a similar approach is adopted. In the Java and C/Unix environments we provide the 'full' semantics of connectors, i.e., we provide the ability to employ user-defined connector factories to customize their behaviors as described above. Currently we do not provide this functionality in the Contiki environment, but there is no a priori reason why the Contiki implementation could not be extended in this way.

4.4. The RUNES Middleware in Action

Using the three kernel implementations described above, we have developed a set of middleware and application components that collectively address the road tunnel scenario outlined earlier. The overall design of the resulting application is depicted in Figure 4.4.

The application is structured as follows. TMote Sky [73] nodes running the Contiki-based kernel support a *Data Acquisition* component and a *Data Dissemination* component that together monitor and disseminate environmental conditions in the tunnel. These report, via gateways running the C/Unix kernel and supporting a *Packet Forwarding* component, to a central control station that includes a *Data Logging* component running on a PC that uses the Java kernel. The communication is handled by an underlying *μAODV* component which provides multi-hop routing.

When an emergency occurs, the Data Acquisition components respond initially by sending readings more frequently. In addition, the μAODV component has the ability to automatically recover from damage to either sensors or communication paths. Eventually, firefighters arrive equipped with mobile, wireless devices, forming a mobile ad-hoc network. The firefighters' devices instruct the sensors to send their readings directly to the

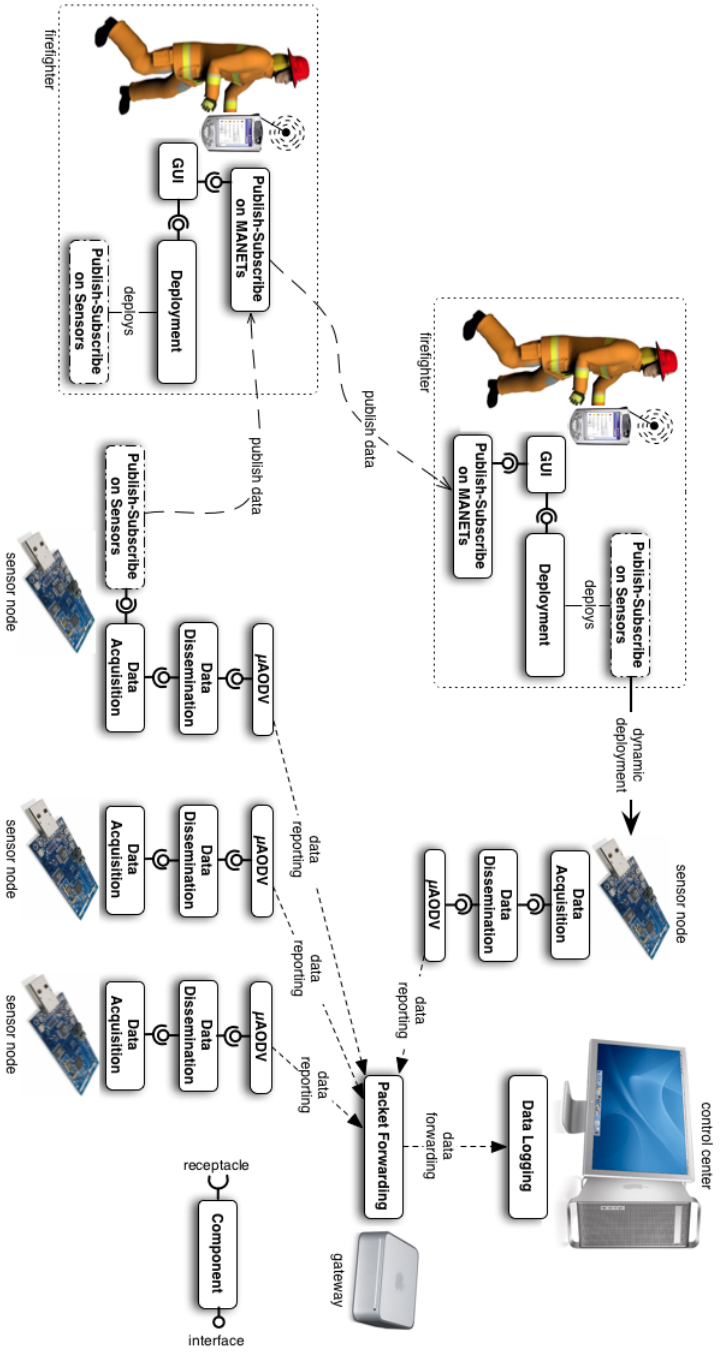


Figure 4.4.: Fire in a road tunnel: application design.

	Device	**Kernel Platform**	**Middleware Components**
Step 1 Quiescent conditions	Sensor	Contiki	Data Acquisition Data Dissemination μAODV
	Gateway	C/Unix	Packet Forwarding
	Control center	C/Unix	Data Logging
Step 2 Fire detected	Firefighter	Java	Publish-Subscribe GUI Component
Step 3 Firefighters reconfigure sensors	Sensor	Contiki	Publish-Subscribe
	Firefighter Firefighter	Java	Publish-Subscribe Deployment

Figure 4.5.: Configuration of the application as the scenario unfolds.

firefighter as well as to the Data Logger. Also, the firefighters coordinate their actions using a *Publish-Subscribe* [118] component that helps them sharing locally sensed information. The firefighters additionally run a *Deployment* component that has the capability to dynamically deploy a Contiki version of the Publish-Subscribe component directly onto the sensor devices so that the latter can start broadcasting directly to any nearby firefighters who subscribe to relevant events, e.g., temperature readings above a safety threshold. The Deployment component first checks if the sensors within range already run the Publish-Subscribe component. If not, the owning firefighter is prompted about the possibility of uploading the component on those sensor devices still lacking it. If there is no space on a sensor for the Publish-Subscribe component, the original Data Dissemination component is removed. All of this behaviour is under the control of the firefighters who interact with their devices using a GUI component. Figure 4.5 summarises the configuration of the devices involved as the situation unfolds.

The Publish-Subscribe component is the most complex of the components described above and deserves further explanation. The component employs a layered architecture, in which two sub-components take care respectively of the two concerns relevant to dealing with host mobility, i.e., overlay maintenance and route reconfiguration on top of the overlay. The separation of these two concerns is especially beneficial in allowing independent customisation of these two aspects. In more detail, the first sub-component takes care of creating and maintaining a tree-shaped overlay based on the algorithm described in [135]. The second sub-component is then in charge, using the mechanism described in [136], of setting up

message routes on top of the overlay, and reconfiguring these routes in case of topology change.

The application provides a clear illustration of the benefits of the RUNES approach. First, a unified component-based software development approach is adopted regardless of the type of device involved. Second, the component approach encourages the development of independent pieces of functionality that can be composed in various useful ways depending on context. Third, the dynamic loading capability relaxes the need to anticipate all the functionality that will be needed on a node. This is especially beneficial for WSN nodes on which it may not be possible to fit all the components required beforehand. Fourth, the dynamic (re)connection capability makes it possible for newly deployed components to interact in complex ways with the existing components in a type-safe manner. For example, initially, the Data Acquisition component is bound to the Data Dissemination component; however, when the Publish-Subscribe component is uploaded, it is dynamically rebound to the latter.

4.5. Evaluating the RUNES Middleware

This section assesses the effectiveness of the RUNES middleware in coping with heterogeneity, resource scarcity and dynamic scenarios, and also assesses its specific competence for our reference scenario. Specifically, we present an evaluation of the different implementations on the three hardware/software platforms we considered, followed by performance figures related to the specific components developed for our prototype application.

4.5.1. Middleware Kernel Evaluation

First we present an evaluation of the middleware kernel implementations. As evaluation testbed, for the Java implementation we used Sun JVM v5.01a running on a Pentium 4 3.2Ghz with 1 GB RAM. For the C/Unix implementation we used a Pentium 4 2.4Ghz running Linux 2.6, and for the C/Contiki implementation we used TMote Sky motes [73].

Metrics. To demonstrate the ability of the middleware to support heterogeneity and resource scarcity, we measure the *kernel memory footprint*, i.e., the data and code memory footprint consumed by the run-time support for the RUNES component model. In addition, to evaluate the memory overhead required to represent the component, interface and receptacle

Performance Measure (Memory Footprint)	Java	C/Unix	C/Contiki
Kernel Code	14.65 KB	16 KB	780 bytes
Kernel Data	840 bytes	4 KB	52 bytes
Null Component Data	544 bytes	24 bytes	9 bytes
Per-Interface Data	200 bytes	40 bytes	2 bytes
Per-Receptacle Data	264 bytes	22 bytes	2 bytes

Figure 4.6.: RUNES Middleware memory overhead.

concepts from the programming model, we measure the *memory footprint of a null component* and the *per-interface, per-receptacle memory footprint.* A null component is a component with no interfaces/receptacles and null initialisation/destruction routines.

To investigate the dynamic aspects of the middleware, we consider the overhead of *null operation calls through a default connector*. A null operation is one with no in/out parameters performed across a connector without intervention in the control flow, and introduces some overhead w.r.t. "native" operation calls (e.g., a method invocation in Java). This measure represents the run-time overhead of introducing connectors in the programming framework. We also consider the operations needed to dynamically modify the software running on a node. To that end, the kernel must *load* a new component, *instantiate* it, and *connect* the new instance to an existing component. In the case of the Java and C/Unix kernels, we measured each of these aspects separately, using a null component. We note that the fine-grained time aspects cannot be measured on the motes due to timer service limitations.

Results. The RUNES approach addresses heterogeneity effectively. This is shown by implementing the same software component model on a variety of devices, ranging from powerful desktop PCs to resource-constrained devices. Different programming languages and concurrency models have been used on different platforms. Our support of heterogeneity is further demonstrated by the relative sizes of the different middleware kernel implementations, shown in Figure 4.6. This highlights that our implementations scale down to severely constrained devices.

Even on the most resource-constrained of the platforms considered, the TMote Sky motes, the kernel footprint of 780 bytes is less than 1% of the total available flash memory of the motes (48KB internal and 1MB external flash memory). The overhead due to the introduction of components, in-

Performance Measure	Java	C/Unix	C/Contiki
Overhead of null Calls (DefaultConnector)	158.93%	99.84%	137.5%
Component Loading Time	0.0006 ms	0.2116 ms	2.4973 s
Component Instantiation Time	0.0047 ms	0.7674 ms	N/A

Figure 4.7.: RUNES Middleware run-time overhead

terfaces and receptacles in the programming model of Contiki is negligible with respect to the amount of RAM (10KB) available on the TMote Sky motes onto which they would be loaded. This minimal overhead obtained is due to the simplicity of the RUNES component model. This enables software reconfiguration through simple, yet powerful, abstractions, that are easily implementable.

Figure 4.7 reports on the dynamic aspects of our implementations[2]. The overhead introduced by null operation calls through default connectors may appear to be non-negligible with respect to their "native" equivalents. However, further investigation revealed that invoking a void Java method through a default connector takes only 23.5 μs, on average. Therefore, the time needed to execute an actual fragment of code inside the method body would consume the majority of the overall computation time, making the overhead of the connector negligible. Similar considerations apply for the C/Unix and the Contiki implementations.

The remaining data in the same table refers to the operations needed to change the software running on a node. Among these operations, component loading and instantiation are the most expensive, because of the work involved in transferring the component and creating data structures within the middleware kernel. Given the values obtained, and also considering that such operations should be triggered only when needed, we argue our kernel implementations are able to adapt sufficiently quickly to a changing environment.

4.5.2. Scenario-Based Evaluation

We now provide a basic evaluation of aspects of the road tunnel scenario reported in Section 4.4. These measurements were made on an experimental set-up consisting of a TMote Sky node representing a sensor device in the tunnel, and two laptops representing firefighter devices. More precisely,

[2]We executed 10,000 iterations and averaged the results.

Performance Measure	Data Acquisition	Data Dissemination	Publish-Subscribe
Source Lines of Code	287 lines	181 lines	197 lines
Memory Footprint	1462 bytes	738 bytes	772 bytes

Figure 4.8.: Application component size.

the firefighter devices each comprise a laptop plus a TMote Sky node attached to the laptop via a USB cable; the TMote Sky node simply forwards packets from the firefighter laptop to the tunnel sensor and vice versa. The sensor device runs the Contiki implementation of the middleware kernel, whereas the firefighter devices run the Java version.

First, we evaluated the sizes of some of the components running on the sensor device. The results in Figure 4.8 show that these are negligible compared to the available resources of the TMote Sky motes. By adding together the footprints of the components and the middleware kernel, we see that the size of the sensor node installation is 3750 bytes. This is still less than 1% of the total memory available on a TMote Sky mote.

We also measured the lines-of-code and memory overhead for the Java Publish-Subscribe component on the firefighter devices. This amounts to 1327 lines of non-commented code, and occupies 8.23 KB of memory. Again, a very acceptable overhead.

Finally, we carried out some basic performance measures to confirm that the network overheads are sufficiently small for run-time reconfiguration to be feasible. To this end, we measured 2.07 seconds to deploy a null component onto the sensor device, and 4.25 ms for a Publish-Subscribe message sent between firefighter devices. These figures indicate that the network overheads are indeed acceptable.

Despite the generality and efficiency of the RUNES middleware, programmers are still required to manage the reconfiguration process by hand, using the API in Figure 4.2. Moreover, they lack support to handle component dependencies and versions, making it difficult to integrate functionality from different parties. To address these issues, we devised the FiGaRo component model, described in the following.

4.6. The FiGaRo Programming Model

FiGaRo is a flexible component-model explicitly conceived for WSN devices. It is currently built atop the Contiki [56] operating system, and

```
DECLARE_INTERFACE(data_collection_if, {
  void (* broadcast_interest)(void* data,  u8_t len);
  void (* report)(uip_ipaddr_t dest, void* data,  u8_t len); })
```

Figure 4.9.: FIGARO: an example of component interface.

```
DECLARE_COMPONENT(tree_routing, data_collection_if, 2)
DECLARE_DEPENDENCY(radio_receptacle, radio_if, 3, MANDATORY | STATIC)
void broadcast_interest(void* data,  u8_t len) {
  CALL(radio_receptacle, send(&broadcast_addr, &msg, 64));
  // ...
}
void report(uip_ipaddr_t dest, void* data,  u8_t len) {
  // ...
}
ON_RUNNING({ // ON_SUSPEND, ON_DESTROY are also available
  // ...
})
```

Figure 4.10.: FIGARO: a component implementing the interface of Figure 4.9.

therefore relies on the C programming language. It leverages off dedicated C macros to move most of the processing to the compilation stages while not requiring any pre-processing step. Its core abstractions are described next.

Components, Interfaces, and Dependencies. Similarly to RUNES, in FIGARO, a *component* represents a single unit of functionality and deployment. The services provided by a component are described by its *interface*. For instance, Figure 4.9 shows the declaration of an interface for data collection. This specifies the signature of two operations to broadcast interests and to report the data, respectively. Components must provide the code for all the operations in the interface declaration, as in the case of Figure 4.10. The `DECLARE_COMPONENT` macro is used to specify the name of the component (`tree_routing`), the interface it implements (`data_collection_if`), and the component version (`2`).

To accomplish its goal, a component interacts with others on the same node. Interaction occurs through function calls across components using `CALL`, as shown in the first operation of the component in Figure 4.10. However, it is not for granted that a component provides an (interface

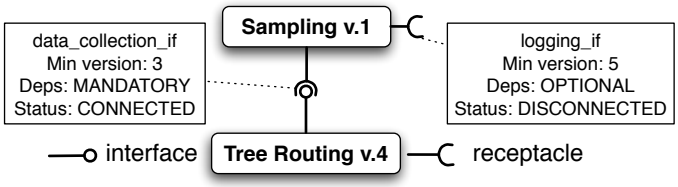


Figure 4.11.: An example of component configuration.

containing the) operation required by another, while the caller component may not be able to continue its execution without a callee component implementing the required interface. Therefore, the presence of a `CALL` statement determines a *dependency* between caller and callee.

In FIGARO, dependencies are explicitly declared by the programmer using the `DECLARE_DEPENDENCY` macro. The first parameter of this macro is a *receptacle*, the dual of an interface. An interface specifies a set of operations provided by a component *to* others, while a receptacle specifies the set of interfaces a component requires *from* others. In the case of Figure 4.10 the dependency being declared specifies the name of the receptacle (`radio_receptacle`), the interface required (`radio_if`), and the minimum component version allowed for a component (`3`). Moreover, the programmer can also specify a bit-masked constant describing the *nature* of the dependency. In the example, `MANDATORY` specifies that the component cannot run without relying on the needed interface. Otherwise, the dependency is considered optional, and the component is expected to work correctly also in absence of the specified interface. Instead, `STATIC` indicates that once a callee component is bound to the caller through the receptacle, the callee component cannot be changed. Otherwise, a reconfiguration can take place substituting the component with another providing the same interface.

Figure 4.11 shows an example of component configuration. The `Sampling` component is responsible for querying the sensor, and calling the `report` function in `TreeRouting`, which transmits the data to a sink. Note how `TreeRouting` satisfies only the `MANDATORY` dependency of `Sampling`, while the `OPTIONAL` one is currently not satisfied. This information is reflected in the receptacle descriptor inside the run-time support, as described in Section 4.7.

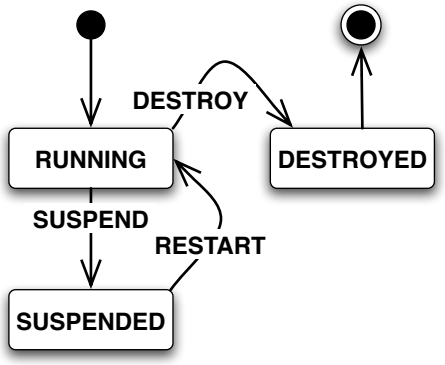


Figure 4.12.: The life cycle of a FiGaRo component.

Component Life Cycle. The life cycle of a component is illustrated in Figure 4.12. A component becomes `RUNNING` when all its dependencies on other components are *satisfied*, i.e., component implementing the required interfaces are available on the node. Note that dependencies are inherently recursive, i.e., a component may depend on some others, which in turn may depend on others, and so on. Therefore, the instantiation of a component may trigger the instantiation of an entire component *closure*, based on the declared dependencies. In practice, however, WSN applications are made of a small number of components with short dependency chains. The instantiation of a set of components bound by dependencies occurs atomically, i.e., control returns to the application only when the instantiation of *all* components is complete. When a component providing services to others undergoes a reconfiguration, the components exploiting those services move to the `SUSPENDED` state, and revert to the `RUNNING` state when the reconfiguration completes. Instead, the `DESTROYED` state is reached when the component has been replaced by another with the same interface.

Programmers can intervene at each step of the life cycle by specifying code fragments to be executed when entering a given state, as shown in Figure 4.10. When starting a new component, for instance, the body of the `ON_RUNNING` macro is executed. Similar operations exist for each state. The ability to intercept run-time activities is particularly important in the case of `SUSPEND`, to give programmers the ability to release resources held by the suspended components, and avoid deadlocks and run-time faults.

Component Reconfiguration. Differently from RUNES, in FiGaRo

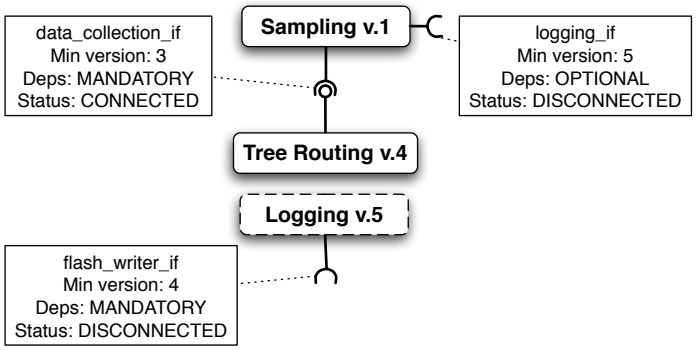
programmers do not need to manage the reconfiguration manually using a dedicated API. Instead, the underlying run-time automatically and transparently manages the reconfiguration process, based on dependencies and component versions. When components are instantiated at start-up, the run-time keeps track of their version, the interface they implement, and their dependencies. Upon receipt of a new component C, reconfiguration unfolds as follows, provided C's `MANDATORY` dependencies can be satisfied:

- C is instantiated if there is no running component with the same interface, or
- C replaces another component C_{old} implementing the same interface as C if:
  - C's version is greater than C_{old}'s,
  - no component currently relying on C_{old} has a `STATIC` dependency on it.

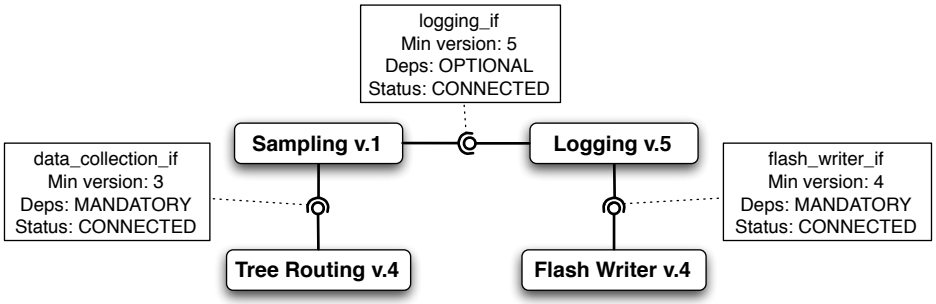
If a component cannot be immediately instantiated because of one or more unsatisfied `MANDATORY` dependencies, it is buffered in the hope that the necessary components are received later on. If this does not happen, the component is discarded after a timeout.

As an example, Figure 4.13 shows a possible evolution of the configuration shown in Figure 4.11. When a `Logging` component is received, the node-level run-time determines that it can be used to satisfy the optional dependency of `Sampling`. However, `Logging` has a `MANDATORY` dependency of its own, which cannot be satisfied right now. Therefore, `Logging` is temporarily buffered and remains disconnected from the other components, yielding the configuration in Figure 4.13(a). In Figure 4.13(b), a `FlashWriter` component satisfying the dependency of `Logging` is received. The run-time determines, by recursively travelling the component graph, that all dependencies are now satisfied, and instantiates the new components in the correct order (i.e., `FlashWriter` before `Logging`), yielding the configuration shown in the figure.

The automatic reconfiguration mechanism relieves the programmer from checking the conditions for the reconfiguration to take place, changing the component interconnections, and managing the coordination among the components involved. Although similar approaches (e.g., [137]) already proved their effectiveness in other contexts, to the best of our knowledge we are the first to enable this functionality in WSNs.



(a) `Logging` is received.



(b) `FlashWriter` is received.

Figure 4.13.: A sample evolution of the component configuration in Figure 4.11.

4.7. FiGaRo Node-Level Run-Time Support

FiGaRo provides the constructs described in Section 4.6, concerned with node-level reconfiguration, by making extensive use of C macros, therefore moving at compilation time most of the added complexity. However, dynamic reconfiguration requires specialized run-time support, provided by library functions we developed, linked against the (unmodified) Contiki kernel.

Similarly to the Contiki version of the RUNES middleware, the FiGaRo run-time maps FiGaRo components to Contiki services [56], and leverages off Contiki's dynamic linking facility [138] to install new code. Conse-

quently, the implementation of the `CALL` macro uses Contiki look-up functions to find a pointer to the callee component, and perform the operation requested. Interfaces and receptacles are represented by descriptors (standard C structs) containing an array of function pointers. In the case of interfaces, these always point to the corresponding functions in the component currently implementing the interface. Instead, the pointers inside receptacles are assigned the function pointer values of the associated interface, when connected, or `NULL` otherwise. In addition, receptacle descriptors contain further fields to keep track of the nature of dependency, as well as the minimum version required by any component connected to it, as shown in Figure 4.13.

Based on the information gathered by our macros during the compilation phase, our run-time maintains on every node an internal representation of the exported attributes and current software configuration. This is represented as a graph where vertexes are components, and edges are labeled to reflect the nature of the dependency at hand, similarly to Figure 4.13. When a new component arrives, simple graph traversal algorithms are used to check the conditions for the installation of a new piece of functionality. If the new component can indeed be installed, the run-time fires the relevant state transitions on all involved components, installs the new component by reconfiguring the involved receptacles, and updates the graph accordingly.

4.8. Evaluating the FiGaRo Component Model

Our objective here is to quantify the overhead imposed by FiGaRo w.r.t. plain Contiki. The implementation of our component model on single nodes may affect several aspects. For instance, further data structures are needed to keep track of the current component configurations, and additional processing is required before and after installing a new component. Motivated by these considerations, we consider the following performance figures:

- The *memory occupation* caused by our component model, w.r.t. both program and data memory. We evaluated the former by looking at the size of binary images after compilation. As for the latter, we manually inspected the code managing components and their interconnections, looking for any data structure we defined.

- The additional *processing time* caused by the presence of components. This is affected both by the installation of a new component compared

Performance Measure	Memory	Footprint
Dependency Checks	Program	1.1 KB
Helper Functions	Program	802 bytes
Helper Data Structures	Data	230 bytes
Per-Component Data	Data	15 bytes
Per-Interface Data	Data	8 bytes
Per-Receptacle Data	Data	10 bytes

Figure 4.14.: Memory overhead.

to the native Contiki dynamic linker, and by function calls across components using `CALL` instead of a direct C call. As for the latter, we placed the call in a loop and repeated the operation a million times, since the single call is too quick to be measured precisely.

- The *energy consumption* during reconfiguration, which may increase as a result of the additional processing required to manage components and dependencies.

We measured processing time and energy consumption using real nodes as opposed to simulation environments, as similar fine-grained aspects are only partially modeled in existing simulators. Practically, we measured the processing overhead using a JTAG programmer attached to the node to measure the time elapsed between the execution of different instructions. Energy consumption was instead evaluated using an Agilent 54832B oscilloscope and a multimeter hooked to a node, which in our case was a TMote Sky [73]. We repeated the experiments concerning these metrics 5 times using 3 different nodes, and averaged the results. New components have been injected via a USB cable attached to the node, to avoid any bias due to the radio.

To gather the above metrics, we employed a `Blinker` component offering a single interface with two operations to start/stop the blinking of a led. We varied the number of receptacles within the component itself to evaluate our performance w.r.t. a varying number of dependencies. The processing within `Blinker` is the same as in [138], and is quite simple being described by only 17 lines of C code. This choice was intentional, as simpler components make the overhead more evident w.r.t. the above metrics.

Results. Figure 4.14 shows the memory overhead, which turns out to be quite reasonable, w.r.t. both program and data memory. As for the former, the binary code deployed in addition to the operating system accounts for

Function Type	Time Overhead %
Empty	157.5%
50 integer additions	20.1%
3 x 3 matrix inversion	5.4%
5 x 5 matrix inversion	0.98%
Fourier Transform (100 input values)	0.78%
Fourier Transform (1000 input values)	0.03%

Figure 4.15.: FiGaRo calls across components vs. native C function calls.

Dependencies	Time (s)		Energy (mJ)	
	Absolute	Overhead	Absolute	Overhead
1	0.518 sec	+0.019	3.45	+0.07
2	0.520 sec	+0.021	3.45	+0.07
3	0.525 sec	+0.026	3.47	+0.09
4	0.528 sec	+0.029	3.49	+0.11
5	0.532 sec	+0.033	3.5	+0.12

Figure 4.16.: Time and energy to install the `Blinker` component.

less than 2 Kbytes in total. This cost, along with the overhead due to helper data structures, is paid once and for all, regardless of the number of components and the number of their interfaces/receptacles. Conversely, the bottom section of Figure 4.14 reports the memory consumption incurred every time a component, interface, or receptacle is loaded on a node. In this case as well, the overhead is fairly limited. Based on these results, we maintain that our approach can scale to a sizable number of components simultaneously running on the same node, presumably well beyond the current needs of common WSN applications. As for the amount of code to be deployed, we compared the size of the binary image of the plain-Contiki `Blinker` process used in [138] against ours, implemented as a FiGaRo component. The size increases from 1.01 Kbytes to 1.11 Kbytes, yielding an overhead of only 9.98%. We believe this value is good, given the little complexity of the processing at hand.

The overhead in performing calls across components against direct C function calls is reported in Figure 4.15. Interestingly, when the function called does not contain any real processing the overhead due to using `CALL`

is high. In this case, performing the look-up of the Contiki service implementing the requested component dominates the processing time. In contrast, some even simple processing within the function called makes this metric drop abruptly. For instance, in the case of a Fourier transform (e.g., employed to perform in-network processing in WSN applications such as [90]) the overhead becomes less than 1%. Therefore, although our programming model does introduce an overhead, the performance penalty is expected to be negligible in real applications.

By the same token, the time for installing a new component, and hence the energy consumed during this process, increases only marginally w.r.t. the standard Contiki dynamic linker, as shown in Figure 4.16 for a varying number of dependencies in the component being installed. Note how these values are independent of the size of the component being deployed, as they represent the overhead imposed by our run-time layer in addition to the Contiki dynamic linker, which we left unmodified. Also, they scale well with the number of dependencies, showing only a very small increase. To place Figure 4.16 in context, consider that the energy *overhead* in the case with 5 dependencies is equal to only about 5% of the *total* energy required to transmit a 32-byte message.

4.9. Related Work

We first compare our work on RUNES against alternative solutions providing support for developing pervasive or embedded applications. Next, we take a closer look at the current state of the art in software reconfiguration for WSNs, and compare our contributions against relevant examples.

4.9.1. System Support for Pervasive Embedded Applications

There is a substantial body of literature on reconfigurable middleware systems for pervasive applications. *Gravity* [139] is a component model built on top of the Open Services Gateway Initiative (OSGi) Framework [140]. *P2PComp* [141] is a lightweight service-oriented component model for mobile devices which is also built using OSGi; it provides location independent synchronous and asynchronous communication between components. The *Dynamically Programmable and Reconfigurable Software* (DPRS) architecture [142] is a component-based design for dynamically programmable and reconfigurable systems. *PCOM* [137] is a distributed component model

for pervasive computing. It allows for designing applications as a collection of potentially distributed components, which make their dependencies explicit. If those dependencies are invalidated, PCOM can attempt to automatically adapt by detecting alternatives according to various strategies. *FarGo-DA* [143] is a distributed component model that uses logical mobility to allow disconnected operation. The *Software Dock* [144] is an agent-based software deployment network that allows negotiation between software producers and consumers. *THINK* [145] presents an approach for building component-based operating system kernels. And finally, *one.world* [146] is a system for pervasive applications that supports dynamic service composition, migration of applications and discovery of context.

In summary, there are two main differences between the approaches outlined above and our work on RUNES. The first difference relates to *generality*: RUNES is a generic software fabric that is designed from the ground up to be implementable on a wide range of devices, and to allow the implementation of a large number of very different primitives. This is an essential requirement of pervasive applications such as disaster management. The second difference relates to our two-layer architecture in which systems are built by selecting (and dynamically reconfiguring) appropriate middleware and application components on top of the middleware kernel. This capability, lacking in other works, results in significantly greater flexibility than current systems offer.

4.9.2. Software Reconfiguration in WSNs

Different component-based programming models are specifically targeted at embedded systems. These are not, however, necessarily networked. Examples include *Pebble* [147], *PECOS* [148], *PBO* [149], *SaveCCM* [150] and *Koala* [151]. Most of these are build-time only technologies—components are not visible at run-time and therefore these systems do not support dynamic reconfiguration.

In WSNs, several solutions enable the installation of new code on individual nodes. At the operating system level, besides Contiki also the SOS operating system [57] provides dynamic linking, while FlexCup [41] enables this functionality in TinyOS [42], where this was initially not possible. These solutions concentrate on efficient dynamic linking, and are therefore complementary to our approach. In principle, both the Contiki version of RUNES and FIGARO can be re-applied in SOS and FlexCup with minimal modifications, as they are mostly based on standard C macros. We chose

Contiki because, unlike FlexCup, it preserves the application state without requiring a reboot after code loading and, in comparison to SOS, its service functionality eases the implementation of our component models.

Alternative approaches to software reconfiguration use interpreted languages and virtual machines (e.g., [50,52,53]), with some also allowing for extensible instruction sets, e.g, [51]. Nonetheless, the trade-offs between interpreting code and executing native binaries, as discussed in [50], suggest the use of the latter for long-running systems where reconfiguration is a not so frequent event, as in the scenarios we target.

Most importantly, none of the above approaches provides support to the programmer for managing the interactions among the different functionality on a node during reconfiguration. Indeed, even though component models for WSN programming have already been proposed (e.g., [28,152]), they do not include any dedicated construct for managing mutable component configurations. Conversely, in FiGaRo we made component dependencies and versions first-class citizens in the programming model, and designed the reconfiguration mechanism by balancing automation and customizability.

5. The TeenyLIME Middleware

In the sense-and-react scenarios we outlined in Chapter 1, actuators are physically interspersed with the sensors that trigger them. This solution maximizes localized interactions, improving resource utilization and reducing latency w.r.t. solutions with a centralized sink. Nevertheless, application development becomes more complex: the control logic must be embedded in the network, and coordination among multiple tasks is needed. The latter requirement usually demands for both reactive and proactive interactions. Unfortunately, mainstream WSN programming frameworks seldom provide similar features.

*In this chapter we present the design, implementation, and evaluation of Teeny*LIME*, a WSN middleware designed to address the above challenges. Teeny*LIME *provides programmers with the high-level abstraction of a tuple space, enabling data sharing among different software components on the same node, as well as neighboring devices. Teeny*LIME *yields simpler, cleaner, and more reusable implementations, at the cost of only a very limited decrease in performance. We support these claims through a source-level, quantitative comparison between implementations based on Teeny*LIME *and on mainstream approaches, and by analyzing measures of processing overhead and power consumption obtained through cycle-accurate emulation. The results presented here have been published in [7, 8].*

5.1. Introduction

The sense-and-react pattern we described in the introductory part of this thesis has a relevant impact on application development. Appropriate programming constructs are required to deal with the increased complexity of specifying how multiple tasks *coordinate* to accomplish the desired global functionality. Dedicated abstractions must be provided to describe the *stateful* interactions commonly present in control mechanisms. Moreover,

the ability to locally *react* based on external stimuli is as important as—if not more important than—the ability to gather data. These aspects are discussed in more detail in Section 5.2, where we describe a paradigmatic sense-and-react application. In addition, in the same section we also illustrate how many characteristics germane to sense-and-react are common to sense-only applications and generic mechanisms, e.g., such as those in the system services layer we described in Section 2.2.

To meet the requirements above, we developed TeenyLime: a WSN middleware whose foundation is the notion of *tuple space* [29], a repository of elementary sequences of typed fields called tuples. This is revisited in TeenyLime by considering WSN requirements (e.g., resource consumption and reliability) in the programming model. TeenyLime adopts a minimalist approach: a limited number of powerful operations, with a simple and yet efficient implementation, allow for the development of both applications and system services. An overview of TeenyLime's core concepts and application programming interface (API) is provided in Section 5.3. Instead, Section 5.4 illustrates concretely the power of TeenyLime's abstractions by showing them in action in the design of the aforementioned sense-and-react application, as well as by briefly describing how TeenyLime can be applied to the development of sense-only applications and system services.

At the same time, the development of WSN middleware must reconcile the need for expressive power and ease of programming with the reality of resource-scarce devices. In this respect, Section 5.5 provides a concise account of the TeenyLime internal architecture and implementation.

Section 5.6 evaluates quantitatively TeenyLime along two dimensions. First, we assess the effectiveness of its *programming model* in different contexts. We examine the implementation of the reference application, whose design we sketched in Section 5.4, and report about uses of TeenyLime in sense-only applications and for implementing system services. We derive code metrics for the TeenyLime implementations and their counterparts, implemented using plain nesC or the higher-level support provided by Hood [21]. Results indicate that the expressive power of TeenyLime yields cleaner, simpler, and more compact code. Second, we analyze the TeenyLime *implementation*. We compare its overhead, in terms of processing time and energy consumption, against existing programming platforms. The results gathered using cycle-accurate emulation demonstrate that the beneficial higher level of abstraction provided by TeenyLime comes with only a very limited overhead.

5.2. Scenario and Motivation

As a paradigmatic example, here we consider *building monitoring and control*. Modern buildings typically focus on the following functionality:

- *heating, ventilation, and air conditioning* (HVAC [27]) systems provide fine-grained control of indoor air quality;
- *emergency control* systems provide guidance and first response, e.g., in case of fire [100].

These applications, as any other embedded control system, feature four main components, illustrated in Figure 5.1. The *user preferences* represent the high-level system goals, e.g., the desired temperature in the building and the need to limit fire spreading. *Sensing devices* gather data from the environment and monitor relevant variables, in our case, humidity and temperature sensors monitor air quality, while smoke and temperature detectors recognize the presence of a fire. *Actuator devices* perform actions affecting the environment under control: air conditioners adjust the air quality, while water sprinklers and emergency bells are used in case of fire. *Control laws* map the data sensed to the actions performed, to meet the user preferences. In our case, a (simplified) control loop may activate air conditioners when temperature deviates significantly from the user preferences, tuning this action based on the humidity in the same location. Further, it may immediately activate emergency bells when the temperature increases above a safety threshold, but operate water sprinklers only if smoke detectors actually report the presence of fire. Oscillating behaviors must be avoided in all situations.

Application development in these scenarios is complicated not only by the peculiarities of devices, but also by the complexity of their interactions. The many requirements can be grouped into high-level challenges:

- *Localized computations* [153] must be privileged, to keep processing close to where sensing or actuation occurs. In sense-and-react applications it is indeed unreasonable to funnel all the sensed data to a single base-station, as this may negatively affect latency and reliability without any significant advantage [2].
- The system performs *multiple tasks* in parallel. In our example, two control laws coexist: one for air conditioning, the other for handling

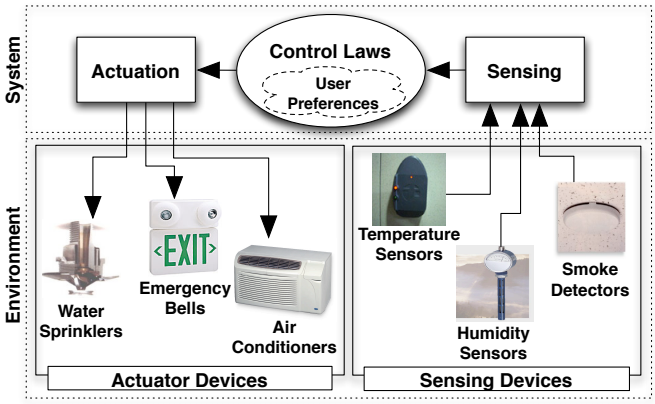


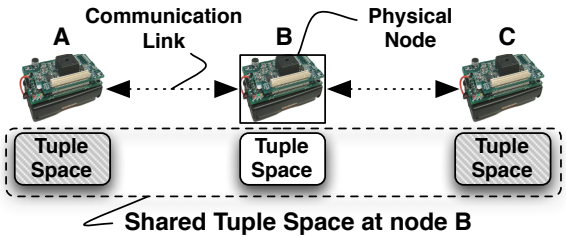
Figure 5.1.: High-level scheme of a building monitoring and control application.

emergencies. These need to *share data* (e.g., temperature readings) generated by a subset of the sensing devices.

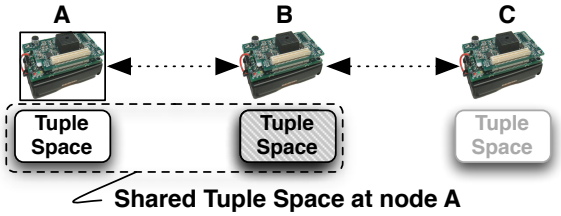
- Differently from sense-only scenarios, sense-and-react applications often require *stateful* coordination, e.g., using current shared conditions (state) to act collaboratively. This, in combination with the use of WSNs for safety critical applications, motivates an explicit account for *reliability* in the programming model.

- *Reactive interactions*, actions that automatically fire based on external conditions, assume a prominent role. In our case, a temperature reading deviating from user preferences triggers an action in both of the two application tasks. *Proactive interactions*, common in many sense-only scenarios, are still needed to gather information and fine tune the actuation about to occur. For instance, the sprinklers in the building ask for smoke readings before taking any action.

Note how subsets of these requirements must be accounted for also at lower levels, below the application. For instance, localization algorithms [34]—often one of the many tasks of object tracking applications [127]—must rely on localized interactions, as most of the approaches in the field base the position estimation on data reported by nearby hosts. Similarly, multi-hop routing mechanisms [33] require reactive interactions to adapt to mutable network conditions, and may also exploit reliable operations



(a) Shared tuple space at node B.



(b) Shared tuple space at node A.

Figure 5.2.: Tuple space sharing in TeenyLime.

to guarantee message delivery [154]. The TeenyLime programming model, described next, supports application development without losing the ability to express system-level mechanisms.

5.3. TeenyLime: Basic Concepts and API

TeenyLime is based on the *tuple space* abstraction, originally proposed in Linda [29] and here re-elaborated in the context of WSNs. A tuple space is a repository of data represented as *tuples*, sequences of typed fields such as ⟨"foo", 29⟩. Three core Linda operations allow processes to manipulate the tuple space by creating (**out**), reading (**rd**), and removing (**in**) tuples. Tuple selection with **rd** and **in** is based on matching patterns such as ⟨"foo", ?integer⟩ against the tuple space content. Patterns may use either *actual* or *formal* values, the latter serving as a kind of "wild card" matching any data of a particular type. The resulting programming model promotes anonymous and data-centric interactions. As such, it blends well with the requirements posed by WSNs, where data is of paramount importance and the identity of the individual nodes is not as significant.

In Linda, the tuple space is assumed globally accessible to all processes,

an undesirable choice in WSNs. Instead, in TeenyLime each node hosts a tuple space, shared among nodes within direct (one-hop) communication range. *Sharing* means that a node views its local tuple space as containing its own tuples, plus those in the tuple spaces hosted by its neighbors, as shown in Figure 5.2. Operations span the whole shared tuple space. For instance, a query issued by a node may return a matching tuple found in any tuple space in the one-hop neighborhood—including the local one. Therefore, TeenyLime programmers can specify interactions among nodes abstractly, by focusing on the application logic (e.g., reading temperature in the neighborhood) and leaving system configuration issues (e.g., tracking node identity and presence) to the middleware.

The choice to limit sharing to one-hop neighbors is motivated by the fact that interactions with these nodes are the most frequent in WSNs. Whitehouse et al. analyzed 16 publicly available applications to determine the node interactions, and

> "All neighborhoods discovered were one-hop neighborhoods [...]"
> ([21], p.9)

Interestingly, all neighborhoods were of limited size (at most ten nodes), and were used either directly at the application level to gain access to *nearby* information, or as a building block for lower-level system services, e.g., to implement multi-hop routing. These considerations also support our design choice, drawing the foundations for a highly-reusable programming model supported a by lightweight, scalable implementation. Furthermore, it should be noted that the applications considered in [21] were conventional sense-only ones. Sense-and-react applications exacerbate the need for localized interactions [153], and are therefore expected to benefit even more from our design. As a result, the TeenyLime programming model can be used in many contexts, ranging from sense-and-react to sense-only, and from application-level to system-level.

Figure 5.3 shows the TeenyLime API. While in principle the programming model is independent of the node platform, we present here the API in nesC [28], as our middleware is currently built on top of TinyOS [42]. The interface provides the operations to manipulate TeenyLime's shared tuple space. The first three operations correspond to the Linda operations discussed earlier, while **rdg** and **ing** are variants (as in [155]) that return all matching tuples, instead of a single match.

TeenyLime operations are asynchronous, allowing the application to

```
interface TupleSpace {

  // Standard tuple space operations
  command void out(TLOpId_t* opId, bool reliable,
                   TLTarget_t target, tuple *tuple);
  command void rd(TLOpId_t* opId, bool reliable,
                  TLTarget_t target, tuple *pattern);
  command void in(TLOpId_t* opId, bool reliable,
                  TLTarget_t target, tuple *pattern);

  // Group operations
  command void rdg(TLOpId_t* opId, bool reliable,
                   TLTarget_t target, tuple *pattern);
  command void ing(TLOpId_t* opId, bool reliable,
                   TLTarget_t target, tuple *pattern);

  // Managing reactions
  command void addReaction(TLOpId_t* opId, bool reliable,
                           TLTarget_t target, tuple *pattern);
  command void removeReaction(TLOpId_t operationID);

  // Returning tuples
  event void tupleReady(TLOpId_t operationId,
                        tuple *tuples, uint8_t number);

  // Request to reify a capability tuple
  event void reifyCapabilityTuple(tuple *capTuple, tuple *pattern);
}

interface NodeTuple {

  // Asks for a tuple containing node-level system information
  event tuple* reifyNodeTuple();
}
```

Figure 5.3.: TeenyLime API.

continue while the middleware completes the operation execution[1]. This approach blends well with the event-driven concurrency model of nesC. Therefore, all operations are *split-phase* [28]: the operation is issued, and later the `tupleReady` event is signaled when the operation completes. The `tupleReady` event contains an identifier (or a special constant `TL_OP_FAIL` in case of error), allowing the application to associate the event with its earlier request. Depending on the operation, one or more tuples, indicated by the `number` parameter, may also be contained in the event.

[1] In most Linda systems **rd** and **in** are blocking, i.e., they do not return until a tuple is matched.

The operations provided in the API deserve further discussion. However, instead of describing them in isolation, in the next section we discuss them "in action", i.e., hand-in-hand with the TeenyLIME-based design of our reference application.

5.4. Application Development with TeenyLIME

This section describes how the TeenyLIME programming model can be used to program sense-and-react applications as well as sense-only ones and system services.

5.4.1. Sense-and-react Applications

As discussed in Section 5.2, our reference application contains two sub-tasks, one managing the air conditioning system (HVAC) and the other for emergency situations such as fire. Each sub-task involves different types of nodes, e.g., humidity sensors in the HVAC sub-task, and smoke detectors to address fire emergencies. Temperature sensors are instead used in both sub-tasks. For all types of nodes, the application processing has been implemented in a single component sitting entirely on top of the `TupleSpace` interface, which masks completely TinyOS' generic communication layer. An additional component is employed to interact with the sensors/actuators attached to the node.

In the following, we explain the rest of our reference application's design and implementation. We illustrate how we exploit data sharing and related operations, and how interactions among nodes benefit from the WSN-specific API features. Throughout, the reference application is used as a motivation and source of examples for the discussion.

Sharing Application Data through Proactive and Reactive Interactions. In our design, *sensed data* and *actuating commands* take the form of tuples. These are shared across nodes (and components on the same node) to enable coordination of activities as well as data communication. Access to this data can occur *proactively*, e.g., using the **rd** and **in** operations. However, TeenyLIME supports also a notion of *reaction*, a code fragment whose execution is automatically triggered upon the appearance of a given tuple anywhere in the shared tuple space. The tuples of interest are identified through pattern matching, and the `tupleReady` event is used to signal a reaction firing. This provides an easy and yet very powerful

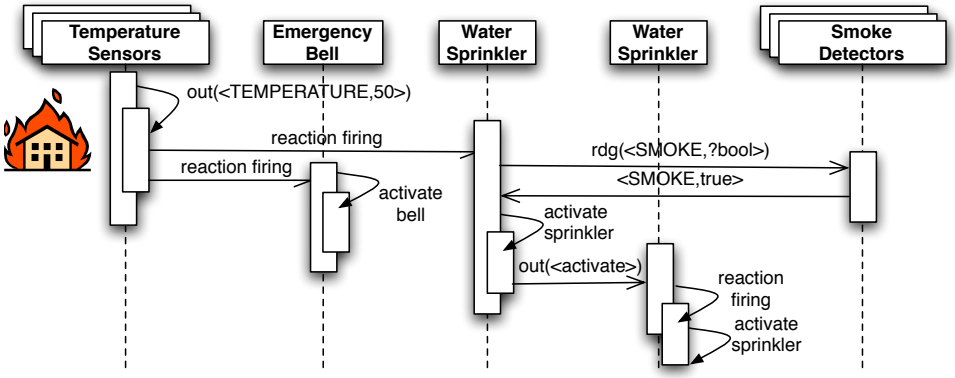


Figure 5.4.: Sequence of operations to cope with fire. Notified about increased temperature, a node controlling water sprinklers queries the smoke detectors to verify the presence of fire. If necessary, it sends a command activating nearby sprinklers.

way to monitor changes in the neighbors' data through the content of the shared tuple space.

Figure 5.4 uses the fire control sub-task to illustrate how proactive and reactive interactions are used together to trigger notifications, to perform distributed operations for gathering data from neighboring nodes, and to request actuation commands. Notably, similar patterns of interactions recur in both sub-tasks of our application.

Both emergency bells and water sprinklers have a reaction registered on their neighbors, watching for temperature tuples, as shown in the code in Figure 5.5. Temperature sensors periodically take a sample and pack it in a tuple, which is then stored in the local tuple space, as shown in Figure 5.6. Insertion is accomplished using **out** by setting the `target` parameter to `TL_LOCAL`, which entails outputting the tuple to the local tuple space. This operation, by virtue of one-hop sharing, automatically triggers all the aforementioned reactions[2], which process the tuple contained in the event `tupleReady`.

However, different types of actuator nodes behave differently when high

[2]We assume that actuators are interested in all temperature values. We show later how notifications can be triggered only when temperature is above (or below) a given threshold.

```
TLOpId_t reactId;
command result_t StdControl.start() {
  tuple tempTemplate = newTuple(2, actualField_uint16(TEMPERATURE),
                                   formalField(TYPE_UINT16_T));
  call TS.addReaction(&reactId, TRUE,
                      TL_NEIGHBORHOOD, &tempTemplate);
  return SUCCESS;
}
 event void TS.tupleReady(TLOpId_t operationId,
                          tuple *tuples, uint8_t number) {
   // Notification triggered ...
}
```

Figure 5.5.: TeenyLime code for an actuator node interested in temperature values.

```
command result_t StdControl.start() {
  return call SensingTimer.start (TIMER_REPEAT, SENSING_TIMER);
}
event result_t SensingTimer.fired() {
  return call TemperatureSensor.getData();
}
event result_t TemperatureSensor.dataReady(uint16_t reading){
  tuple tempValue = newTuple(2, actualField_uint16(TEMPERATURE),
                                actualField_uint16(reading));
  call TupleSpace.out(NULL, FALSE, TL_LOCAL, &tempValue);
  return SUCCESS;
}
```

Figure 5.6.: TeenyLime code for a temperature node.

temperatures are detected. The node hosting the emergency bell immediately activates its device. Instead, the water sprinkler node proceeds to verify the presence of fire, as shown in Figure 5.4. The latter behavior, specified as part of the reaction code, consists of proactively gathering the readings from nearby smoke detectors, using a **rdg** restricted (by setting `target` to `TL_NEIGHBORHOOD`) to the union of their tuple spaces. If fire is reported, the water sprinkler node requests activation of nearby sprinklers through a two-step process that relies on reactions as well. The node requesting actuation inserts a tuple representing the command on the nodes where the activation must occur, using **out** with `target` set to the sprinkler node address. The presence of this tuple triggers a locally-installed reaction delivering the command tuple to the application, which reads the tuple fields and operates the actuator device accordingly.

Reliable Operations. Since fire detection requires the maximum degree of reliability, its implementation takes advantage of *reliable operations* for guaranteeing correct communication of reactions and query results of the **rdg** operation on smoke detectors and of the **out** operations towards actuators. In this case, TeenyLime makes sure that all involved network-level operations are performed correctly. Several solutions can be employed to implement this feature, as discussed in Section 5.5.

The HVAC sub-task, instead, uses reliable operations only for actuation, but gathers data using non-reliable **rdg** operations. Furthermore, in the HVAC sub-task the system runs the risk of oscillating behavior if multiple nodes controlling air conditioners in the same location (e.g., same floor) independently run the control algorithm. To prevent this, we designed a mechanism to assign a master role to only one of the co-located controller nodes, achieving a sort of distributed mutual exclusion. The master node is identified as the one holding a special token tuple, periodically exchanged among co-located nodes to achieve a form of load-balancing. As a token loss implies no controller acting as the master, strong guarantees on token transfer are imperative. Therefore, the token exchange from the previous to the new master node is accomplished using a reliable **in** operation performed by the latter.

As shown in Figure 5.3, the selection between unreliable and reliable is done using a flag, available in most operations. The former offers a lightweight form of best-effort communication suitable for state-less applications (e.g., data collection), while the latter offer stronger guarantees to applications requiring stateful interactions.

Sharing System Data. Coordination of activities across heterogeneous nodes sometimes relies on system information, such as the node location or capabilities. In TeenyLime, this information is made available in the same way as application data, i.e., as tuples shared among neighboring nodes. In our scenario, these tuples contain a field describing the (logical) location (e.g., a room) where a node is deployed, and the sensor/actuator devices attached. Which data to provide is defined by the application programmer, by specifying the body of the handler for the `reifyNodeTuple` event, shown in Figure 5.3. This event is signaled periodically by the TeenyLime runtime, and the execution of the corresponding handler regenerates the tuple with new application-defined values. In our implementation, the local tuple space on every node contains tuples describing each of its neighbors. This is accomplished by appending the `Node` tuple to all outgoing messages;

therefore, when the message is overheard by neighbors, they extract the `Node` tuple and insert it locally. This way, it is easy to query the tuple space to obtain information on neighbors with specific capabilities.

Filtering Data. In many WSN applications, including ours, action must be taken only when a sensed value crosses a given threshold. Nodes controlling air conditioners must receive notifications when temperature falls outside a user-defined threshold. Similarly, the nodes controlling water sprinklers and emergency bells described previously only need to receive notifications when temperature rises above a safety threshold. These conditions require a predicate over tuple field values—something that cannot be achieved with the standard Linda matching semantics, which is based on either types or exact values. In TeenyLime, patterns are extended to support custom matching semantics on a per-field basis. For instance, the requirement concerning safety thresholds can be expressed concisely by using *range matching*, requiring the temperature field to be greater than a given parameter, as in:

```
tuple temperatureTempl = newTuple(2, actualField_uint16(TEMPERATURE),
                                     greaterField(TEMPERATURE_SAFETY));
```

where `TEMPERATURE_SAFETY` is a constant representing a specific safety threshold. The second field in the tuple above uses the default range matching, which the programmer can easily redefine.

Note how the issue is *not* simply one of expressive power, as it deeply affects communication. Without filtering, the programmer can only specify a *generic* pattern matching *any* temperature. All matching, outputted tuples would be transmitted (in our case, each time a new sample is available) and frequently discarded as out of range by the reaction code of the requester in Figure 5.5, wasting significant communication resources.

Dealing with Short-Lived Data. In some cases, sensor data remain useful only for a limited time after collection. For instance, an emergency bell is not interested in temperature values sensed an hour before. Instead, the same data may be of interest for a component that is periodically run to build a day-long analysis of temperature trends.

In TeenyLime, time is divided into *epochs* of constant length, and every data tuple is stamped with an application-accessible field containing the current epoch value. Three helper functions allow the application developers to deal with time:

```
setFreshness(pattern,freshness)
getFreshness(tuple)
```

```
command result_t StdControl.start(){
  tuple capTSmoke = newCapabilityTuple(2, actualField_uint16(SMOKE),
                                          formalField(TYPE_BOOL));
  call TupleSpace.out(NULL, FALSE, TL_LOCAL, &capTSmoke);
  return SUCCESS;
}
event void TupleSpace.reifyCapabilityTuple(tuple *ct, tuple *p){
  // Request a reading from the sensor
  return call SmokeDetector.getData();
}
event result_t SmokeDetector.dataReady(uint16_t reading){
  // Sensor reading ready
  tuple smokeValue = newTuple(2, actualField_uint16(SMOKE),
                                 actualField_bool(reading));
  call TS.out(NULL, FALSE, TL_LOCAL, &smokeValue);
  return SUCCESS;
}
```

Figure 5.7.: TeenyLIME code for a smoke detector node. Initialization routines and error handling are not shown, capitalized keywords represent constant values.

```
setExpireIn(tuple,expiration)
```

The first customizes a pattern, similarly to range matching above, to impose the additional constraint to match tuples no more than `freshness` epochs old. If a pattern does not specify freshness, it matches any tuple regardless of its age. The second function returns the number of epochs elapsed since the `tuple` was created. Finally, the third specifies how many epochs the `tuple` is allowed to stay in the tuple space. When the timeout associated to the tuple expires, the tuple is automatically removed.

Generating Data Efficiently. In our application, humidity sensors and smoke detectors need not be monitored continuously: their data is accessed proactively only when actuation is about to occur. However, when a sensed value is requested (e.g., by issuing a **rd**) fresh-enough data must be present in the tuple space. If these data are only seldom utilized, the energy required to keep tuples fresh is mostly wasted. An alternative is to require that the programmer encodes requests to perform sensing in a way similar to actuation commands, enabling the receiving node to perform sensing on-demand and return the result. However, this solution requires extra programming effort, is error-prone, adds processing overhead, and is therefore equally undesirable.

To deal with these (frequent) situations, TeenyLIME provides the ability

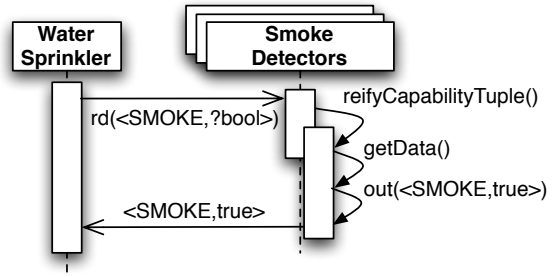


Figure 5.8.: Processing of capability tuples.

to output *capability tuples* indicating that a device has the capability to produce data of a given pattern. A code example for a smoke detector is shown in Figure 5.7. When a query is remotely issued with a pattern matching a capability tuple, the `reifyCapabilityTuple` event is signaled. This reports the pattern included in the query and the matching capability tuple. The application handles this event by taking a fresh reading and outputting the actual data to the tuple space. The sequence of operations is depicted in Figure 5.8. Note how, from the perspective of the data consumer, nothing changes. Instead, on the side of the data producer, capability tuples enable considerable energy savings as the readings are taken only on-demand, without the need to maintain constantly fresh data in the tuple space.

Interestingly, capability tuples can be generalized to allow *any action* to be taken by the data producer. For example, matching a pattern to a capability tuple may invoke any application function (e.g., computing the average of all recent temperature tuples), whose results are inserted in the tuple space and returned to the requester.

5.4.2. Sense-only Applications and System Services

The abstractions offered by TeenyLime support a wide range of settings. To demonstrate this, we implemented a well known tracking application and a multi-hop routing protocol on top of TeenyLime. The former has been widely recognized in the WSN literature as a benchmark to test the flexibility of programming abstractions [14, 21, 32]. The latter illustrates that the one-hop operations of TeenyLime are powerful enough to enable the implementation of multi-hop mechanisms.

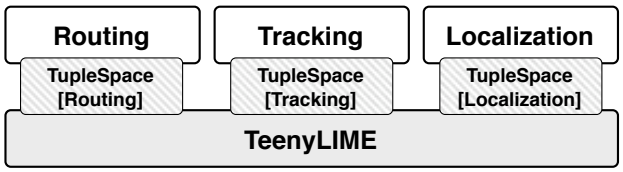


Figure 5.9.: Component configuration in object tracking.

Object Tracking in TeenyLIME. Object tracking is usually implemented as the composition of three core functionality [21]: *i)* a *localization* service in charge of determining the node position (e.g., using a GPS receiver), *ii)* a multi-hop *routing* protocol responsible for transporting data about the moving object to a central, fixed base-station, and *iii)* a *tracking* mechanism that determines the location of the moving object based on readings gathered from neighbors also sensing the target. As Figure 5.9 illustrates, using TeenyLIME we can achieve full decoupling between the three components even on a single node, i.e., no explicit interfaces are needed to connect them. Instead, the various components exchange data anonymously through the (local) tuple space. This greatly improves the re-usability of the single functionality.

In our implementation, the `Node` tuple contains an encoding of the device's physical position, and is therefore output by the localization component. This information is therefore available to the other components through the local tuple space. In particular, it is used by the tracking component to evaluate the position of the moving object w.r.t. the node's own location. As in [21], this component runs a simple leader election algorithm that identifies a specific node among the ones currently sensing the target to perform the aforementioned computation. To determine the potential candidate nodes, distributed reactions are used to notify a node's neighbors when the target is within sensing range.

When the leader node needs to send data to the collection point, it locally outputs a tuple containing the relevant information. This triggers a reaction installed by the routing component that immediately removes the information from the local tuple space using **in**, and propagates it towards the base-station. The decision on the next hop node is made by querying the local tuple space to identify the neighbor closest to the intended destination, then using a remote **out** operation to forward the data. The same processing is carried out on the target node, where the

same reaction has been previously installed. This allows to deliver the relevant data to the base station through a sequence of reaction firings and **out** operations.

Multi-hop Routing in TeenyLime. The routing component in object tracking already gives evidence of TeenyLime's ability to implement multi-hop communication. However, to better investigate this aspect, we implemented Mutation Routing [156]: a protocol to route messages from a moving source to a moving destination along a multi-hop path. Mutation Routing has been implemented atop Hood [21], a programming abstraction where nodes in a neighborhood share data based on the application interests. As Hood was motivated by scenarios like Mutation Routing, the comparison is worth to be explored, as discussed in Section 5.6.

In Mutation Routing, a field of fixed sensors is deployed, and two of them are appointed the roles of source and destination. The former sends data to the latter through a multi-hop path. For instance, the source may be the sensor node closest to a moving object, while the destination may be the sensor node closest to a different moving object trying to pursue the former. The source sends periodic updates on the moving object to the follower. The source/destination role must be passed between neighboring nodes as the objects move, making it difficult to maintain a multi-hop route connecting source to destination. Indeed, as the new source (destination) is assumed to be in communication range of the former one, a trivial solution may be to use the old source (destination) as the upstream (downstream) node. However, this may result in inefficient, snake-like routes. Mutation Routing tries to adjust these routes by overhearing messages sent within the same neighborhood. Two techniques are employed, one for detecting the possibility to setup a local shortcut and get rid of loops, the other to build a new path when the current one has many redundant links.

To implement Mutation Routing in TeenyLime, we separated out the task of passing the source (destination) role between neighboring nodes, and re-expressed it as the passing of a token associated to a specific role. Interestingly, to implement this processing we reused the token-based mechanism discussed in our reference application. Therefore, every time the current source (destination) wants to pass its role, it makes the token available. This will be picked by the neighboring node interested in becoming the new source (destination). The routing mechanism in itself is fairly simple: each node has a pointer to a neighbor which considers its upstream node, and is associated to a cost value (e.g., the number of hops traversed) increasing

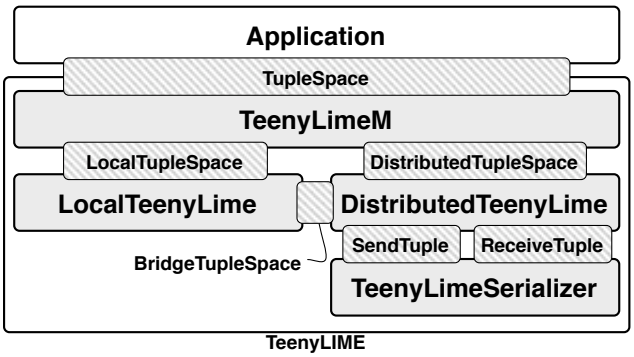


Figure 5.10.: TeenyLime component configuration.

along the source-destination path. Both information are part of the `Node` tuple. A reaction similar to that used for routing in object tracking is installed locally, and fires when a neighbor outputs a message tuple. The local tuple space is then queried to gather the neighbors' position on the route. With this information, the node processing the message can realize, for instance, that some intermediate nodes are no longer needed and therefore update its upstream node, thus shortcutting the path.

5.5. The TeenyLime Middleware

Here we briefly describe the internal architecture of the middleware, as well as the implementation of the most relevant functionality behind the API in Figure 5.3

5.5.1. Architecture

The design of TeenyLime aims at enabling easy customization and extension of the middleware. Therefore, *local* processing, *distributed* processing, and *communication* concerns are fully decoupled, and one aspect can be changed without impact on the rest of the system. The component configuration within the middleware is illustrated in Figure 5.10. The `TupleSpace` interface is provided[3] by a `TeenyLimeM` component delegating the operations either to the `LocalTeenyLime` or `DistributedTeenyLime` component,

[3]Here, the meanings of "providing" and "using" are the ones defined in the TinyOS programming model [28].

depending on the operation target. The former essentially provides storage space for the local tuple space, and performs tuple matching. The latter is responsible for distributed operations (e.g., maintaining the set of remotely installed reactions) and exchanges data with `LocalTeenyLime` component through the `BridgeTupleSpace` interface. The actual communication is implemented in the `TeenyLimeSerializer` component which marshalls and un-masrshalls TeenyLime operations into TinyOS messages.

5.5.2. Implementation

Here we focus on the most relevant aspects of our implementation, namely, *distributed reactions*, *capability tuples*, and the customizable *matching semantics*. Moreover, we discuss our design choices w.r.t. the support for *reliable operations*.

The implementation of *remote reactions* currently rely on a *soft-state* approach to deal with nodes joining or failing. Each node periodically sends a message containing control data for all reactions that should be installed on its neighbors. Upon receipt of this message, a timer associated with installed reactions is refreshed. If and when a timer expires, the corresponding reaction is removed. This may happen either because the registering node became unreachable, or the application deregistered the reaction thus no longer refreshing it. Similar approaches are widely used in WSN, (e.g., in [37]), as they are sufficiently lightweight and effective.

The *matching semantics* is completely decoupled from the rest of the implementation. Our implementation includes the standard value- and type-based matching as well as some TeenyLime-specific semantics, such as range matching described in Section 5.4. However, developers are free to modify or add new field matching mechanisms to better meet their requirements. To do so, they only need to define an additional constant to distinguish the new matching criteria, define the format of a customized tuple field if needed, and implement a boolean function that takes two fields as parameters and returns whether the former matches the latter according to the required semantics.

Processing *capability tuples* requires keeping track of the source nodes whose query matched a local capability tuple so that, once the actual tuple is (locally) output by the application, it can be returned to the appropriate node. Due to nesC split-phase operations [28], this processing requires a lot of bookkeeping code. However, we noted that this processing is the same *as if* a reaction (for the same pattern as the query) were installed

by a neighbor *before* the application outputs the actual tuple. Our implementation leverages off this observation and installs a local reaction for the query pattern before firing the `reifyCapabilityTuple` event. When the node outputs the tuple, this matches the aforementioned reaction and is subsequently, automatically delivered to the intended recipient. The only additional processing required is to remove the reaction right after it fires. This solution only requires 24 nesC lines.

Finally, TeenyLime poses only a single requirement on the communication layers: the ability to overhear messages for populating the tuple space with `Node` tuples. As a result, many existing solutions (e.g., [69, 71]) can be employed to provide *reliable operations*. Nevertheless, if reliability is only seldom required, the solutions above may be overkill, e.g., because scheduling mechanisms (as in [71]) negatively impact latency. To meet scenarios where reliable operations are rare, our current prototype includes a simple reliability scheme based on explicit acknowledgments. Messages contain a unique identifier, reported in the corresponding acknowledgment when transmission succeeds. Therefore, lost packets are easily recognized and retransmitted upon timeout expiration. Control information is piggybacked on application messages whenever possible, to reduce overhead.

5.6. Evaluation

In this section, we compare quantitatively TeenyLime against common alternatives, analyzing its impact on the application source code and on run-time performance.

5.6.1. Evaluating the Programming Model

Our objective is to assess the effectiveness of TeenyLime in enabling a flexible design and clean implementations. To the best of our knowledge, there are no programming abstractions expressly designed for application scenarios such as sense-and-react. Therefore, we compare a TeenyLime-based implementation of our reference application against one implemented on top of TinyOS. On the other hand, the applicability of TeenyLime goes beyond sense-and-react applications, and reaches into layers below the application. We substantiate this claim by reporting about implementations in both TeenyLime and Hood [21], a programming abstraction designed around similar requirements.

```
bool pendingMsg, pendingReading;
TOS_Msg sendMsg, queueMsg[MAX_QUEUE_SIZE];
uint8_t nextQueueMsg, lastQueueMsg;
nodeInterest interests[MAX_AIR_CONDITIONERS];
void interest(uint16_t node, uint8_t t,
              uint16_t tShold, uint16_t tStamp){ // ... }
bool isRecipient(struct InterestMsg* msg,
                 uint16_t nodeId) { // ... }
bool matchesInterest(uint16_t reading) { // ... }
bool enqueueMsg(TOS_Msg msg) { // ... }
bool messageWaiting() { // ... }
bool sendQueuedMsg() { // ... }
command void Boot.booted() {
  // ... data initialization ...
  return call SensingTimer.start(TIMER_REPEAT, SENSING_TIMER);
}
event result_t SensingTimer.fired() {
  pendingReading = TRUE;
  return call TemperatureSensor.getData();
}
event TOS_MsgPtr ReceiveInterestMsg.receive(TOS_MsgPtr m) {
  struct InterestMsg* payload = (struct InterestMsg*) m->data;
  if (!pendingReading && isRecipient(payload, TOS_LOCAL_ADDRESS))
    interest(payload->sender, payload->type,
             payload->threshold, payload->timestamp);
  return m;
}
event result_t TemperatureSensor.dataReady(uint16_t reading){
  TOS_Msg msg;
  struct DataMsg* payload = (struct DataMsg*) msg->data;
  payload->sender = TOS_LOCAL_ADDRESS;
  payload->type = TEMPERATURE;
  payload->value = reading;
  if (!pendingMsg && matchesInterest(reading)) {
    pendingMsg = TRUE;
    sendMsg = msg;
    if (call SendDataMsg.send(TOS_BCAST_ADDR, sizeof(struct AppMsg),
                              &sendMsg)!= SUCCESS) {
        pendingMsg = FALSE;
    }
  } else if (pendingMsg)
    enqueueMsg(msg);
  pendingReading = FALSE;
  return SUCCESS;
}
event result_t SendDataMsg.sendDone(TOS_MsgPtr msg,
                                    result_t success) {
  if (msg == sendMsg) pendingMsg = FALSE;
  if (messageWaiting()) sendQueuedMsg();
  return SUCCESS;
}
```

Figure 5.11.: A temperature node in our reference application, using plain TinyOS. The processing above is equivalent to the TeenyLime version in Figure 5.6.

Component	Explicit states		Lines of code		
	TeenyLime	**Plain TinyOS**	**TeenyLime**	**Plain TinyOS**	**% of application data in TeenyLime**
AirConditioner	3	8	93	282	72%
MutualExclusion	(ML × 2)	(ML × 3) + 1	153	205	48%
TemperatureSensor	0	NC + 2	44	107	100%

Figure 5.12.: Comparing the TeenyLime-based implementation against TinyOS. ML represents the maximum number of co-located air conditioners needing to exchange the same token tuple, NC represents the maximum number of air conditioners around a temperature sensor.

Reference Application. In the TinyOS version of our reference application, each type of node (e.g., temperature sensors or air conditioners) has a component configuration similar to the one mentioned in Section 5.4, where however TeenyLime is replaced by the TinyOS `GenericComm` component[4]. However, the TinyOS-based implementation is far more complex. The reader can informally verify this statement by visually comparing the *excerpt* of TinyOS code for a temperature sensor in Figure 5.11 against the *complete* (and *much* simpler) TeenyLime-based equivalent shown earlier in Figure 5.6. The superior expressive power of TeenyLime manifests itself in several aspects:

- Developers using plain TinyOS must keep track within the application code of all the potential data consumers. This requires several dedicated functions, such as `matchesInterest()` in Figure 5.11. Using TeenyLime, the same functionality is achieved using *reactions*: no application-level bookkeeping is required.

- Figure 5.11 contains two separate execution flows: one begins when a message is received (`ReceiveInterestMsg.receive`), the other when a reading from the sensing device is ready (`TemperatureSensor.dataReady`). These two flows are not at all evident in the code, due to nesC split-phase operations [28]. Thus, maintenance and debugging

[4]Or with our reliability component if reliable interactions, not natively supported by TinyOS, are required by the application. We elaborate further on reliability in Section 5.6.2.

are greatly complicated [64]. This problem is significantly alleviated using TeenyLime, as only the latter execution flow is necessary.

- Distributed processing forces TinyOS programmers to delve into the details of message transmission, parsing, and buffering, therefore mixing communication aspects with the application semantics. Instead, the TeenyLime component in Figure 5.6 contains only application-specific processing related to the actual data of interest.

- As a consequence of all the above, TinyOS programmers must manage *state variables* to deal with nearby air conditioners (`interests`), the sensing device (`pendingReading`), and the radio (`pendingMsg`). These can easily be source of race conditions [28]. Conversely, in TeenyLime these aspects are either handled by the middleware, or no longer required.

A good way to assess the complexity of implementations is to analyze them as state machines and count the number of *explicit application states*, as in [21]. These are typically stored in state variables, modified by commands and event handlers to express state transitions. The higher the number of application states, the harder it is to express state transitions [64], and the more complex and error-prone applications become.

Figure 5.12 reports this and other metrics for the temperature sensor and other components of our sense-and-react application, showing that the advantages of TeenyLime hold for all the (diverse) tasks of our application. For instance, the plain-TinyOS component implementing the air conditioner control law has 8 explicit application states, whereas the TeenyLime-based one has only 3. The reduction is due to the aforementioned ability of TeenyLime to hide communication details, here complemented by the ability to express data filtering as patterns. The former avoids the use of several state variables, while the latter delegates most of the data processing to the middleware. Nicely, the reduction of explicit states in the application code causes the *number of lines of code* to decrease as well, as shown in the second column of Figure 5.12. Indeed, fewer state transitions, and therefore far less bookkeeping code, are needed.

It is worth noting that the above simplifications are *not* accomplished by *removing* application information. Doing so would indeed affect the application semantics. Rather, they are obtained by *moving* information and related processing from the application components into TeenyLime. This is not possible using plain TinyOS, as its abstractions provide only message

passing and do not explicitly represent *state*. This is instead achieved in TeenyLime using the tuple space, as its content is persistent. For instance, a reading tuple output by a temperature sensor node represents its current state and remains in its tuple space until a new reading becomes available.

To quantify this aspect, the rightmost column in Figure 5.12 indicates the amount of information that can be moved from the application component into TeenyLime, expressed as the percentage ratio between the TeenyLime-based and the TinyOS-based applications. We compute it by looking at the per-component storage of *global variables* concerned with application data. The results confirm the reasoning above, showing that a considerable portion of the application state can be managed inside the middleware. Remarkably, *all* the application data and related processing for a temperature sensor can be moved into the tuple space, as shown by comparing Figure 5.6 and 5.11.

The advantages above come at the price of a slight increase in the size of the binary code deployed on the motes. Considering a MICA2 mote as compilation target, the code of a temperature node occupies 29 Kbytes using plain TinyOS and 38 Kbytes using TeenyLime (including the middleware itself). These figures increase to 33 Kbytes and 41 Kbytes, respectively, for the air conditioner. We note, however, that the latter is a complex component, and yet it remains well within the limits imposed by commercially available sensor platforms (e.g., 128 Kbytes for MICA2).

As for the occupation of random access memory on the WSN node, our TeenyLime implementation is fully configurable in terms of i) maximum number of tuples stored in the local tuple space, ii) maximum number of registered reactions, and iii) maximum number of tuples packed in a single physical message. As of today, our most efficient implementation targeting the TMote Sky node [73] can deal with about 100 tuples with 4 fields each, keep about 10 active reactions, and pack at most 5 tuples in a physical message while consuming less than 4 Kbytes of RAM. Considering that TinyOS occupies about 1 Kbyte, 5 Kbytes are still available to the application out of the 10 Kbytes available on a TMote Sky in total. As most of the application data will be managed by TeenyLime itself, as observed above, we maintain that this performance well addresses the limitations of today's WSN hardware.

Sense-only applications and system services. TeenyLime provides relevant benefits also to the development of sense-only applications and system-level functionality. We support this statement by illustrating in-

sights obtained by re-implementing some of the applications used in [21] to evaluate Hood, a programming abstraction geared towards sense-only applications and system mechanisms that, like TeenyLime, focuses on one-hop interactions. Notably, by limiting ourselves to sense-only (instead of sense-and-react) applications, and comparing against Hood on the same applications used for its evaluation, we put ourselves in the most challenging situation.

Specifically, we consider the object tracking application and the multi-hop routing protocol we described in Section 5.4.2. In these applications, the evaluation using the same *quantitative* metrics considered earlier for plain-TinyOS applications shows that TeenyLime achieves performance comparable to Hood. Nonetheless, we can also draw *qualitative* considerations showing that TeenyLime yields cleaner and more reusable designs:

- TeenyLime achieves a *more flexible software architecture* w.r.t. Hood. In Hood, the three components implementing object tracking need to be wired together using dedicated nesC interfaces. Therefore, adding a further component (e.g., to log the position of the moving object on external memory) requires modifications in several places. Instead, in TeenyLime the three components are fully *decoupled*, and exchange data anonymously through the local tuple space. Thus, adding a logging component can be easily achieved without affecting the rest of the application.

- TeenyLime fosters *code re-use* to a great extent. In Hood the processing to pass the source (destination) role between neighboring devices is interspersed with message processing, preventing its reuse. In a TeenyLime-based implementation, this processing can be accomplished by reusing *as is* the component implementing the token-based, mutual exclusion mechanism described in Section 5.4. Simply, we create a token for each role at system start-up, exchanged based on the presence of the moving target close to a given node.

- TeenyLime's one-hop shared tuple space and associated operations are sufficiently powerful to express *multi-hop mechanisms*. In both Mutation Routing and the geographical routing component of object tracking, messages are easily described as tuples. At each hop, these are output to the tuple space of the next-hop node, where a previously-installed reaction delivers the tuple to the routing component. There, the subsequent forwarding to the next-hop node is

determined based on the status of neighboring devices, as reflected by the information locally available in the tuple space. As a result, all the routing decisions are encapsulated in the `tupleReady` event handler. This provides an easy and clean way to implement this functionality that cannot be achieved in Hood due to the absence of abstractions to describe the node state.

The observations above confirm that TeenyLIME's benefits in terms of better design and simpler code hold not only for the development of the application logic in sense-and-react scenarios, but also for sense-only applications and system services.

5.6.2. Evaluating the Middleware Implementation

To verify that the advantages we identified do not negatively affect the system performance, we extend our evaluation beyond the programming model, into TeenyLIME's implementation. Specifically, a middleware layer may impact the *network overhead* and *execution time*, due to additional processing w.r.t. a plain TinyOS implementation. As a consequence, the *system lifetime* may decrease as well. The latter is key in WSNs, as nodes are usually battery-powered and must operate unattended for long periods.

To investigate the above concerns, we conducted experiments using Avrora [157], an instruction-level emulator for WSNs equipped with a precise energy model. The latter is based on experimental data relative to MICA2 [72] nodes, a widespread hardware platform for WSNs. This approach allows us to gather realistic, fine-grained statistics regarding the energy consumption of arbitrary nesC code. We consider two benchmarks:

- The HVAC sub-task we illustrated in Section 5.2, whose TeenyLIME implementation is described in Section 5.4.1. We place a variable number of temperature/humidity sensors in the same neighborhood as an air conditioner node. Every 10 seconds, each temperature sensor randomly generates a reading, whose value can deviate from the user preference with a 20% probability. This triggers actuation at the air conditioner controller, which first queries nearby humidity sensors for their most recent reading, and then decides on the specific actions to be taken.

- A simple application using the token-based, mutual exclusion component illustrated in Section 5.4.1. A variable number of nodes, in

the same neighborhood, express the intention to obtain the token. Every 10 seconds the token is released by the node holding it, and a different, randomly chosen node is selected as the new token holder.

Both applications above involve several TeenyLime-specific constructs. In the first one, a temperature reading may trigger a remote reaction previously installed by the air conditioner, whose pattern contains a dedicated range field to express the user preference as a temperature interval. Moreover, humidity values are represented as capability tuples. Therefore, the (unreliable) query coming from the air conditioner triggers the execution of the `reifyCapabilityTuple` event on the humidity sensors. These react by locally outputting the actual tuple[5], which is delivered by TeenyLime to the air conditioner as the result of the initial query. Similarly, in the mutual exclusion application, releasing the token entails outputting a token tuple in the local tuple space, and possibly triggering some previously installed, remote reaction. Nodes receiving this notification then perform a reliable **in** operation to obtain the token. Among them, only one succeeds.

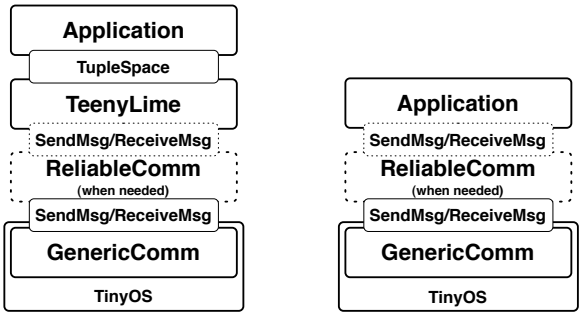
The processing above is the same in other scenarios where the data involved have different semantics. For instance, the processing to exchange the token (i.e., a reaction firing followed by a reliable query) is the *same* executed by a water sprinkler in the fire sub-task, shown in Figure 5.4: only the tuple content changes. In this sense, the meaning of our results extends beyond the benchmark applications we consider here.

For comparison, we consider a plain TinyOS implementation of the same applications. Figure 5.13 illustrates the component configurations in the two cases. To compare them on common ground when required, we provide TinyOS with reliable communication by using our reliable protocol, mentioned in Section 5.5.2.

The emulation settings, in Figure 5.14, are taken from real MICA2 motes. The larger message size in TeenyLime is due to the additional control information contained in the tuples. As independent variables, we vary the number of nodes in a neighborhood and the probability ε of losing a message, to investigate TeenyLime's overhead w.r.t. system scale and network conditions.

Results. In our benchmark applications, TeenyLime does not generate any increase in the *number* of messages exchanged w.r.t. a TinyOS-based implementation. Therefore, TeenyLime's overhead in execution time is es-

[5]Gathering of physical readings from the sensor device is assumed to be instantaneous.



(a) TeenyLime-based. (b) TinyOS-based.

Figure 5.13.: Component configurations.

Parameter Name	**Value**
MAC Layer	standard TinyOS MAC for CC1000 chip
Initial Energy Budget	$\approx$ 2 AA batteries
Message Size	47 bytes (TinyOS), 74 bytes (TeenyLime)

Figure 5.14.: Emulation parameters.

sentially due to extra *local* processing. In this respect, Figure 5.15 analyzes the CPU time taken to perform a set of relevant operations in our benchmark applications. The worst case accounts for a 10.08% overhead, which seems reasonable given the absolute values involved. We believe these results are due to the generality of TeenyLime's abstractions. These can capture commonly-used sequences of operations in a natural way, which allows our TeenyLime implementation to perform close to application-specific mechanisms.

Figure 5.16 further elaborates on the timing aspects in our TeenyLime implementations, showing the breakdown of CPU time in the different layers. Figure 5.16(a) illustrates the aforementioned metric for an air conditioner node in the HVAC application, against the number[6] of temperature/humidity nodes in its neighborhood. TinyOS is responsible for most of the processing, as it handles all hardware interrupts and radio-related functions, triggered quite frequently. The trend of the processing dedicated to the application and to TeenyLime is due to the number of notifications

[6]Half of the nodes in the x-axis are temperature nodes, while the other half are humidity nodes.

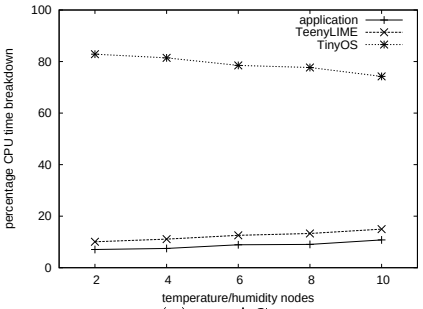
Operation	TeenyLime	Plain TinyOS	Overhead
Notifying the Air Conditioner	2.18ms	1.99ms	9.54%
Sending a Humidity Query	1.97ms	1.85ms	6.48%
Replying to a Humidity Query	2.25ms	2.03ms	10.08%

(a) HVAC.

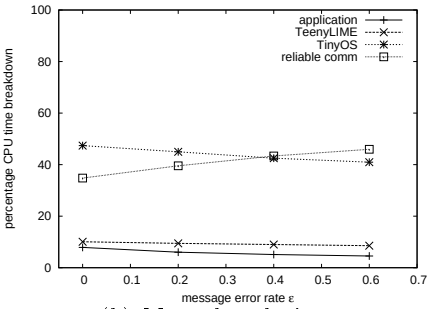
Operation	TeenyLime	Plain TinyOS	Overhead
Releasing the Token	2.03ms	1.97ms	3.04%
Sending a Token Notification	2.28ms	2.07ms	8.21%
Requesting the Token	2.09ms	1.92ms	8.85%

(b) Mutual exclusion.

Figure 5.15.: Execution times in the components of our benchmark applications.



(a) HVAC.

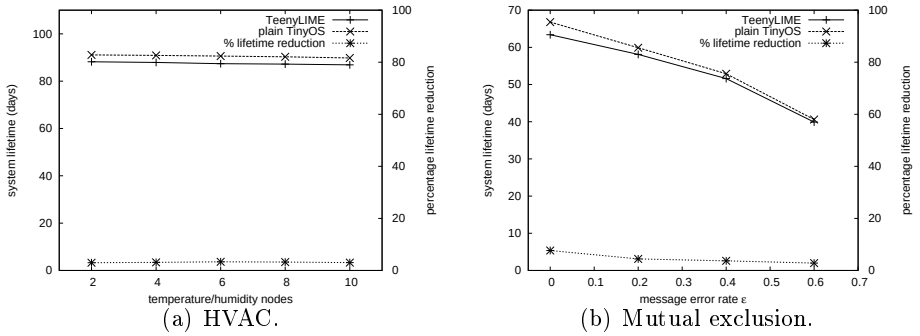


(b) Mutual exclusion.

Figure 5.16.: CPU time breakdown in TeenyLime-based implementations.

and query replies received at the air conditioner, that grows with the number of nearby nodes. TeenyLime engages the CPU at most 15% of the time, when 10 nodes are in reach of the air conditioner. The above metric is not directly affected by the message error rate in the HVAC application, as reliability guarantees are not required.

Conversely, when reliability is required it becomes the dominant factor, and system scale bears little effect on our metrics. Figure 5.16(b) analyzes the CPU time breakdown in the mutual exclusion application against a varying message error rate, with eight nodes in the neighborhood. The chart indeed shows how the reliability protocol increasingly engages the CPU as communication becomes less reliable. In fact, our reliable protocol runs periodic activities (e.g., checking whether messages not yet acknowl-



(a) HVAC. (b) Mutual exclusion.

Figure 5.17.: System lifetime.

edged need a retransmission) that take a time proportional to the number of buffered messages. In absolute values, TeenyLime execution times remain the same regardless of mutable network conditions. Therefore, its relative contribution decreases as the reliable protocol is more stressed. This is a result of our design: TeenyLime and the reliable communication component are fully decoupled, and the processing implemented in the former is independent from the latter.

It is interesting to look at how TeenyLime affects the overall system lifetime. Figure 5.17(a) shows the time until the air conditioner node in the (unreliable) HVAC application runs out of power. This metric is only marginally affected by TeenyLime, whose additional overhead is always under 4%. The chart also illustrates an almost constant behavior w.r.t the number of temperature/humidity nodes. This is expected: reactions and queries are issued in broadcast by the air conditioner, therefore the energy expenditures for communication are independent of the number of neighbors. Conversely, the number of temperature/humidity sensors affects the local processing, since more neighbors correspond to more replies received—as we already observed in Figure 5.16(a). Nevertheless, the extra overhead imposed by TeenyLime has a very limited impact on the overall lifetime. Along the same lines, Figure 5.17(b) shows the lifetime in the (reliable) mutual exclusion application, measured as when the last node depletes its battery. The trends here are strongly tied to the message error rate: an increasing number of retransmissions are indeed required as communication becomes less reliable. TeenyLime's overhead, however, is comparable to the HVAC application, and becomes less relevant as the

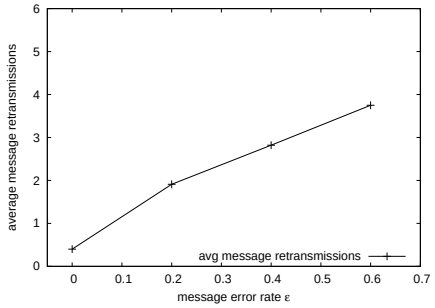


Figure 5.18.: Performance of TeenyLime reliable protocol.

probability of losing a message increases and, consequently, the reliable protocol is involved more.

Finally, we analyzed our reliable protocol, to verify that our results are not biased by an inefficient implementation. Instead, Figure 5.18 shows that our solution can provide 100% message delivery with a very small number of retransmissions. This performance is in line with alternative reliability mechanisms in the literature [66], and therefore confirms that our reliable protocol is a valid choice in our evaluation.

In conclusion, the trade-offs between the benefits of the programming model and its run-time overhead are reasonable, making TeenyLime an effective middleware solution for WSNs.

5.7. Related Work

TeenyLime is inspired by Lime [158], which originally introduced the notion of shared tuple spaces in mobile ad hoc networks. However, not only is TeenyLime's implementation based on entirely different technologies and mechanisms from Lime, but its model and API introduce novel concepts geared expressly towards WSNs, such as range matching, capability tuples, freshness, and explicit control over reliability. TeenyLime follows in time another adaptation of Lime to WSNs, called TinyLime [159]. The two, however, profoundly differ in target scenario, model, and implementation. TinyLime focuses on mobile data collection and employs the standard Lime middleware to provide data sharing over 802.11 among mobile sinks (the data consumers) that, in turn, gather data from nearby WSN sensor nodes (the data producers). Therefore, intelligence is on sinks: the TinyLime

code deployed on sensors is "dumb" and largely application-agnostic, reporting data to external sinks (its only interlocutor) on request. Instead, TeenyLime is expressly designed for scenarios where the application intelligence is in the network, built around node-to-node interactions inside the WSN.

In the context of WSN programming, the work most closely related to TeenyLime is Hood [21], a neighborhood abstraction where nodes can share state with selected one-hop neighbors. Selection is based on attributes periodically broadcast by neighbor nodes. Neighborhoods are specified using extensions to the basic nesC constructs, precompiled into plain nesC. Therefore, unlike TeenyLime, in Hood data sharing is decided at compile-time. Moreover, Hood provides neither the ability to affect the state of another node nor the abstractions to *react* to changes in the shared state. This hampers its use in sense-and-react applications.

In Abstract Regions [14] $\langle key, value \rangle$ pairs are shared among nodes in a region (i.e., a set of topologically-related nodes), and manipulated through read/write operations. Similarly to Hood, there is no way to receive notifications when some given data appears in the system, unlike TeenyLime. Moreover, although nodes in a region may leverage multi-hop communication, this and other aspects must be coded explicitly by the programmer on a per-region basis.

Context Shadow [160] exploits multiple tuple spaces, each hosting only locally-sensed information representing a given context. Applications retrieve the data of interest by explicitly connecting to one of them. Therefore, even if the system is inherently context-aware thanks to the locality of information, the application is likely to increase in complexity due to the absence of any federated data space. Similarly, the tuple spaces used in Agilla [18] for coordinating among mobile agents are shared only local to a node. Instead, TeenyLime enables data sharing in a neighborhood by creating the illusion of a single address space. Moreover, these systems lack WSN-specific tuple space constructs.

Finally, reliable communication in WSNs is an active field of research. Solutions have been proposed both at the network and at the MAC layer. In the former case, reliability is commonly achieved by making data redundant, as in [161], or with per-hop feedback techniques, e.g., [154]. Differently from this work, these solutions target multi-hop routes leading to one or more base-stations. The requirement in TeenyLime is instead of supporting reliable communication within a neighborhood, and in the

absence of a constant data rate. At the MAC layer, reliability is usually provided by sophisticated transmission scheduling algorithms, e.g., as in [70, 71]. However, these solutions often make fairly strong assumptions on constant transmission rates and the data path in the network [66]. This rules out their use in our scenarios, where the communication patterns are hard to predict due the presence of decentralized computation and reactive operations are triggered in response to application-specific events.

Part III.

From Physical to Logical Neighborhoods

6. The Logical Neighborhood Abstraction

As we mentioned in Chapter 1, in sense-and-react applications multiple tasks need to run concurrently, each having different sets of sensors as input, and directing the outputs to a different actuator nodes. Therefore, programmers need to focus on subsets *of devices, as opposed to single devices or the whole network. In these settings, new programming abstractions are required to manage complexity without sacrificing efficiency.*

To tackle the issue above, in this part of the thesis we depart from programming individual devices by providing a foundation to deal with subsets of nodes. To this end, we present the Logical Neighborhoods *programming abstraction. A logical neighborhood includes nearby nodes that satisfy predicates over their static (e.g., type) or dynamic (e.g., sensed values) characteristics. Logical neighborhoods enable the programmer to "illuminate" different areas of the network according to the application needs, effectively replacing the physical neighborhood provided by wireless broadcast with a higher-level, application-defined notion of proximity. In this chapter, we present the definition of a declarative language for specifying logical neighborhoods, highlighting its expressiveness and simplicity. Instead, Chapter 7 illustrates a dedicated routing scheme in support of Logical Neighborhoods. The contributions discussed in this part of the thesis appeared in [9, 10, 162, 163].*

6.1. Introduction

As outline in the introductory chapter, WSNs involving actuation [2] are increasingly employed to implement sophisticated monitoring and control systems. In these scenarios, the system not only observes and gathers data from the environment, but is also capable of performing a variety of

actions on the physical world. Therefore, in similar scenarios nodes are intrinsically heterogeneous and collaborate in a decentralized fashion to carry out a complex task, e.g., home automation [26] or chemical attack detection [1].

Similar applications are often composed of many collaborating sub-tasks, each affecting only a given part of the overall system. For instance, consider a fire detector and controlling system deployed in a building. Actuator nodes control water sprinklers and monitor the values of temperature sensors and smoke detectors nearby (e.g., in the same room). When a significant fraction of these sensor nodes reports high values to an actuator, it immediately activates the sprinkler it controls, and alerts the actuators nearby (e.g., in neighboring rooms). Programming such an application is complex, as the developer needs to worry not only about the implementation of the application logic, but also about which *subset* of the system should be involved and how to reach it. As no dedicated programming constructs and mechanisms exist for the latter task, the result is additional programming effort, increased complexity and, in absence of well-established and reusable solutions, less reliable code. Therefore, to manage the inherent complexity of this kind of system, new programming abstractions are needed.

In WSNs, communication is constrained by the range of the wireless media. A node is therefore able to exchange data directly only with the nodes located within its communication radius. These nodes effectively constitute the *physical neighborhood* of a given device. The core programming facilities of many WSN operating systems (e.g., TinyOS [42]) provide mechanisms for managing and exploiting communication to and from nodes in the physical neighborhood, leveraging directly off the underlying network communication.

Differently from the aforementioned solutions, here we address the above issues by introducing the notion of *Logical Neighborhood*. The span of a Logical Neighborhood is not limited by the physical communication range, rather is controlled by the programmer using applicative and contextual information. Logical neighborhoods are specified declaratively using the SPIDEY language we designed, illustrated in the rest of the chapter, conceived to be a simple extension to existing WSN programming languages (e.g., nesC [28] in the case of TinyOS). In this context, Logical Neighborhoods are used as a higher-level communication primitive, to address only the nodes in the system that satisfy the constraints imposed by the

neighborhood definition.

With reference to the example above, an actuator can use SPIDEY to define a logical neighborhood that contains nodes hosting a temperature sensor, reading a value greater than 100^oC, deployed in the same room, and placed at a maximum distance of three hops, and successively use this neighborhood to simply "broadcast" (logically) a message to all the nodes contained in it. By redefining the conventional broadcast-based communication primitives and coupling them to the logical neighborhood instead of the physical one, we provide programmers with a powerful abstraction, where neighboring relations are no longer restrained to the communication in the immediate physical neighborhood.

The benefits of our proposal impact two orthogonal aspects. First, developers can concentrate on the actual application goals while relying on SPIDEY and the associated communication framework as a way to logically partition the system and interact with it. We conjecture that applications built on top of our abstraction result in cleaner, simpler, and more reusable implementations. Second, this strategy opens up opportunities for achieving a longer system lifetime and a better resource utilization, by focusing only on the nodes that actually need to be involved.

The rest of the chapter is organized as follows. Section 6.2 describes the SPIDEY framework and definition language. Next, Section 6.3 illustrates the API made available to programmers for exploiting Logical Neighborhoods. Finally, we conclude the chapter in Section 6.4 by discussing a demonstration case study, illustrating how to develop a road tunnel monitoring and control application using Logical Neighborhoods.

6.2. Programming with Logical Neighborhoods

In this section, we begin with an overview of the basic concepts underlying the notion of Logical Neighborhood, followed by the definition of the SPIDEY language supporting it.

6.2.1. Basic Concepts

The abstraction we propose revolves around only two concepts: *nodes* and *neighborhoods*. As illustrated in Figure 6.1, a (logical) node is the application-level representation of a physical node, and defines which portion of the node's state and characteristics is made available by the programmer to the definition of any logical neighborhood. The definition of

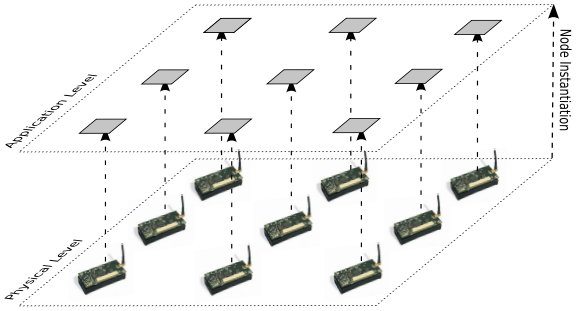


Figure 6.1.: A portion of a node's state and characteristics is exported at the application-level by means of (logical) node instances.

```
node template Sensor
  static Device
  static Type
  static Location
  dynamic Reading
  dynamic BatteryPower

create node ts from Sensor
  Device as "sensor"
  Type as "temperature"
  Location as "room123"
  Reading as getTempReading()
  BatteryPower as getBatteryPower()
```

Figure 6.2.: Sample node definition and instantiation.

a node is encoded in a *node template*, which specifies the node's exported attributes. This is used to instantiate the (logical) node, by specifying the actual source of data. To make these concepts more concrete, Figure 6.2 shows a code fragment specified using the SPIDEY language that defines a node template for a generic sensor, and then instantiates a node with the structure prescribed by the template. During this operation, each attribute in a node template is bound to an expression of the target language, e.g. a variable or a function.

A (logical) neighborhood is the set of nodes satisfying a predicate on the nodes' attributes. As with nodes, the definition of neighborhoods is encoded in a template, which contains the predicate that essentially serves as the membership function determining whether a node is to be included

```
neighborhood template HighTempSensors(threshold)
  with Device = "sensor" and
       Type = "temperature" and
       Reading > threshold

create neighborhood htsn100
  from HighTempSensors(threshold: 100)
  max hops 2
  credits 30
```

Figure 6.3.: Sample neighborhood definition and instantiation.

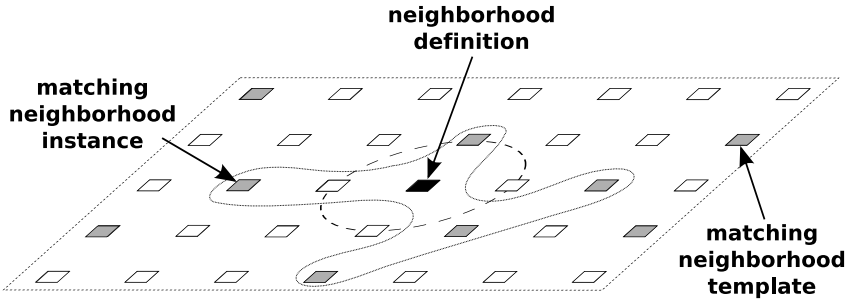


Figure 6.4.: A pictorial representation of the example in Figure 6.3. The black node is the one defining and using the logical neighborhood for communication, and its physical neighborhood (i.e., nodes lying in its direct communication range) is denoted by the dashed circle. The grey nodes are those satisfying the neighborhood template `HighTempSensors` when the threshold is set to 100^oC. However, the nodes included in the actual neighborhood instance `htsn100` are only those lying within 2 hops from the sending node, as specified through the `hops` clause during instantiation.

in the set associated to the logical neighborhood. For instance, the neighborhood template `HighTempSensors` in Figure 6.3 selects nodes that host temperature sensors and are currently reading a value higher than a given threshold. As exemplified in the SPIDEY code fragment of Figure 6.3, a neighborhood template can be parameterized, and the actual values for the parameters are provided when instantiating the neighborhood. Also in this case, each formal parameter can be bound to an expression of the target language.

In addition, the instantiation of a neighborhood template specifies addi-

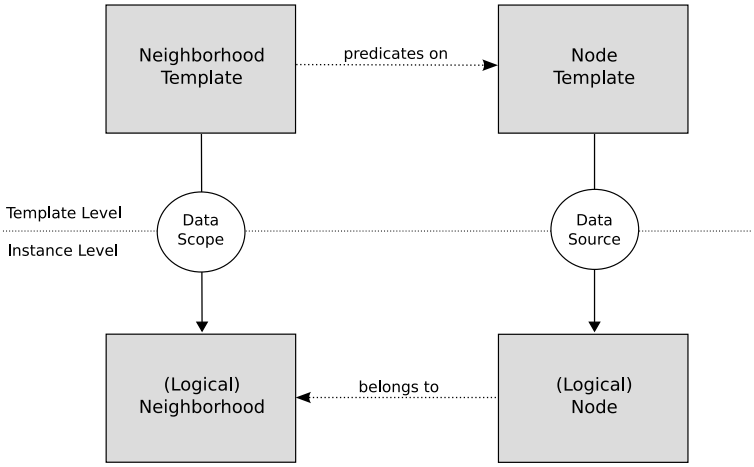


Figure 6.5.: The conceptual relationship between templates and their instantiation.

tional requirements about *where* and *how* the neighborhood is to be constructed and maintained. For instance, the instantiation in Figure 6.3 specifies that the predicate defined in the `HighTempSensors` template is evaluated only on nodes that are at a maximum of 2 hops away and by spending a maximum of 30 "credits". The latter is an application-defined measure of cost, detailed further in the next section, which essentially enables the programmer to retain some degree of control over the resources consumed during the distributed processing necessary to maintain the logical neighborhood abstraction. A pictorial representation of the example, visualizing the logical neighborhood concept, is provided in Figure 6.4.

In essence, as graphically illustrated in Figure 6.5, templates define *what* data is relevant to the application, while the instantiation process constrains *how* this data should be made available by the underlying system. Separating the two perspectives has several beneficial effects. The same template can be "customized" through different instantiations. For instance, the same template in Figure 6.3 could be used to specify a logical neighborhood with a different threshold or a different physical span. Moreover, this distinction naturally maps on an implementation that maintains a neighborhood by disseminating its template to be evaluated against the values exported by a node instance, and uses instead the additional constraints specified at instantiation time to direct the dissemination process.

The simple example we just outlined already shows two distinctive features of our approach:

- In sharp contrast with most existing works, a neighborhood definition does *not* make any assumption as to where the member nodes are geographically located, or whether they are physically close to each other. Indeed, in Logical Neighborhoods programmers reason on a logical plane that is completely decoupled from the physical topology. Therefore, they enjoy great flexibility in slicing the network depending on the application requirements.

- Although a neighborhood definition may not refer directly to properties or characteristics of the node defining the neighborhood itself, in most cases we expect its *instantiation* to do so, albeit implicitly. For instance, the `max hops` construct necessarily makes the selection of the neighborhood members dependent on the physical location of the instantiating node. Therefore, having neighborhoods defined on different nodes with same member sets will be a very unlikely situation.

In light of the above features, implementing the semantics required by our abstraction calls for dedicated routing support, as we discuss in Chapter 7. In the following, we present the complete definition of the SPIDEY language, enabling the definition of node and neighborhood templates and their instantiation.

6.2.2. The Spidey Language

A comprehensive view of the features and constructs of the SPIDEY language[1] is provided by its grammar, shown in Figure 6.6. The grammar, however, represents only an abstract syntax, that needs to be adapted to the characteristics of the target language in which SPIDEY is integrated.

A node template is declared with the `node template` construct, followed by the list of attributes to be exported, as shown in our example in Figure 6.2. An attribute must be declared as either `static` or `dynamic`. Static attributes represent information assumed not to vary in time, e.g., the type

[1] The name SPIDEY is inspired to the comic book series "Friendly Neighborhood Spider-Man", whose title is derived from a self-referential comment often made by Spider-Man: "just another service provided by your friendly neighborhood Spider-Man!" [164].

```
<node_template> ::=
  node template <node_templ_id>
    ({static | dynamic} <field_name>)+

<node_instance> ::=
  create node <node_id>
    from <node_templ_id>
      (<field_name> as <target_lang_expr>)+

<nhood_template> ::=
  neighborhood template <nhood_templ_id>
                        [(<par_name>(,<par_name>)*)]
    [with <node_predicates>]
    [{union | intersect | minus | on} <nhood_templ_id>
                                      [<par_bindings>]]*

<nhood_instance> ::=
  create neighborhood <nhood_id>[<par_bindings>]
    from <nhood_templ_id>
    [[{min | max}] hops <integer_value>]
    [credits <numeric_value>]

<par_bindings> :: =
  (<par_name>:<target_lang_expr>
  (,<par_name>:<target_lang_expr>)*)

<cost_function> ::= use cost <numeric_target_lang_expr>
```

Figure 6.6.: Grammar showing the abstract syntax of the SPIDEY language. `<target_lang_expr>` is any valid expression in the target language that evaluates to a type compatible with the attribute or parameter at hand. `<numeric_targetlangexpr>` further constraints the expression in the target language to evaluate to a numeric type. `<node_predicates>` is any well formed boolean predicate over node attributes.

of measurement a sensor node provides or the kind of system an actuator node controls. Instead, dynamic attributes represent information that by definition changes with time, e.g., the current sensor reading or the residual battery power. Note that the decision about whether an attribute is static or dynamic depends on the deployment scenario. For example, the location of a node physically attached to a wall in a monitoring building application may be well considered as a static attribute. Instead, the location of a node attached to an animal in a habitat monitoring application should be considered as a dynamic information (e.g., being derived from a GPS receiver). Forcing this distinction enables optimizations of the communication strategies supporting our abstraction. For instance, in case of rapidly

changing dynamic attributes, we may resort to routing using only static information, and match neighborhood templates against dynamic attributes locally. More details on the spectrum of available solutions to deal with these aspects are reported in Chapter 7.

The instantiation of a node is achieved with the `create node` construct, which contains a list of `as` clauses establishing the binding between an attribute and a value source. In its simplest incarnation, the `as` clause accepts any expression of the target language. These are evaluated each time a message addressed to a logical neighborhood arrives at a node, and the associated neighborhood template must be matched against the node's values to determine its membership in the neighborhood. Template attributes that are not bound through an `as` clause are not exported by the node instance.

A neighborhood template can be defined using the `neighborhood template` keywords, followed by two optional clauses. To describe the semantics of the construct we refer to the example in Figure 6.7, which defines a template for a logical neighborhood encompassing temperature or smoke sensors reading values over some thresholds and deployed in a given room. The `with` clause is used to define the predicate on node templates that is to be checked against actual node instances to determine whether they belong to the logical neighborhood. Complex predicates can be composed from basic (in)equalities by means of the usual logic operators such as `and`, `or`, and `not` (not shown in the grammar). If the `with` section is omitted, the `true` predicate is implicitly assumed, therefore any node can belong to the logical neighborhood. The symbolic names provided as parameters of the template (the room name as well as temperature and smoke thresholds in our case) can be used on the right-hand side of the neighborhood predicate, with the actual values bound at neighborhood creation time to expressions of the target language. These are evaluated on the originating node each time an application message is sent to a given (logical) neighborhood.

The second section of the `neighborhood template` construct enables the programmer to specify composition among neighborhoods. A logical neighborhood essentially identifies a set of nodes, therefore it is natural to express a neighborhood as a composition with already existing ones, using the conventional set operators. For instance, the example in Figure 6.7 can be rewritten equivalently (and more elegantly) as:

```
neighborhood template HighTempSmokeSensors
                      (room, tempT, smokeT)
  on RoomSensors(room)
```

```
neighborhood template HighTempSmokeSensors
                      (room, tempT, smokeT)
  with Device = "sensor" and Location = room and
         (Type = "temperature" and Reading > tempT)
       or (Type = "smoke" and Reading > smokeT)

create neighborhood htss
  from HighTempSmokeSensors(room: "room123",
                            tempT: TEMP,
                            smokeT: SMOKE)
  max hops 10
  credits 30
```

Figure 6.7.: An example of a complex neighborhood template, where `TEMP` and `SMOKE` are variables in the target language.

```
  intersect HighSensors(type: "temperature",
                        threshold: tempT)
  intersect HighSensors(type: "smoke",
                        threshold: smokeT)
```

assuming that the following definitions exist:

```
neighborhood template HighSensors(type,threshold)
  with Device = "sensor" and
       Type = type and
       Reading > threshold

neighborhood template RoomSensors(room)
  with Device = "sensor" and
       Location = room
```

The `on` operator can be used for defining a neighborhood as a subset of an existing one, while the `union`, `intersect`, and `minus` operators have the usual meaning. The mechanics of implementing these operators are trivial, since they ultimately rely on composing the neighborhood predicates properly (e.g., by disjunction in the case of `union`).

Once a neighborhood template is defined, it can be instantiated by means of the `create neighborhood` construct, as illustrated in Figure 6.7. Instantiation allows the programmer to "customize" the template by specifying the actual parameters to be used in the evaluation of the neighborhood predicate. For instance, we used this feature in defining the template `HighTempSmokeSensors` above, to customize the definitions of `HighSensors` during composition. Note that any valid expression in the target language can be given to bind a parameter to an actual value. In a C-like language, for instance, a parameter can be bound to the return value of a function. This is evaluated each time a message is sent to that

neighborhood, thus giving programmers some degrees of freedom in changing the neighborhood membership dynamically. An example leveraging off this feature is illustrated in our case study, reported in Section 6.4.

Moreover, appropriate clauses enable programmers to retain control on the span of the logical neighborhood. The `hops` clause enables the programmer to limit such span directly, in terms of number of hops from the sending node. Instead, the `credits` clause specifies the maximum *cost* the system can incur in for delivering a message to logical neighbors, as detailed next. If both are specified, the extent of propagation is determined by the `hops` clause only when enough credits are available. If neither are specified, the nodes to be included in the logical neighborhood can reside anywhere in the system.

Communication *cost* is defined in terms of the basic operation of sending a broadcast message to physical neighbors (called the node' s *sending cost*) and is measured in *credits*, whose semantics is application-defined. The mapping between the sending of a single physical message and its cost in credits is indeed specified by the programmer through the `use cost` construct, which delegates the computation of this mapping to an expression of the target language. This way, the programmer can define a vast array of mappings, from a straightforward one where message sending has a fixed cost, to sophisticated ones where the cost varies dynamically to adapt to a node's conditions (e.g., low battery power). Moreover, different nodes can have different functions, therefore enabling the programmer, say, to set higher costs for tiny, battery-powered sensors, and lower costs for resource rich, externally-powered nodes. The overall number of credits necessary to communicate with the members of a logical neighborhood is evaluated as the sum of the costs that each node involved in routing messages incurs in, with each node evaluating its own cost according to the function specified in the `use cost` declaration. Therefore, the ability to set the maximum amount of credits enables programmers to exploit different trade-offs between accuracy and resource consumption. Neighborhoods instantiated with a high number of credits have a broader coverage of the system, but at the expense of a higher number of consumed credits and hence overhead. Notably, the support for credits has been one of our chief motivation for devising a dedicated communication scheme underpinning Logical Neighborhoods. More details on these aspects are provided in Chapter 7.

`void send(Neighborhood n, Message m)`	Send message `m` to all nodes belonging to the logical neighborhood `n`.
`receive(Message m)`	Receive a message.
`void reply(Message r, Message m)`	Reply to message `m` with message `r`.

Figure 6.8.: API for the communication component providing the logical neighborhood abstraction.

6.3. Communication API

The SPIDEY language we just described is used to define and instantiate logical neighborhoods. However, neighborhoods are ultimately used in conjunction with communication facilities, to enable interaction with the neighborhood members. On the other hand, the notion of logical neighborhood is essentially a scoping mechanism, and therefore is independent from the specific communication paradigm chosen. For instance, one could couple it with the tuple space paradigm to enable tuple sharing and access only within the realm of a logical neighborhood. Similarly, logical neighborhoods could be used in a publish-subscribe approach, to properly limit the dissemination of subscriptions and event notifications.

Figure 6.8 shows the API we are currently developing, where we explore the combination of logical neighborhoods with a simple message passing communication model. The `send` operation takes as parameters a variable (instantiated through SPIDEY) representing the logical neighborhood and a message to be delivered to its members. Essentially, we are replacing the broadcast facility commonly made available by the operating system with one where the message recipients are not determined by the physical communication range, rather by membership in a programmer-defined logical neighborhood. The others are only auxiliary operations and enable the application, respectively, to receive a message, and to reply to a message received through the neighborhood.

6.4. Demonstration

To assess the flexibility of the Logical Neighborhoods abstraction, we investigated its use in the context of a sense-and-react application for road tunnel monitoring and control [4]. Researchers are envisioning tunnels equipped with WSN nodes that gather physical readings (e.g., light), monitor the structural integrity of the tunnel, and sense the presence of vehicles

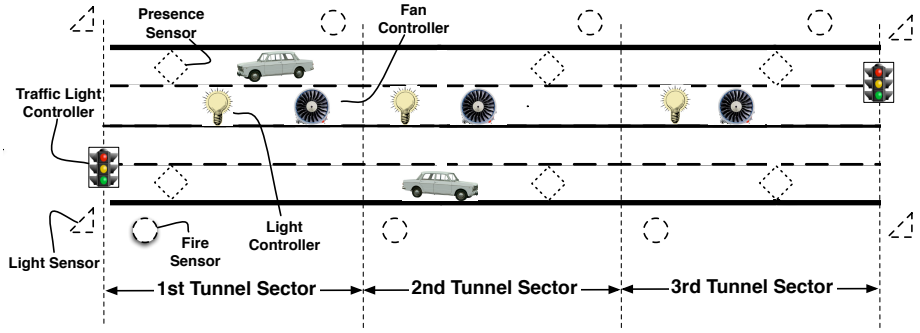


Figure 6.9.: Tunnel scenario.

Figure 6.10.: Setup and nodes controlling fans and lights.

to detect a possible traffic congestion. Based on sensed data, the system operates a variety of devices such as ventilation fans inside the tunnel, and traffic lights at the entrances. For instance, when a sensor detects the presence of fire in a sector, the fans in the same sector are activated, and the traffic lights are turned red to prevent further vehicles from entering the tunnel.

The above scenario presents most of the characteristics motivating Logical Neighborhoods: the system is highly heterogeneous, and the processing

```
neighborhood template TrafficJam(sector, lane)
  with Function = "actuator" and
       ((Type = "traffic_light" and Location = lane) or
        (Type = "fan_controller" and TunnelSector <= sector))

create neighborhood tj
  from TrafficJam(sector : mySector(), lane : myLane())
```

Figure 6.11.: A neighborhood including nodes controlling fans in given sectors or traffic lights on a specific lane.

mostly involves *subsets* of nodes sharing similar characteristics, e.g., all the nodes controlling a fan in a specific tunnel sector. Therefore, the programmer must be provided with appropriate abstractions to "slice" the system based on the application requirements.

To demonstrate the above application, we used 20+ TMote Sky nodes [73] to model three tunnel sectors, as illustrated in Figure 6.9. We decreased the transmission power to create a *multi-hop* scenario in a limited space. As for actuation, we modified some of the nodes to control externally attached devices. Specifically, 12 V mini-fans and lights are used to model the fans inside the tunnel and the traffic lights at the entrances. For practical reasons, fire and presence sensors are "implemented" with light sensors, triggered using flashlights. Our setup is shown in Figure 6.10. Based on this setup, we showcase various use cases involving different logical neighborhood definitions.

6.4.1. Traffic Control

Requirement. When presence sensors recognize a traffic jam on a lane, the fans are activated along the same lane from that location to the corresponding entrance, and the traffic light is turned red only on that lane.

Design and implementation. To meet the above requirement, at all presence sensors we define a logical neighborhood including nodes controlling fans from the node's sector to the corresponding entrance, or traffic lights on the same lane as the node. This is quite straightforward to achieve using SPIDEY, as illustrated by the fragment of code in Figure 6.11. Note how we define a parametrized template, and instantiate it depending on the physical location of the node. Therefore, the shape of the (logical) neighborhood changes depending on the sender, as Figure 6.12 exempli-

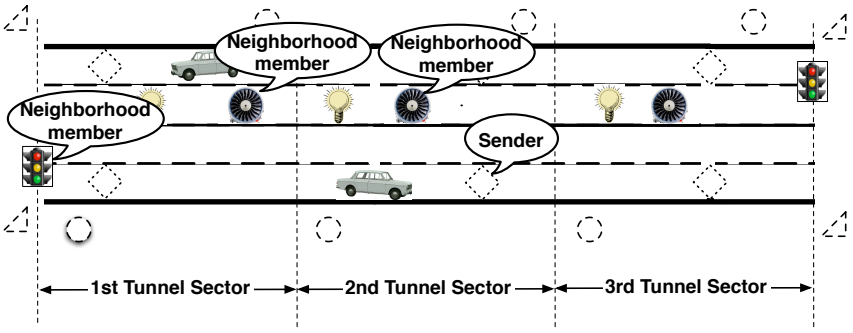


Figure 6.12.: Example of nodes involved in traffic control.

```
neighborhood template LightControllers(depth)
  with Function = "actuator" and
       Type = "light_controller" and
       TunnelSector >= mySector and
       TunnelSector <= mySector + depth

create neighborhood lc
  from LightControllers(depth: getNormalizedLightIntensity())
```

Figure 6.13.: A neighborhood including nodes controlling the lights for a given number of consecutive sectors inside the tunnel.

fies. When a presence sensor recognize a blocked vehicle, it starts sending periodic messages to nodes in the aforementioned neighborhood using our communication API. The receivers parse the data in the message, and decide on the actions to be taken.

6.4.2. Adaptive Lighting

Requirement. When light sensors read values above a safety threshold, the lights inside the tunnel are activated to avoid shadowing effects and improve the visibility to drivers entering the tunnel. The number of sectors inside the tunnel where lights are to be activated must be proportional to the light intensity.

Design and implementation. Implementing adaptive lighting requires programmers to address a different subset of nodes depending on contextual information, i.e., how intense is the light reading. To do so, in Figure 6.13

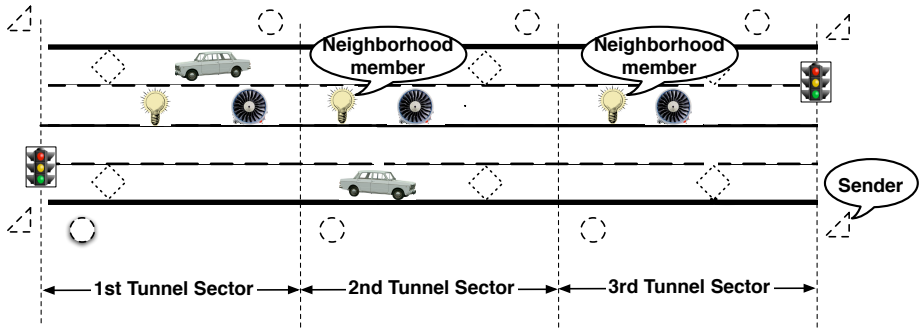


Figure 6.14.: Example of nodes involved in adaptive lighting. `getNormalizedLightIntensity()` returned 2 when the message was sent to the logical neighborhood.

```
neighborhood template Fire(sector)
  with Function = "actuator" and
       (Type = "traffic_light" or
        (Type = "fan_controller" and
         TunnelSector <= sector+1 and
         TunnelSector >= sector-1))

create neighborhood fe from Fire(sector : mySector())
```

Figure 6.15.: A neighborhood including nodes controlling traffic lights or fans in three adjacent tunnel sectors.

we define a neighborhood template with a parameter specifying how many sectors in the tunnel we want to address. In the application code, we query the light sensor periodically. If the current reading crosses a given threshold, the node sends a message to the above logical neighborhood. Before handing the message over to the network stack, the Logical Neighborhood run-time evaluates `getNormalizedLightIntensity()` in Figure 6.13 to determine how many sectors in the tunnel need to be addressed. Based on this value, a different subset of light nodes receives the message, as illustrated in Figure 6.14. This occurs transparently to the programmer, who simply bound one of the template parameters to a function instead of a constant value.

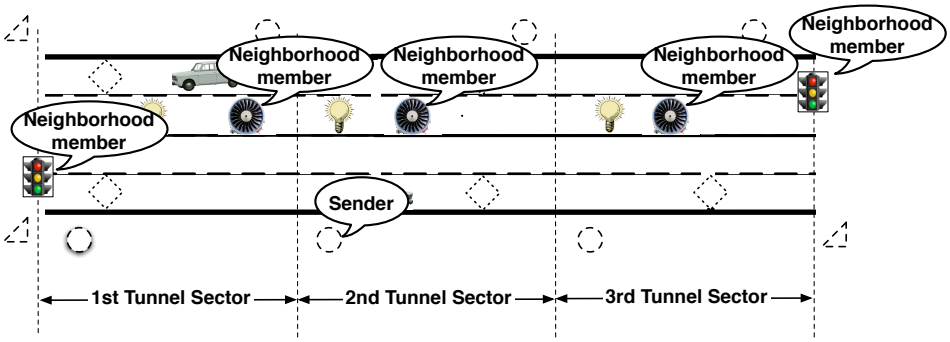


Figure 6.16.: Example of nodes involved in fire control.

6.4.3. Fire Control

Requirement. When fire sensors detect the presence of fire in a sector, the fans in the same and adjacent sectors are activated, and the traffic lights are turned red on both ends of the tunnel.

Design and implementation. Similarly to the traffic control case, upon detecting the presence of fire the sensor nodes send periodic messages to the neighborhood defined in Figure 6.15. The parameters defined in the template allows the neighborhood instantiation to change depending on the sender location, as exemplified in Figure 6.16. In this case, however, the actuators wait to see more than a single sensor reporting fire, as a single reading may signal a malfunctioning node instead of a real fire.

7. Routing for Logical Neighborhoods

The Logical Neighborhood abstraction is ultimately of practical interest only in the presence of an efficient routing mechanism implement its semantics. In principle, existing solutions can be exploited (e.g., [37]), but they exhibit various performance drawbacks, as they are based on different assumptions and scenarios. Therefore, we devised a novel routing protocol that is expressly devised to support our abstraction. Notably, our solution leverages off the kind of localized interactions [153, 165] characterizing the decentralized scenarios we target with Logical Neighborhoods. The corresponding mechanisms are described and evaluated in this chapter. Based on the performance results obtained, we maintain that our routing protocol efficiently supports Logical Neighborhoods, therefore demonstrating the overall feasibility of our approach.

7.1. Motivation and Overview

The Logical Neighborhood abstraction is essentially independent of the underlying routing layer. Nevertheless, its characteristics cannot be easily accommodated by existing routing approaches. Protocols geared towards data collection focus on how to collect efficiently the data from many sensors to a single node—the sink. In our approach the perspective is *reversed*: routing must efficiently transmit a message from a single node—i.e., the device defining the neighborhood—to those matching the neighborhood specification. Moreover, the Logical Neighborhood *abstraction* is essentially a scoping mechanism, and therefore can be used in conjunction with several mechanisms other than data collection, e.g., to direct code updates only towards nodes with obsolete versions, as we describe in Chapter 8. As such, some of the techniques exploited by these protocols (e.g., route

reinforcement based on data rates as in [37]) cannot be directly applied.

In principle, we may try to adapt some multicast protocol for wireless networks to our scenario. One way of doing so is to map the single atomic predicates to individual channels. This solution is trivially inefficient, as even a simple predicate such as `Type = "temperature" or Type = "vibration"` would force the sender node to duplicate data unnecessarily by addressing two distinct groups, even if the target nodes are physically close to each other. Alternatively, we may map the entire neighborhood definition to a multicast group. As programmers are free to use any combination of atomic predicates in defining their neighborhoods, and even compose or intersect them, the number of neighborhoods may be quite large. This would force the underlying protocol to maintain a large number of multicast groups, again with significant overhead. Moreover, adding to a running system a further node defining a previously unseen neighborhood would force the protocol to create a new multicast group on the fly, a functionality that is seldom supported in existing protocols.

Finally, to the best of our knowledge there are no existing communication schemes that implement an application-defined measure of communication cost as we do with credits. Their management is a distinctive feature of our approach that would anyway require appropriate integration.

Motivated by these considerations, this chapter describes a routing strategy to provide a generic, yet efficient communication layer supporting the Logical Neighborhood abstraction. Our routing approach is *structure-less* (i.e., no overlay is explicitly maintained) and is based on the notion of *local search* [166]. Nodes advertise their *profile*, i.e., the list of attribute-value pairs specified by their template, and in doing so they build a distributed state space containing information about the cost of reaching a node with given data. This information dissemination is localized and governed by the density of devices with similar profiles. Messages sent to a neighborhood contain its template, which determines a projection of the state space, i.e., the part to be considered for matching. In a nutshell, the message "navigates" towards members of a neighborhood by following paths along which the cost associated with a given neighborhood template is decreasing.

The following section describes our routing solution, whereas Section 7.3 reports on its performance. We conclude in Section 7.4 by comparing the Logical Neighborhood approach against related work.

Source	Timestamp	Node Profile		Cost
		Attribute	Value	
N54	72	Function	*sensor*	2
		Type	*temperature*	
		Location	*room123*	

Figure 7.1.: An example of ProfileAdv.

7.2. Routing for Logical Neighborhoods

The proposed routing approach is composed of two parts: the *state space generation* and the *search algorithm*. These are individually described next.

7.2.1. Building the State Space

In our scheme, whenever a new node is added to the system it broadcasts a ProfileAdv message containing the node identifier, a (logical) timestamp, the node's profile containing attributes and their values, and a cost field initialized to zero. The first two message fields are used to discriminate stale information, as the ProfileAdv message is periodically re-broadcast (possibly with different content) by the node. An example ProfileAdv is reported in Figure 7.1.

In addition, each node in the system stores a *State Space Descriptor* (SSD) containing a summary of the received ProfileAdv messages. An example is shown in Figure 7.2. The *Attribute* and *Value* fields store information previously received through a ProfileAdv message. For each entry, *Cost* contains the minimum cost to reach a node with the corresponding information, and *Source* contains the identifier of such node. The *Links* field allows to store information more compactly, by retaining associations among entries instead of duplicating them in the SSD. In Figure 7.2 each entry is linked to the others as they all come from the same ProfileAdv advertised by node N8. *DecPath* and *IncPaths* contain routing information to direct the search process, as described in Section 7.2.2. Finally, each entry in an SSD is associated with a lease (not shown), whose expiration causes the removal of the entry if not refreshed by a new ProfileAdv.

Upon receiving a ProfileAdv message, a node first updates the cost field in the message by adding its own sending cost, obtained by evaluating the expression in the `use cost` statement described in Section 6.2.2. Then, it compares each attribute-value pair in the message against the content of the local SSD. A modification (entry insertion or update) of the SSD

Id	Attribute	Value	Cost	Links	DecPath	IncPaths	Source
1	Function	*sensor*	5	2,3	N37	N98, N99	N8
2	Type	*acoustic*	5	1,3	N37	N98, N99	N8
3	Location	*room123*	5	1,2	N37	N98, N99	N8

Figure 7.2.: An example of *State Space Descriptor (SSD)*.

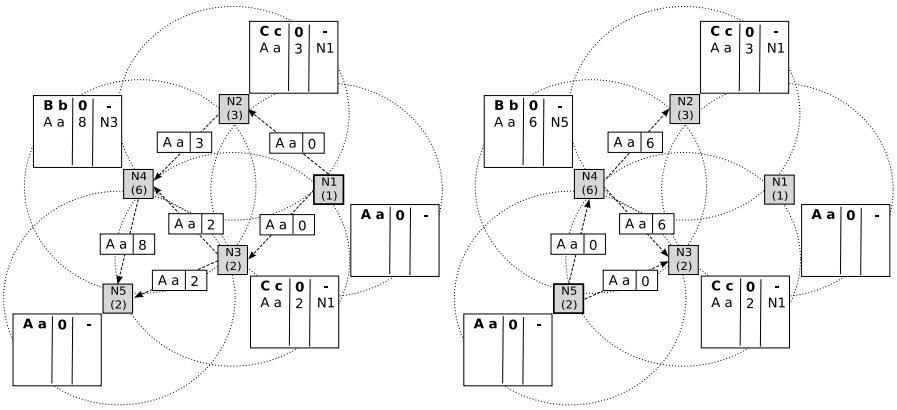
Id	Attribute	Value	Cost	Links	DecPath	IncPaths	Source
1	Function	*sensor*	**3**	3,**4**	**N77**	N98, N99	**N54**
2	Type	*acoustic*	5	1,3	N37	N98, N99	N8
3	Location	*room123*	**3**	1,**4**	**N77**	N98, N99	**N54**
4	Type	***temperature***	**3**	**1,3**	**N77**	-	**N54**

Figure 7.3.: The SSD of Figure 7.2 at a node with a sending cost of 1, after receiving the ProfileAdv message in Figure 7.1.

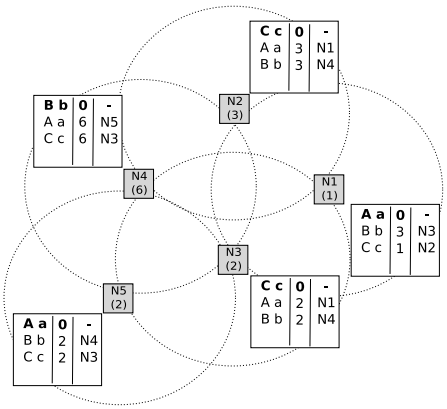
is performed if an attribute-value pair: i) does not exist in the local SSD, or ii) it exists with a cost greater than the one in the message (after the local update above). The update (or insertion) of an SSD entry involves establishing the proper values in the *Links* field to keep track of the rest of the ProfileAdv message, updating the *DecPath* field with the identifier of the physical neighbor that sent the ProfileAdv, and setting the *Source* field to the identifier of the node whose information has been inserted in the ProfileAdv. For instance, assume the node storing the SSD in Figure 7.2 has a sending cost of 1, and receives the ProfileAdv in Figure 7.1. Its local SSD is then updated as described in Figure 7.3 (changes shown in bold). Note how the *Links* fields are updated so that only the minimum cost to reach an entry is kept, and yet the information about which entry came with which profile is not lost.

After a ProfileAdv has been processed locally, it is rebroadcast *only* if at least one SSD entry was inserted or updated, to propagate the state change. An example is shown in Figure 7.4(a). The ProfileAdv is rebroadcast as received, except for the updated *Cost* and *Source* fields. Interestingly, the propagation of ProfileAdv messages enables a node to determine if it lies, for some attribute-value pair, on a path where costs are increasing. This occurs when a ProfileAdv is overheard, through passive listening, with a cost greater than the corresponding pivot entry in the SSD. In this case, the identifier of the broadcasting node is inserted in the *IncPaths* field of the pivot entry.

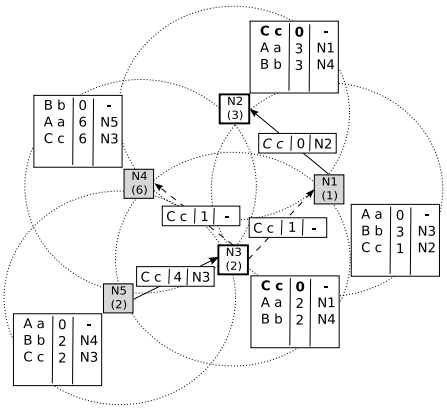
Thus far, we assumed that ProfileAdv messages contain the whole node profile. Nevertheless, if some dynamic attribute changes frequently,



(a) Building the state space (time goes from left to right). Arrow labels denote sending of ProfileAdv messages, showing only the attribute-value (e.g., `A a`), and Cost fields. SSDs are shown only with attribute-value, Cost and DecPath fields. After N1 disseminated its profile, N5's ProfileAdv need *not* be propagated system-wide, but only where updates in SSDs are needed to make its presence known.



(b) After all the nodes performed at least one profile advertisement, the SSDs contain the costs to reach the closest node with a given attribute-value pair.



(c) A message navigating the state space: dashed lines represent exploring directions, solid lines denote decreasing paths. Arrow labels represent application messages showing only the unreserved credits and the intended recipient.

Figure 7.4.: Building and navigating the state space. (In parenthesis is a node's sending cost.)

there is a trade-off between the network load necessary to refresh the advertisements and the accuracy of the information being propagated. A straightforward alternative approach is to disseminate only part of the profile (e.g., static attributes) and perform additional matching at the receiver. These trade-offs are ultimately solved based on the characteristics of the deployment scenario, e.g., by considering information about the size of the logical neighborhood or the network density.

Finally, note how, as shown in Figure 7.4(a), profile advertisements do *not* flood the entire network, as a ProfileAdv is rebroadcast only upon an SSD update. Flooding occurs only for the first advertisement, or more generally when only one node contains a given attribute-value pair—a rather unusual case in the scenarios we target. Instead, for a given set of attribute-value pairs, the state space generation builds a set of non-overlapping *regions*, each containing a node with the considered information. Within a region, each node knows how to route a message addressed to a neighborhood template that includes attributes matching those of a node, along the routes stored in *DecPath*. Each region can be regarded as a "concavity" defined by costs in SSDs, with the target node at the bottom (cost to reach it is zero) and nodes with increasing costs around it. This is illustrated in Figure 7.4(b), where we show the SSDs after all the nodes performed at least one profile advertisement. Next we describe how this distributed state space is exploited for routing.

7.2.2. Finding the Members of a Logical Neighborhood

Local search procedures proceed step by step with subsequent *moves* exploring the state space [166]. At each step, a set of further local moves is available to proceed in the search process. Among them, some moves are accepted and generate further moves, while the remaining ones are discarded. In general, accepting moves depend on the heuristics one decides to employ given the particular problem tackled. In our case, a *move* is simply the sending of an application message containing the neighborhood template. Upon receiving a message, the move is accepted and further send operations are performed if the maximum number of hops, if any, has not been reached (as per the `hops` construct), and either i) the move proceeds along a decreasing path, or ii) enough unreserved credits are available on an exploring path. The notions of *decreasing path*, *exploring path* and *credit reservation* are at the core of our routing solution and are described next.

Decreasing paths. A path is *decreasing* if it brings the message closer to a node whose profile matches the neighborhood template. To do so, message proceeds towards minima of the state space by traversing nodes that report an always smaller cost to reach a potential neighborhood member.

To determine decreasing paths, a node must identify the projection of the state space determined by a neighborhood template. To this end, the node finds in the local SSD the entry matching the neighborhood template with the greatest cost, if any. This entry is called *pivot*. If a pivot exists and is associated, via the SSD *Links* field, to a set of other entries matching the neighborhood template, the cost associated to the pivot represents the number of credits needed to reach the closest matching node via the path indicated by the *DecPath* field. For instance, imagine the application issues a `send(m,n)` operation through our communication API to send the application message `m` to the neighborhood `n`, and assume `n` is defined to address all acoustic sensors. This neighborhood has its *pivot* in entry 2 of the SSD in Figure 7.3, and its predicate (`Function` = *sensor* **and** `Type` = *acoustic*) is matched via the link pointing from entry 2 to entry 1. Consequently, the node evaluates the cost to reach the closest acoustic sensor as 5 and forwards the message towards N37.

Due to the state space generation process, messages following a decreasing path are certainly forwarded towards nodes matching the neighborhood template. Indeed, these paths simply follow the reverse paths previously setup by ProfileAdv messages originating from nodes whose profile contains information matching the neighborhood template. Additionally, note how the reply feature provided by our communication API can be implemented trivially by keeping track of the reverse path along which a message is received.

Exploring paths. If a message were to follow decreasing paths only, it would easily get trapped into local minima of the state space. To avoid this, we allow messages to be propagated also along *exploring paths*, i.e., directions where the cost to reach the closest node with a particular attribute-value pair is non-decreasing. Exploring paths include directions where the cost does not change (e.g., at the border between two regions) or where it increases. The latter are stored in the *IncPaths* SSD field, as discussed in Section 7.2.1.

Activating multiple exploring paths at each hop is ineffective, as it is likely to generate many routes that are shortly after rejoined. Therefore, exploration proceeds along a single increasing path, if available. Explo-

ration on multiple paths, achieved through physical broadcast, is activated only when the message reaches a neighborhood member (i.e., a minima of the state space), or after the message has travelled for E hops, with E being a tunable protocol parameter. This design choice stems from the observation that increasing paths are key in enabling the message to "escape" local minima by directing it towards the boundary where a region confines with a different one, and a different decreasing path may become available.

Credit reservation. The instantiation of a neighborhood template may specify the credits to be spent for communicating with neighborhood members, as discussed in Section 6.2.2. To support this feature, the number of credits is appended by the sender to every application message sent to a given neighborhood. A node may decide to split these credits in two: one part *reserved* to be spent along decreasing paths and the other along exploring ones. The splitting occurs at the first node that identifies a decreasing path for the message being routed, and is effected by removing the reserved credits from the amount in the message, therefore effectively reserving the credits along the entire decreasing path. For instance, Figure 7.4(c) shows a message sent by N5 with 6 credits, targeting a neighborhood defined by a single predicate $\texttt{C} = c$. Neighborhood members are shown in white. As the pivot in N5's SSD reports a cost of 2 to reach the node N3 matching the predicate, the message is forwarded to N3 with 4 unreserved credits.

To deal with credit reservation, a node checks whether its identifier is inserted in the message by the sender node as the next hop along a decreasing path towards a matching node. If so, the node simply forwards the message to the next hop on the decreasing path (found in its SSD) without modifying the credit field, since the necessary credits have already been reserved by the first node on the decreasing path. Otherwise, if exploring paths are to be followed, the node "charges" the message for the number of credits associated to the node sending cost, as per the `use cost` declaration. The remaining (unreserved) credits are assigned to the exploring paths the local node decides to proceed on. Normally, all these credits are assigned to the single message forwarded along the increasing path. However, if multiple paths are explored in parallel through broadcast according to the heuristics described above, the unreserved credits are divided by the number of neighbors before broadcasting the message. In Figure 7.4(c), N3 receives a message with 4 remaining credits. Since it is a neighborhood member, the message must be broadcast along all the available exploring paths. Therefore, N3 charges the message for its own sending cost (2) and divides the

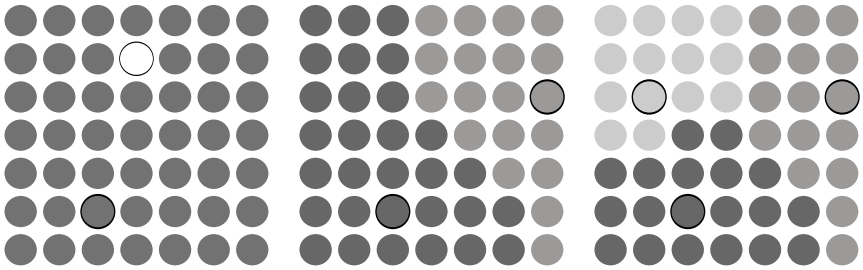


Figure 7.5.: State space generation. The first ProfileAdv message spreads throughout the system as no node disseminated its profile yet. Profiles advertised by other nodes propagate only until a smaller cost is encountered, partitioning space in regions centered on neighborhood members. The white node does not receive the message in the first propagation—due to collisions—but eventually receives it in later retransmissions.

remaining credits by the number of its physical neighbors. This results in activating two exploring directions, each with a 1-credit budget.

7.3. Evaluation

This section reports about an evaluation of our routing protocol for Logical Neighborhoods. To this end, we implemented it on top of TinyOS [42] and evaluated it using the TOSSIM [124] simulator. Our goals were to verify that the protocol behaved as expected for what concerns the generation of the state space and the cost-aware routing of messages, and to characterize its performance. This is key to assess the feasibility of our approach and abstraction. The deployment scenario we simulated is a grid where each node can communicate with its four neighbors. This choice not only simplifies the interpretation of results by removing the bias induced by more unstructured scenarios, but also models well some of the settings we target, e.g., indoor WSN deployments [167].

7.3.1. Analyzing the Routing Behavior

Before characterizing the performance of our routing protocol, we analyze whether its behavior matches our design criteria. First, we verify separately the two basic mechanisms underlying our routing, i.e., the state space generation and its "navigation" by application messages addressed to a logical

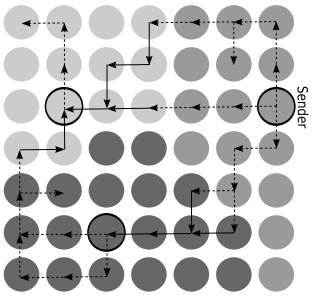


Figure 7.6.: An application message navigates the state space. Solid lines are decreasing paths, dashed lines are exploring paths.

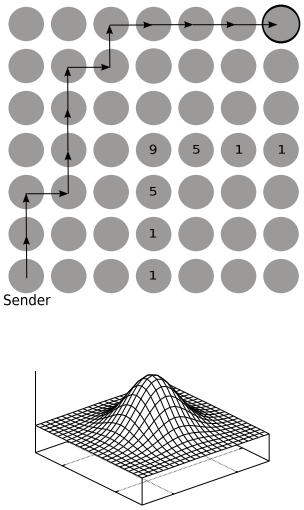


Figure 7.7.: A message navigating a state space where sending costs follow the distribution at the bottom.

neighborhood. As for the former, the key property we want to verify is that the propagation of ProfileAdv messages is localized and partitions the system in non-overlapping regions, each with routing information towards a neighborhood member.

To simplify the analysis of results we developed a simple visualization tool that, given a simulation log and a neighborhood template, displays the propagation of ProfileAdv as well as applicative messages. Figure 7.5 shows a sample output of our tool where the logical neighborhood

we consider selects three nodes (represented as circled nodes), and the node sending cost is equal for all devices. The three snapshots correspond to the points in time when a given ProfileAdv, generated by one of the neighborhood members, has ceased to propagate. As it can be observed, the first ProfileAdv is propagated in the whole system, as no other profile information exists yet. However, when the second member propagates its profile, this is spread only until it reaches a node where the cost is less than the one in ProfileAdv. This process partitions the state space in two non-overlapping regions. Eventually, the system reaches a stable situation where the number of regions is equal to the number of neighborhood members, as shown in Figure 7.5—right.

For what concerns routing of applicative messages, Figure 7.6 shows the output of our visualization tool when a message is sent to the same neighborhood of Figure 7.5. The credits associated to the neighborhood are set as an over-approximation of the credits needed to reach the same three nodes along the shortest path. (More details about setting credits are reported later.) Note how the situation in the picture is a worst-case scenario where the sender belongs to the same neighborhood the message is addressed to. In this case, the message starts from a minimum of the state space, i.e., without any decreasing path. Therefore, the initial moves must be exploring ones, until a region different from the one where the message originated is reached. Despite this unfavorable situation, the message reaches all the intended recipients by alternating moves along decreasing paths with exploration steps.

The effectiveness of our mechanisms in reducing communication costs is unveiled when heterogeneous devices with different sending costs are deployed. Figure 7.7 shows a situation with a single neighborhood member and a message sender placed at the opposite corners of the grid, and where sending costs are assigned according to an integer approximation of a bi-dimensional Gaussian distribution. The figure shows the message dutifully steering away from the network center, where sending costs are higher, and striking a balance between the length of its route and the sending cost of the nodes on it. Thanks to the way our state space is generated through profile advertisements and SSD updates, this path is guaranteed to be, within a region, the one with the minimum cumulative sending cost.

7.3.2. Performance Characterization

Here, we study quantitatively the performance of our protocol. Therefore, we defined a set of synthetic scenarios with a variable number of nodes placed 35 ft apart and with a communication range[1] of 50 ft. Each run lasted 1000 s—a value for which we verified all the measures exhibit a variance less than 1%.

Each node is configured with a single (static) attribute whose value is randomly chosen from a predefined set $\mathcal{A}$ at system start-up. This profile is disseminated by ProfileAdv messages once every 15 s. A single sender node is placed in the center of the grid, generating applicative messages at the rate of 1 msg/s towards a single neighborhood defined with an equality predicate over the node profiles. In this setting, the number of receivers is determined by $|\mathcal{A}|$, and in our case yields a number of neighborhood members of about 10% of the nodes in the system. The node sending cost is constant and identical throughout the system.

Credits are assigned by computing the average cost to reach each node in the system along the shortest path and weighing this value by the probability of the node being a receiver. Then, we increased this minimal value by about one third, to give each message some extra credits to spend on exploratory paths. This approach clearly overestimates the actual cost to reach a receiver, e.g., because it does not consider that two receivers may share part of the path from the sender. The definition of a model supporting fine-tuning of credit assignment to neighborhoods deserves further investigation based on the large body of literature on ad-hoc network density and random graph theory, and is our immediate research goal.

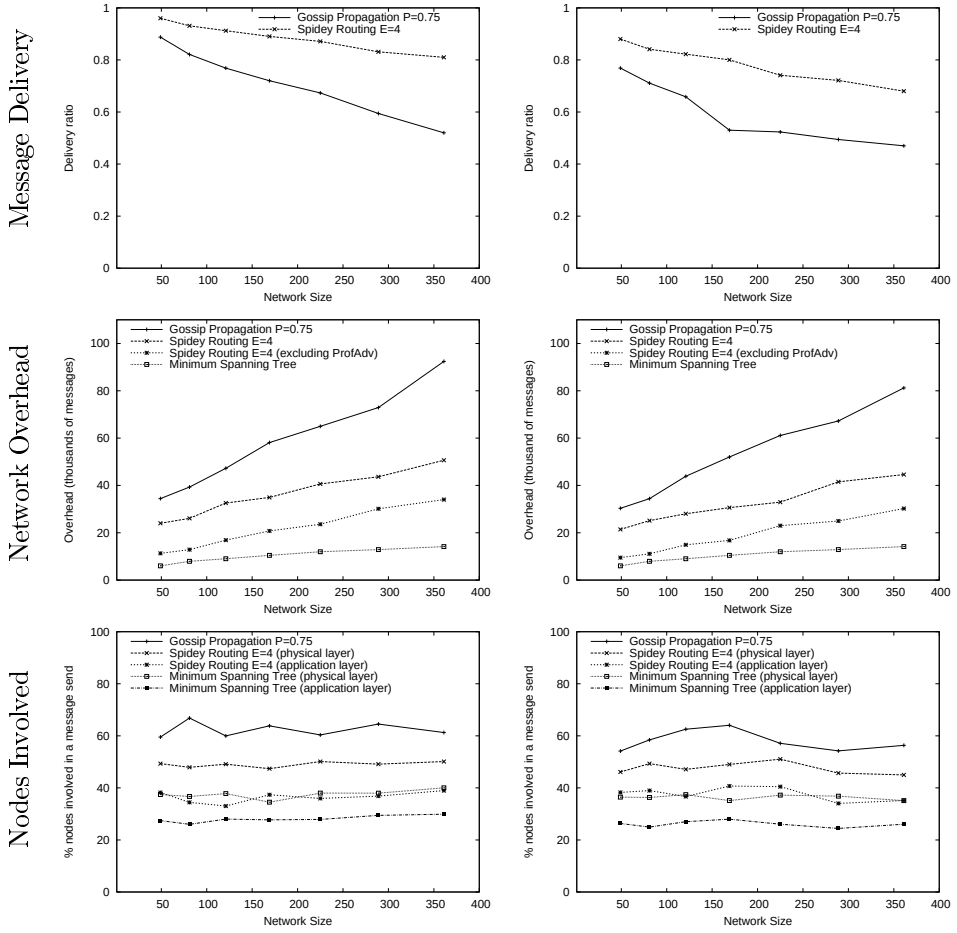
In the absence of directly applicable solutions to compare against, we chose a gossip approach as a baseline, because it is general enough to address the characteristics of our scenarios (e.g., lack of knowledge about the nature of applicative data) and yet generates less traffic than a straightforward flooding protocol. We set the protocol parameters so that gossip rebroadcasts a packet received for the first time with a probability $P = 0.75$, and our solution triggers new exploring directions once every $E = 4$ hops. This latter choice is a reasonable trade-off between generating too many redundant exploratory paths (E too small) and never activating exploratory paths within a region ($E > d$, with d the region diameter).

We based our evaluation on three metrics, namely i) the *message delivery*

[1] We used the TinyOS' `LossyBuilder` to generate topology files with transmission error probabilities taken from real testbeds.

ratio, defined as the ratio between the messages received by neighborhood members and those that have actually been sent; ii) the *network overhead*, defined as the overall number of messages exchanged at the MAC layer, thus including ProfileAdv messages; and iii) the average number of *nodes involved* in routing. This metric is further divided into the nodes processing a message at the MAC layer, and those processing one at the application layer. Message delivery is a measure of how effectively a protocol steers messages towards the intended recipients. On the other hand, in the absence of a precise model to evaluate a node's power consumption, ii) and iii) provide a sense of how a protocol exploits communication and computational resources, respectively.

Figure 7.8 illustrates our simulation results along the aforementioned metrics and w.r.t. the network size. Each chart is the average result of 5 different runs. As it is clear from the figures, our protocol outperforms gossip in all metrics. Message delivery is consistently higher than in gossip, and is even significantly less sensitive to an increase of the network size. As for network overhead, we provide additional insights by showing the results for our protocol with and without ProfileAdv advertisements, and by comparing against the ideal lower bound provided by routing along the minimum spanning tree rooted at the sender and connecting all neighborhood members (computed with a global knowledge of network topology). The chart evidences that we generate almost half of the overhead of gossip and yet deliver significantly more messages. The gap between the two is even more evident in the curve without the ProfileAdv messages, which essentially highlights how efficient is the pure routing mechanism, once the routing information is in place. This is particularly significant because the dissemination of ProfileAdv during state space generation is a fixed cost that is paid once and for all. In other words, adding another sender—regardless of the neighborhood it addresses—does *not* increment the overhead due to state space generation. In addition, the chart shows how the performance of our protocol in this case is closer to the ideal lower bound than to gossip. Finally, for what concerns the nodes involved in processing, Figure 7.8 shows that our performance at MAC layer is in between gossip and the minimum spanning tree, while at the application layer our routing requires only about half of the nodes exploited by gossip to process application messages and exhibits a performance closer to the minimum spanning tree. Therefore, our protocol is likely to provide a considerably longer network lifetime. This result is due to our guided exploration process, which



(a) Static Network (b) Dynamic Network (failures)

Figure 7.8.: Evaluation against gossip and ideal multicast, in static and dynamic scenarios.

privileges unicast messages (that, unlike broadcast, do not reach the application layers at all nodes in range), thus saving processing. In contrast, gossip explores the system in a completely "blind" way.

As shown in the right column of Figure 7.8, the evaluation was carried out also in a more dynamic scenario where 10% of the nodes are randomly turned off for 30 s and then reactivated without allowing any settling time

in between[2]. A similar setting has already been used in existing works on routing for WSNs (e.g. [37]) to simulate node failures or the addition of new nodes. As Figure 7.8 shows, our protocol still provides higher delivery than gossip at lower communication and computational costs, despite node failures. In particular, although nodes joining or leaving the system generate additional profile advertisements to change the shape of the state space, the network overhead remains far from the one of gossip. This result is due to the ability of the state space to change its shape very rapidly in response to network topology changes. For instance, a single ProfileAdv message dissemination among nodes in close proximity to the changing one is usually all it is needed to restore a stable situation.

Finally, the results illustrated in this section should be regarded as worst-case. Indeed, not only the credit assignment can likely be fine-tuned to waste less resources, but also our choice of neighborhood predicates (single constraint) is restrictive. Indeed, it forces each message to follow at most a single decreasing path at a time: neighborhood templates containing multiple elementary disjuncts instead can be routed more accurately by exploiting multiple decreasing paths, therefore further increasing delivery. Moreover, setting uniform costs throughout the system does not leverage the ability of our protocol to route in a cost-aware fashion. Nevertheless, we chose these settings to be fair to gossip, which does not provide these capabilities.

7.4. Related Work

Only few works propose distributed abstractions for WSNs that support some notion of scoping. Moreover, unlike the strongly decentralized scenarios we target with Logical Neighborhoods, many assume a single data sink.

The work closer to ours is the neighborhood abstraction described in Hood [21], where each node has access to a local data structure where attributes of interest provided by (physical) neighbors are cached. However, communication is expected to flow only according to a many-to-one paradigm. In addition, the current implementation considers only 1-hop neighbors and is mainly based on broadcasting all attributes and performing filtering on the receiver's side. Our framework is more flexible as it

[2]We excluded from this random selection the intended message recipients, as this would irremediably impact the message delivery ratio.

provides a general application-defined neighborhood abstraction, which is decoupled from the application functionality and therefore can be used for a wide range of purposes (e.g., network reprogramming).

The work on Abstract Regions [14], instead, proposes a model where <*key,value*> pairs are shared among the nodes belonging to a given *region*. The span of a region is based mainly on physical characteristics of the network (e.g., physical or hop-count distance between sensors), and its definition requires a dedicated implementation. Therefore, each region is somehow separated from others. This results in a much lower degree of orthogonality and flexibility with respect to our approach. In Abstract Regions, the concept of *tuning interface* provides access to a region's implementation, enabling the tweaking of low-level parameters (e.g., the number of retransmissions). Differently, our approach provides a higher-level notion of cost that can be used to control resource consumption.

In TinyDB [17], materialization points create views on a subset of the system. In this sense, common to our work is the effort in providing the application programmer with higher-level network abstractions. However, the approach is totally different, as TinyDB forces the programmer to a specific style of interaction (i.e., a data-centric model with SQL-like language) and targets scenarios where a single base station is responsible for coordinating all the application functionality. SpatialViews [168] is a programming language for mobile ad-hoc networks where *virtual networks* can be defined depending on the physical location of a node and the services it provides. Computation is distributed across nodes in a virtual network by migrating code from node to node. Common to our work is the notion of scoping virtual networks provides. However, SpatialViews targets devices much more capable than ours, focuses on migrating computation instead of supporting an enhanced communication facility as we do, and yet provides less general abstractions.

Finally, in [20], the authors propose a language and algorithms supporting generic role assignment in WSNs with an approach that, in a sense, is dual to ours. In fact, their approach *imposes* certain characteristics on nodes in the system so that some specified requirements are met, while in our approach the notion of logical neighborhood *selects* nodes in the system based on their characteristics.

As for our routing protocol, we were influenced by Directed Diffusion [37] in using a soft-state approach based on periodic refresh of routes. However, our solution is radically different as it targets more general scenarios.

We do not assume data collection as the main application functionality, and therefore we cannot rely on any knowledge about message content, required in Directed Diffusion for interpolation along failing paths. Similarly, we take into account an explicit notion of communication cost without relying on an application-defined notion of data rate. Moreover, our profile advertisements do not propagate to the whole network, unlike interests in Directed Diffusion. Finally, routing in Directed Diffusion is entirely determined by gradients, while we make the system more resilient to changes by allowing exploratory steps, whose use is nevertheless under the control of the programmer through the credit mechanism.

Part IV.

Building upon Logical Neighborhoods

8. Fine-Grained Software Reconfiguration in WSNs

In the introductory part of this thesis, we discussed how WSNs are increasingly being employed in scenarios whose requirements cannot be fully predicted, or where the system must adapt to changing conditions. In these scenarios, the ability of injecting new functionality in the system is of paramount importance. In addition, recent WSN proposals often employ heterogeneous *nodes (e.g., sensors and actuators), which require the deployment of different code on different devices, based on their characteristics. Unfortunately, existing work in the field largely focuses on scenarios where the same, monolithic program is distributed to all the nodes.*

Chapter 4 presented component-based programming models to address the first issue above, i.e., identifying what functionality needs to be reconfigured on the single nodes. In this chapter, we leverage off our work on Logical Neighborhoods to address the latter challenge, namely, specifying where *reconfiguration occurs. To achieve this, we develop a customized version of Logical Neighborhoods combined with the* FiGaRo *component model. Notably, this also includes a novel, dedicated routing scheme providing code distribution to given subsets of nodes. We describe the integration between the two solutions and report on the performance of our code distribution mechanism, showing that its communication overhead lies within 9% of the theoretical optimum. The results presented here appeared in [6].*

8.1. Introduction

The nodes of a WSN are often deployed in large numbers and inaccessible places, making individual code uploading an impractical solution. This problem was early recognized in the WSN research field, leading to solutions exploiting the wireless link for on-the-fly software reconfiguration [169].

However, these solutions were designed to suit the needs of early WSN architectures, i.e., application-specific systems with homogeneous devices.

Problem and Motivation. Today, WSNs are proposed in contexts where their functionality changes over time and/or cannot be predicted a priori. For instance, in emergency response [4] systems the WSN must be reconfigured on-the-fly by mobile operators which demand customized behavior to carry out their activities. In similar scenarios, anticipating all expected needs, if at all possible, may lead to complex and unreliable code cluttered with rarely-used functionality. Therefore, software reconfiguration—even if representing a rare activity compared to the application operations—becomes a much-needed feature. For reconfiguration to be fully effective, however, programmers must retain fine-grained control over *what* is being reconfigured, by updating selected functionality to minimize energy consumption. However, most platforms allow updates only of the full application image. In the very few exceptions, programmers sorely miss proper constructs to deal with dependencies among different functionality, versions, and other fundamental aspects of reconfiguration [169].

Moreover, modern WSNs are typically heterogeneous, containing a mixture of sensing devices and/or actuators. In building monitoring, for instance, a wide range of sensor and actuators is deployed, e.g., to implement heating, ventilation, and air conditioning (HVAC) control [27]. As different nodes are likely to run different application code, software reconfiguration may be limited to a specified portion of the WSN. For instance, a structural engineer inspecting a building may want to load a new piece of functionality only on seismic sensors deployed in a specific location (e.g., the floor being inspected), to process the sensed data in a previously unanticipated manner [90]. In this case, fine-grained control over *where* the code is deployed, based on application attributes of the nodes, is largely missing from existing approaches, which instead are designed to distribute the *same* code to *all* the nodes, regardless of their function [169].

Contribution. We already tackled the former issue above in Chapter 4, using component-based programming solutions. Specifically, we illustrated how our FIGARO component model defines flexible constructs for structuring the code on the single nodes. Moreover, differently from other component models for WSNs (e.g., [28]), FIGARO provides dedicated constructs to deal with component dependencies and versions, and to simplify the reconfiguration process.

By leveraging off the work on Logical Neighborhoods, described in the

```
DECLARE_NODE({
  Function = SENSOR
  Type = TEMPERATURE
  Floor = 1
  Battery = getBatteryReading()
})
```

Figure 8.1.: Declaring node attributes.

third part of the thesis, here we describe how we augment the FIGARO programming model with constructs to restrict component dissemination only to a given subset of nodes—the reconfiguration target—based on programmer-specified characteristics of the nodes or their current software configuration. These features are described in Section 8.2. This way, FIGARO tackles the two problems in an integrated way, spanning all the aspects from the programming model to code distribution.

To enable the approach above, we need an efficient code distribution scheme able to deliver the new functionality to a subset of nodes only. These requirements call for a customized approach to routing. Our solution to this issue, illustrated in Section 8.3, is both lightweight and efficient, as demonstrated in the evaluation reported in Section 8.4. Notably, our solution for code distribution results in a communication overhead within 9% of the theoretical optimum, which is instead computed in a centralized manner and with global knowledge of the system topology.

Finally, in Section 8.5 we compare the code distribution scheme of FIGARO against the current state of the art.

8.2. Distribution Model and Tool Support

Borrowing from our earlier work on Logical Neighborhoods, here we describe how we empower FIGARO with the ability to delimit the portion of the WSN where reconfiguration takes place. This is achieved with dedicated programming constructs that enable programmers to: *i)* declare the attributes characterizing a node; *ii)* specify the reconfiguration target—i.e., the subset of nodes for component deployment—using boolean predicates over the nodes' attributes. Our current implementation leverages off the Contiki [56] operating system.

Figure 8.1 shows an example where we use the `DECLARE_NODE` macro to specify that a node hosts a temperature sensor and is located on a

```
DECLARE_TARGET({
  Function == SENSOR && Battery >= 70 &&
  (Type == TEMPERATURE || Type == VIBRATION) &&
  RUNNING(TreeRouting) &&
  VERSION(TreeRouting) <= 11
})
```

Figure 8.2.: Declaring the reconfiguration target.

given floor. Note how, in principle, attributes can be assigned any legal C expression, including C functions as in the case of the `Battery` field. The nodes targeted by the reconfiguration can be specified declaratively as an (arbitrary) boolean predicate over node attributes using the macro `DECLARE_TARGET`. In Figure 8.2, we specify as reconfiguration target the set of temperature or vibration sensors with at least 70% of battery left, and running a `TreeRouting` component with version less than 11. Notably, the latter requirement leverages off information automatically exported by the FiGaRo run-time layer illustrated in Section 4.7, which describe the current component configuration on a node. Specifically, the parametric, built-in predicate `RUNNING` takes as input the name of a given component C, and yields true when evaluated on a node where C is currently in such state. Instead, the built-in function `VERSION` returns the version of the component given as parameter.

Differently from the FiGaRo language constructs described in Chapter 4, however, the constructs concerned with the reconfiguration target require a minimal amount of pre-processing. On the user base-station, reconfiguration is triggered using a dedicated executable, whose arguments are two files: one containing the component binary image and one with the reconfiguration target (e.g., as in Figure 8.2). A dedicated pre-processor we developed parses them together, generates a unique reconfiguration identifier, divides the binary image into smaller chunks fitting in single physical messages, and starts injecting them into the network. The details of the routing protocol determining their propagation are described next.

8.3. Routing Protocol for Selective Code Distribution

Our dedicated distribution scheme revolves around two base mechanisms:

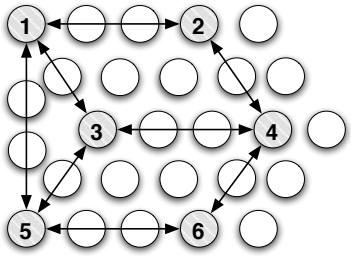


Figure 8.3.: A mesh connecting all target nodes.

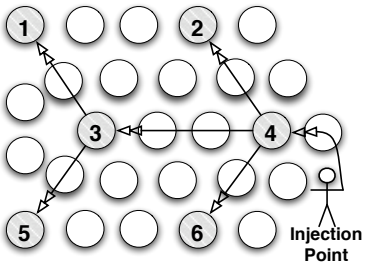


Figure 8.4.: A distribution tree exploiting the mesh.

- While the application is running, we exploit its message traffic to build a *mesh* topology interconnecting all nodes with same attribute-value pairs, as in Figure 8.3, to identify all possible alternative paths connecting the relevant nodes.
- When a reconfiguration is about to occur, a subset of the mesh paths are exploited to build a tree rooted at the target node closest to the injection point, as in Figure 8.4. The tree is then used to propagate the component chunks to all target nodes.

In principle, the two mechanisms above could be designed independently. Nonetheless, our solution is explicitly conceived to take advantage of their mutual interplay. As our objective is to build shortest paths to the target nodes, we make all paths in the mesh bi-directional. This allows us to exploit the same shortest paths regardless of where the code is injected. Moreover, our solution is designed to create a *planar* mesh topology, i.e., one in which no two paths with different end-points cross at any intermediate node, as in Figure 8.3. Results in graph theory indeed demonstrated

Source	Attr	Val	Cost	Bridging	Bridge Cost	Next Hop	Timestamp
Node 3	B	3	0	null	null	self	4
Node 4	A	1	1	Node 1	3	Node 4	25
Node 1	A	1	2	Node 4	3	Node 2	72

Figure 8.5.: Routing table at node 3 in the situation of Figure 8.6(c).

how planar graphs involve fewer routing loops [170]. As a result, the tree topology built atop the mesh easily identifies near-optimal paths, as we demonstrate in Section 8.4.

8.3.1. Building the Mesh Topology

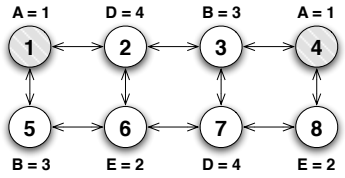
Architecture and Data Structures. As the mesh is built during normal system operation, we must minimize the impact of the mesh-building protocol on the application behavior. We obtain this goal by designing a solution that does *not* generate explicit control messages. Rather, we leverage off the application traffic by piggybacking the current value of a node's attributes on every outgoing message[1]. This is achieved by interposing a thin software layer between the application and the underlying network layers whose interface is the same as the original network stack, making its use transparent to the application.

The information piggybacked is overheard by all nodes in range[2], and used to populate a simple routing routing table (e.g., as in Figure 8.5), that describes the paths of the mesh. Each entry in the table contains a node identifier and the associated attribute-value pair, the next hop to reach that node along with the corresponding cost in hops, and a timestamp to discriminate stale information. In addition, the *Bridging* and *Bridge Cost* fields are used to distinguish entries corresponding to bidirectional paths. The former possibly contains the identifier of another node with same attribute-value pair, representing the opposite end-point of the path itself, whereas the latter stores the total path length in hops. Each entry in the table is associated with a lease (not shown) that, if not refreshed, causes the entry removal.

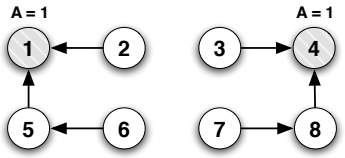
Protocol Operation. Figure 8.6 describes an example of mesh construction. The initial situation, depicted in Figure 8.6(a), illustrates the physical

[1] In case a node is silent, we generate dummy messages at a pre-specified rate.

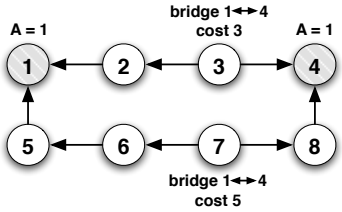
[2] A simple hook within the Contiki radio layers allows us to overhear also unicast messages.



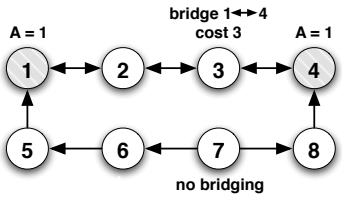
(a) Initial situation. Arrows describe the physical topology.



(b) Node 1, 4, 5 and 8 send application messages. Bold arrows describe the Next Hop field for `A = 1`.



(c) Node 3 and node 7 recognize it is possible to build a bidirectional path connecting node 1 and node 4.



(d) The path through node 3 is complete. The one through node 7 is pruned as unnecessary.

Figure 8.6.: Example of mesh construction (grey circles are target nodes).

network topology and the attributes defined in the node declarations, along with their corresponding values. Initially, all routing tables contain only entries relative to the local node. For instance, let us focus on the nodes having attribute `A` equal to `1` as target. When node 1 first sends an application message, we append a subset of node 1's routing table entries to it[3]. The nodes in range parse this information, increments all cost fields by one, and add these entries to their routing tables provided no other entry with same attribute-value pair but smaller or equal cost exists. By doing this at every node, node 1's specification spreads across multiple hops. For instance, node 5's piggybacked information also includes node 1's initial entry, as it was overheard from node 1's transmissions. Assuming node 4 and 8 eventually send some application message as well, the resulting situation is as depicted in Figure 8.6(b).

[3]Entries are selected in round-robin, their number limited by a configuration parameter.

To recognize when a bidirectional path can be established, we look for received entries containing an attribute already stored in the local table, but from a different source and greater or equal cost. This is the case in Figure 8.6(c), where node 3 receives from node 2 an entry for attribute `A` with value `1` and cost 2. In this situation, a bidirectional path for the same attribute can be established, with node 1 and node 4 as end-points. To establish the path, we insert the newly received entry in node 3's routing table with the Bridging field set to the identifier of the opposite end-point of the path (e.g., node 4 in case of node 3 in the last entry of Figure 8.5), and the Bridge Cost field set to the total cost of the path itself. Similarly, we update any entry already in the table that refers to the other end-point of the bidirectional path—as it is the case for the second entry in Figure 8.5—modifying the Bridging and Bridge Cost accordingly. Afterwards, entries with non-null Bridging fields are propagated only towards the node reported in the Bridging field itself. Thus, the second entry in Figure 8.5 is propagated only towards node 1, whereas the last entry spreads only towards node 4. This is as simple as appending an optional field to all outgoing messages stating what nodes propagate what entries.

As a side-effect of the above processing, more than a single bidirectional path connecting node 1 and node 4 could be established. For instance, a further path is eventually built through node 5, 6, 7 and 8, with a total cost of 5. This, however, poses unnecessary communication overhead. To alleviate this undesirable behavior, non-null Bridging entries are propagated only if the node is not aware of other (bidirectional) paths with smaller cost. In our example, node 7 eventually stops propagating its non-null Bridging entry after overhearing the last entry at node 3, which contains a smaller cost. This ultimately yields the situation in Figure 8.6(d). Although this scheme does not completely prune all redundant paths, it greatly diminishes their number. Pruning all the paths but the shortest one would indeed require propagating the minimum cost entry multiple hops away from the shortest path. How far to propagate is hard to determine without knowledge of the network topology. Also, the additional paths may be used as back-ups in case of sudden faults. We plan to investigate this in the near future.

Dynamic Attributes and Topology Changes. The protocol operation occurs whenever the application generates network traffic. Therefore, in the case of time-varying attributes, the accuracy provided by the mesh topology w.r.t. the current values of attributes is ultimately dictated by the amount of application traffic over time. Applications generating more traffic allow

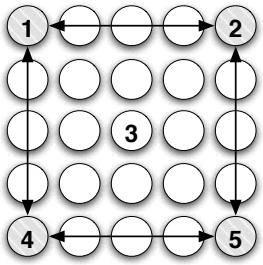


Figure 8.7.: Node 3 has equal cost to all target nodes.

our protocol to build more accurate topologies, whereas it is difficult to do so if the amount of traffic flowing in the network is insufficient to keep up with the dynamics of the varying attribute. As for topology changes, e.g., due to failing nodes, invalid routes will eventually expire without being refreshed. As soon as the application generates further messages, our protocol identifies alternative routes according to the new topology. Still, the time taken to build the new routes depends on the amount of traffic generated by the application. In case the system needs to react quickly to changes, yet the application traffic is insufficient to adapt the routes immediately, our protocol can be forces to generate dummy messages at a high rates.

Enforcing Planarity. By construction, our scheme does not generate multiple paths with different end-points crossing at an intermediate node. Indeed, the only way this can be obtained is to have, in the same routing table, more than one non-null Bridging entry for the same attribute-value pair with different source. Consider Figure 8.7: node 3 may try to establish two crossing paths, e.g., connecting node 1 to node 5 and node 2 to node 4. This cannot occur in our protocol, as received entries with cost greater or equal to the local table for the same attribute-value pair are ignored, end every non-null Bridging entry can be used to establish a single bidirectional path. Therefore, node 3 in Figure 8.7 will never be able to generate crossing paths.

8.3.2. Distributing Code

When a reconfiguration takes place, code is distributed along a tree: redundant paths in the mesh are identified based on the position of the code

injection point, using a *marker* message. This contains the reconfiguration identifier generated by our pre-processor, and an encoding of the predicate defining the required reconfiguration target. The former serves to support multiple concurrent reconfigurations. The latter is used by nodes to determine, based on their routing table, the next hop for the marker. Upon forwarding, target nodes add to the marker the cost accumulated along the last bidirectional path traversed. This way, the marker eventually reaches all the target nodes, making them aware of their distance from the injection point. This information is used at each target node to configure a dedicated distribution tree by selecting as parent the target node that, along the links of the mesh, is the closest to the injection point. The selection is communicated to the parent with a message containing the identifiers of the source target node and of the selected parent. Note how code dissemination can start before the entire tree is built. When receiving a code chunk, a node that has not yet determined its children simply defers forwarding and buffers the chunk. Buffering would happen in any case, since a component cannot be reconstructed until all chunks are received.

The code distribution phase demands reliable communication, e.g., because all code chunks must be correctly delivered. We employ a simple hop-by-hop reliability mechanism, based on implicit acks. Nodes on a tree path buffer every message, waiting for the downstream node to re-send it. When this occurs, the upstream node overhears the transmission, and concludes the message was received; otherwise, it is re-sent. Similar techniques have already been successfully employed in WSNs [171]. However, our implementation decouples this aspect, enabling the use of alternative reliability schemes.

8.4. Evaluation

In this section we assess the effectiveness of our solution for code distribution by reporting about simulations performed using Cooja, the Contiki simulator.

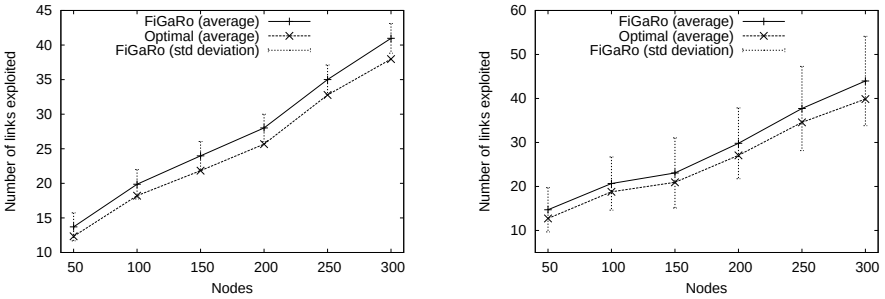
The evaluation of code distribution protocols for WSNs has hitherto focused on metrics such as latency and message overhead [169]. However, these are usually affected by mechanisms other than the distribution protocol itself. For instance, latency is affected also by the MAC layer protocol, as back-off timers, random transmission delays, and transmission slots in TDMA schemes are employed to reduce collisions. Similarly, message

overhead is affected by the specific reliability mechanism employed.

However, the above concerns are orthogonal w.r.t. the problem we are tackling and the *essence* of the solution we presented, whose performance is determined *primarily* by the shape of the tree used during the distribution phase. Indeed, the number of hops separating the injection point from the target nodes strongly impacts both latency and message overhead irrespective of the MAC layer and reliability mechanism employed, which instead affect the individual 1-hop transmissions. Therefore, we choose to evaluate our protocol by focusing on the *number of links* employed during the code distribution phase[4], and compare this metric against the optimal distribution tree computed with a shortest path algorithm and global knowledge of the network topology. We also measured the *convergence speed* of our mesh-building algorithm, i.e., how many messages the application must generate for the routing tables to stabilize. In both cases, we rely on the standard Contiki MAC layer as implemented in Cooja. Moreover, we used the reliability mechanism discussed in Section 8.3, for which simulations confirm a 100% delivery in all the experiments discussed next.

As for simulation settings, each node exports a single attribute whose value is randomly selected at start-up. Reconfiguration targets are defined by a single equality predicate on this attribute. During the mesh-building phase each node sends an application message every $5 + \mathcal{D}$ seconds. $\mathcal{D}$ is a random delay we introduced to avoid locking effects among nodes, and to generate executions with varying traffic rates at different nodes. Application messages are 64 bytes in size, to which we piggyback 24 bytes of control information corresponding to 4 entries from the local routing table. During the simulations, the mesh-building phase takes place first. The convergence speed is determined when routing tables at all nodes do not change for 5 consecutive message sends. At this point, the mesh is considered stable: a random node is chosen as injection point and the tree-building phase is started. We discuss results obtained in regular grids and random topologies. In the former, each node can communicate with 4 neighbors. This setting models some of the applications we target (e.g., indoor WSN deployments [167]). In the latter the number of neighbors varies from 3 to 7. For each scenario, we averaged the results over 20 repetitions with varying distribution scopes and injection points.

[4] In cases where nodes can forward a message towards n neighbors with a single physical packet we still count n links, as most reliability mechanisms would send separate messages anyway.



(a) Grid topologies. (b) Random topologies.

Figure 8.8.: FiGaRo performance vs. topology and system size (target nodes are 10% of the total).

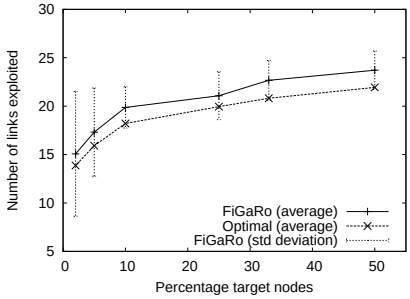


Figure 8.9.: FiGaRo performance vs. number of target nodes (100 nodes arranged in a grid).

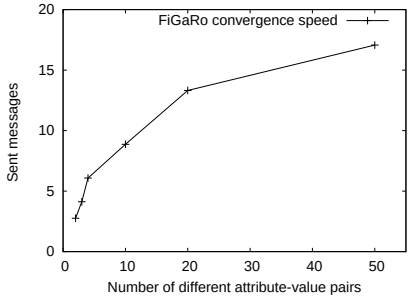


Figure 8.10.: FiGaRo convergence speed (100 nodes arranged in a grid).

Results. Figure 8.8 shows how the number of links exploited by our solution varies according to the system size and topology. Remarkably, the performance of our protocol remains always within 9% of the theoretical optimum, and is almost constant as the number of nodes increases. By examining the simulation logs, we realized that the gap is mostly due to cases where it may be more convenient to access the mesh from more than a single entry point. When this does not hold and the injection point is very close to a target node (i.e., within 2-3 hops), the average gap w.r.t. the optimal solution is even lower, around 3%. This confirms that our mesh-building algorithm, thanks to its planarity property, yields near-optimal routes in the distribution trees built atop. Further, note how Figures 8.8(a) and 8.8(b) exhibit similar trends, although the results on random networks show higher variability due to the irregularity of the topology.

Figure 8.9 provides a different perspective by analyzing the behavior of our protocol w.r.t. the percentage of target nodes. As shown in the chart, our solution is barely affected by this parameter. The high variability observed with few target nodes is due to cases where nodes end up aligned w.r.t. the injection point, and the distribution tree degenerates in a chain. In these configurations, intermediate nodes are reached at essentially no cost. The probability of these configurations decreases as the number of target nodes grows. We limited our experiments to half of the nodes in the system as targets. Beyond this point, the scenario starts bearing similarities with traditional code distribution in homogeneous networks, where all nodes are target. In this case, existing solutions are better suited, e.g., [45].

Finally, we verified that the convergence speed of the mesh-building phase is not affected by the system scale. Indeed, the extent to which routing entries are propagated is not dictated by the overall number of nodes, rather by the amount of redundancy among attribute-value pairs. This claim is supported by Figure 8.10, showing the number of messages required to build the mesh against the number of (distinct) attribute-value pairs in the system. When the latter is small the mesh builds quickly, as the bidirectional paths connecting nodes with the same attribute-value pairs are likely to be short. Instead, when attribute-value pairs are highly heterogeneous the mesh takes more time, due to the dual argument. Overall, the values in the chart are good: only 17 messages need to be sent when 50 different attribute-value pairs are present, i.e., only 2 nodes in the 100-node network of Figure 8.10 have the same attribute-value pair—a rather unusual setting. In any case, the values in the chart should represent only

a very little fraction of the overall system lifetime, typically measured over months or even years.

8.5. Related Work

To the best of our knowledge, we are the first to provide efficient distribution of code to an arbitrary subset of nodes identified by programmer-provided information. In doing so, we leveraged off our work on Logical Neighborhoods [10], described in the second part of this thesis. Nonetheless, tackling the issues germane to code distribution required a completely different routing support, as described in Section 8.3.

In the field of code distribution, the approach closest to ours is the TinyCubus framework [172], where code can be distributed to all nodes with a given role, e.g., all cluster-heads. This is far less flexible than predicate logic over programmer-defined attributes, and does not encompass the ability to identify the target nodes based on their current software configuration, e.g., as provided by the `RUNNING` built-in-predicate. At the network level, TinyCubus assumes *a priori* knowledge of the system topology and of the location of nodes with a given role, as it requires to specify an upper bound on the number of hops separating nodes with the same role. In contrast, our solution is fully dynamic and decentralized.

Network-wide distribution of code has been widely investigated, tackling different facets of the problem. On one hand, solutions have been proposed to reduce the size of the code to be distributed by employing differential patching and smart linking mechanisms, e.g., [43, 44]. Still, similar concerns are orthogonal to the problem we tackle in this work, and the corresponding solutions may be integrated in our framework for even better performance, e.g., by injecting a patch instead of the whole binary when the new component is going to replace an older version. Instead, other approaches focused on routing. Trickle [45] uses a counter-based technique called "polite gossip", whose objective is to suppress redundant transmissions while guaranteeing eventual delivery. Deluge [46] uses a similar technique, with the addition of a negotiation phase to guarantee the proper sequencing of packets. This is also used in MNP [47] to address the hidden terminal problem before transmitting the actual code. Sprinkler [48] and Firecracker [49] instead leverage off node hierarchies, by first sending code to "core" nodes up in the hierarchy, which then forward the code to nodes in their vicinity. As the objective of all the above solutions

is to distributed code to all nodes, they can avoid any background activity under normal operating conditions. For the same reason, however, these mechanisms are hardly applicable in our case. For instance, it would be fairly inefficient to add multi-hop negotiation in Deluge to address the case where the target nodes are multiple hops away.

9. The Virtual Node Abstraction

The Logical Neighborhoods abstraction, described in the third part of this thesis, provides a basic building block to identify arbitrary subsets of nodes, and interact with them. By redefining the traditional broadcast-based communication API, Logical Neighborhoods enable building a range of higher-level language constructs atop, thus providing a stepping stone for the development of flexible programming abstractions.

In this chapter, we explore this possibility by defining virtual nodes*, a programming abstraction that greatly simplifies the development of WSN applications. Virtual nodes are built as a natural extension of the Logical Neighborhoods concept, by abstracting programmer-specified subsets of nodes (e.g., all temperature sensors in a room) into a single, logical one. Spanning both ends of the control loop, virtual nodes take the form of virtual sensors or virtual actuators. The former abstract the data sensed by real sensors into the reading of single, fictitious node; whereas the latter provide a single handle to control a distributed set of actuators. Virtual nodes drastically increase the programmer productivity and foster a higher-quality design, as illustrated in our quantitative evaluation. Moreover, by virtue of our dedicated distributed run-time support, they also yield improved system performance—e.g., an 80% increase in lifetime w.r.t. traditional solutions. A preliminary version of the work described here appeared in [11].*

9.1. Introduction

As we illustrated in Chapter 1, in sense-and-react applications decentralized architectures are the prominent design choice to ensure efficient and timely operations [2]. As a result, the complexity of application development increases dramatically, and ease of programming becomes fundamental.

Scenario. Consider a *building automation* [173] scenario. Sensors are deployed to monitor various phenomena, (e.g., temperature, humidity, vi-

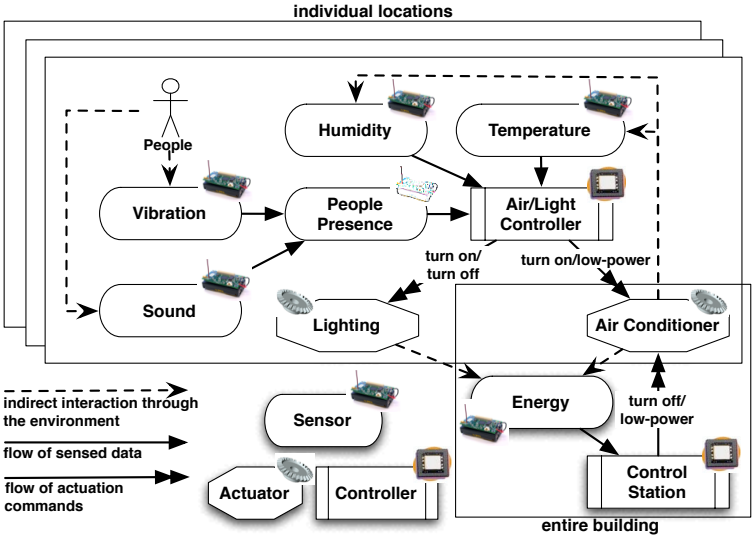


Figure 9.1.: Interactions in building automation.

bration, sound) and the energy consumption of air conditioners and lights. The system controls the activation of these devices to minimize the overall energy consumption. As depicted in Figure 9.1, programmers are to express complex interactions among different subset of nodes, e.g., vibration sensors and light controllers in individual locations such as private offices, as well as energy sensors and air conditioners throughout the whole building. Typical requirements are [173]:

R1: when people is detected from (averaged) vibration and sound readings in individual locations, lights must be activated and air conditioners set for predefined temperature and humidity values.

R2: in the absence of people, lights must be turned off and air conditioners switched to a low-power mode.

R3: when the overall energy consumption exceeds a safety threshold, a central control station must deactivate the air conditioners to avoid blackouts.

State of the art & motivation. Based on the taxonomy described in Chapter 2, WSN programming has hitherto leveraged off solutions providing communication in the *physical neighborhood*, e.g., nesC [28], or based

on *system-centric* computation, e.g., TinyDB [17]. Using the former, programmers reason at a very low-level of abstraction, where the application processing is described in terms of pair-wise interactions between nodes within radio range. In contrast, in the latter developers program the system as a whole, regardless of the individual nodes. The interactions involved in our reference application, however, make these approaches ill-suited to the requirements at stake. Several distinctive traits can indeed be noted:

- The use of *heterogeneous nodes*: different types of sensors are deployed, and multiple actuators are installed to influence the environment differently. This prevents the use of approaches providing system-centric computation, as they usually assume a homogeneous scenario.
- The focus is on programmer-specified *subsets* of nodes, not on the whole network: adjusting the lighting in a corridor involves only actuator nodes in a specific location. However, traditional WSNs programming frameworks lack the appropriate abstractions to partition the system based on application requirements.
- A high degree of *decentralization*: processing must be kept close to where the actuation is to be performed, to minimize latency and save on resource consumption [2]. Physical neighborhood communication, however, only provides the ability to exchange messages between nodes within radio range, and every additional functionality must be coded explicitly.

Contribution. In this chapter we propose *virtual nodes*: a programming abstraction that strikes a balance between the two extremes above. As illustrated in Figure 9.2, virtual nodes abstract application-defined subsets of nodes into a single, logical entity. This applies at both ends of the control loop, i.e., while sensing from the environment, as well as in performing actuation. As for the former, *virtual sensors* process the data sensed by a subset of real nodes through programmer-provided functions, and offer the corresponding output as the reading of a logical node. Dually, *virtual actuators* provide a single entry point to control a group of geographically sparse actuators. The set of nodes to be abstracted into a virtual one is specified with Logical Neighborhoods, using an extension of the SPIDEY language we described in Chapter 6. The salient characteristics of the virtual node abstraction are illustrated in Section 9.2.

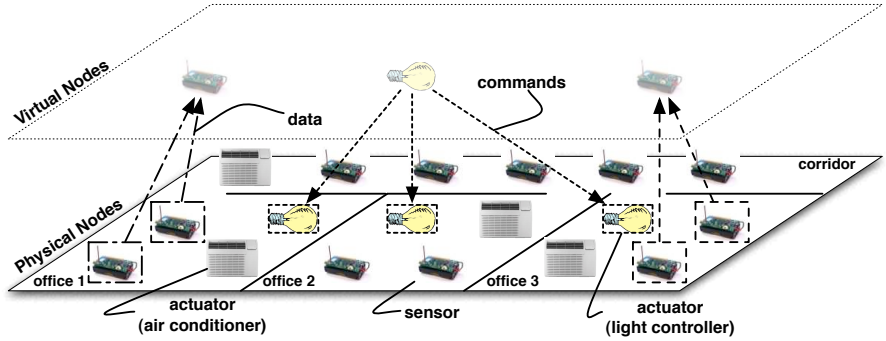


Figure 9.2.: Virtual sensor and actuators. Dashed lines show the real nodes associated to a virtual one.

Our current implementation, described in Section 9.3, revolves around a dedicated *preprocessor* for the SPIDEY language, coupled with a flexible *run-time* support that our preprocessor automatically customizes. For instance, some of the functions describing the mapping to a virtual sensor can be computed *in-network*, i.e., by letting intermediate nodes between sources and receivers evaluate and transmit partial results instead of raw data. This reduces the amount of transmitted data, and yields a better distribution of the processing load. When this technique can be applied, our preprocessor properly customizes the run-time support.

As illustrated in Section 9.4 by comparing our approach against traditional solutions, virtual nodes boost programming performance and design quality, while improving the system performance. Indeed:

- Virtual nodes give developers an intuitive way to deal with subsets of nodes and their interactions. Hence, it is easier to decompose and decouple application concerns into smaller and simpler ones, each affecting different parts of the system. This ultimately yields simpler, cleaner, and more reusable implementations.

- The SPIDEY preprocessor provides a tight coupling between the programming abstraction and the distributed protocols in our customizable run-time, enabling remarkable performance improvements. Indeed, using cycle-accurate emulation of WSN nodes we demonstrate an increase of 80% in system lifetime.

We conclude the chapter in Section 9.5 with a brief survey of related work in the field.

```
node template Sensor
  static Function
  static Type
  static Location
  dynamic Reading

create node vibration from Sensor
  Function as "sensor"
  Type as "vibration"
  Location as "office1"
  Reading as getVibrationReading()
```

(a) Sensor node.

```
node template Actuator
  static Function
  static Type
  static Location
  operation Activate(int tuning)
  operation Deactivate()

create node lighting from Actuator
  Function as "actuator"
  Type as "light"
  Location as "office1"
  Activate(int tuning) as setLightIntensity(tuning)
  Deactivate() as setLightIntensity(0)
```

(b) Actuator node.

Figure 9.3.: Node definition and instantiation.

9.2. Programming WSNs with Virtual Nodes

In this section we illustrate the core concepts and language constructs of virtual nodes, while Section 9.3 describes the corresponding language and run-time support.

9.2.1. Focusing on Relevant Nodes

In our reference application, R1 and R2 require a node controlling the lighting in a private office to average nearby vibration and sound readings, and combine them to infer the presence of people. The ability to focus on a subset of relevant nodes is key to achieve this functionality, and is readily provided by Logical Neighborhoods.

Therefore, before specifying the relevant node subsets, the programmers

```
neighborhood template VibrationSensors(loc)
  with Function = "sensor" and
       Type = "vibration" and
       Location = loc

create neighborhood vb_my_location
  from VibrationSensors(loc: myLocation())
```

Figure 9.4.: Neighborhood definition and instantiation on a node controlling the lighting. (`myLocation()` returns where in the building the node is deployed.)

of our reference application specify the characteristics of the nodes, based on which selection occurs. Figure 9.3(a) shows a fragment of SPIDEY code describing a sensor node, using the language constructs illustrated in Chapter 6. Similarly, Figure 9.3(b) illustrates the definition of an actuator node. This time, however, the latter template also contains *operations*, in our case to activate and deactivate the actuator. Note how the instantiation in Figure 9.3(b) binds them to the same function, i.e., the one used to regulate the light intensity. The desired subset of nodes is then identified using boolean predicates on the node characteristics. Figure 9.4 illustrates a logical neighborhood including all vibration sensors in a location. This can be used, for instance, on a node controlling the lighting to identify the relevant input sensors.

The SPIDEY constructs shown so far identify subsets of nodes without expressing any interaction among them. In a sense, we are still playing on the bottom plane of Figure 9.2, by "drawing" the dashed lines denoting the relevant nodes. Next, we describe how virtual nodes build upon Logical Neighborhoods to provide expressive abstractions that greatly simplify the programming chore.

9.2.2. Virtual Sensors

If our reference application were to be developed using a programming model such as nesC [28], the processing to average the vibration readings in a location would need to be coded in all its gory details. A large fraction of these include communication, and therefore serializing outgoing data, parsing and buffering received packets, storing partial results, and dealing with concurrency issues due to multiple activities occurring simultaneously. The complexity increases further if in-network processing is employed to

```
create node virtual_avg_vibration from Sensor
  Function as "virtualSensor"
  Type as "avgVibration"
  Location as myLocation()
  Reading as average input vb_my_location every 30
             summary algebraic

PartialAvg average(PartialAvg pa, Node n) {
  pa.sum += n.Reading;
  pa.avg = pa.sum / ++pa.count;
  return pa;
}
```

Figure 9.5.: Definition of a virtual vibration sensor on a node controlling the lighting.

reduce resource consumption. Indeed, this entails processing on *all* nodes, not only where the final measure is required. The programmer is thus forced to delve into application-level routing, understand when and how intermediate nodes forward the data, intercept the corresponding processing—and therefore progressively losing touch with the very application logic.

Instead, virtual nodes enable the programmer to achieve the same functionality above in a few lines of code. Notably, as shown in Figure 9.5, the virtual sensor is instantiated using the same `Sensor` template of Figure 9.3(a). As such, it is perceived by the application exactly *as if* it were a local sensing device. The relevant data, however, comes from a distributed data source, as depicted in Figure 9.2. The mapping from the physical readings to the virtual node is expressed during instantiation. The value of `Reading`, formerly provided by the underlying hardware, is now derived by applying the function `average()` to data sensed by nodes in the logical neighborhood `vb_my_location` defined in Figure 9.4. The `every` clause determines the rate at which the virtual sensor reading is updated.

Besides these basic features, the extended SPIDEY also provides constructs enabling language-driven in-network processing. This is accomplished by defining: *i)* a uniform way to express the functions employed, and *ii)* constructs to characterize their mathematical properties.

Functions are expressed in a way similar to [130], namely:

$$\langle state' \rangle = f(\langle state \rangle, \langle input \rangle)$$

where $\langle state \rangle$ is a partial state record computed over one or more sensor values, and $\langle input \rangle$ is a set of input values. For instance, averaging a set

Function	Output	Aggregation
`average()`	Summary	Algebraic
`max()`	Exemplary	Distributive
`sum()`	Summary	Distributive
`median()`	Exemplary	Holistic

Figure 9.6.: Classification of some example functions.

of values can be expressed as:

$$average(\langle avg, sum, count\rangle, \langle value\rangle) ::= \langle \frac{sum+value}{count+1}, sum+value, count+1\rangle$$

which straightforwardly translates into the code for `average()` in Figure 9.5. Note that complex functions can be expressed this way, e.g., isobar-finding algorithms [130].

Instead, to characterize the properties of the functions employed we take inspiration from Gray et al. [174] and consider two dimensions, shown in Figure 9.6. One is concerned with the nature of the output w.r.t. the inputs. The output of an *exemplary* function contains one or more representative values from the input set. Instead, a *summary* function computes some property over all values, not necessarily corresponding to any value in the input set. The other dimension is concerned with the nature of the aggregation performed. The output of a *distributive* function can be obtained by applying the same function on disjoint subsets of input values individually. This does not hold for *algebraic* functions, although the partial states are still of constant size. Finally, in *holistic* functions the size of the partial states grows with the size of the subset of the input values involved.

The SPIDEY preprocessor uses the information above to decide *if* and *how* in-network processing can be applied, and to customize our run-time layer accordingly. For instance, in-network processing can be applied for summary, algebraic functions like `average()`. Conversely, the input data of holistic functions must be reported to a central node where processing takes place in a single step. Details about how this customization is achieved are in Section 9.3.

9.2.3. Virtual Actuators

In our reference scenario, R3 requires the control station to deactivate the air conditioners when the overall energy consumption exceeds a safety

```
neighborhood template AirConditioners()
  with Function = "actuator" and
       Type = "airConditioner" and
       provides Deactivate() and
       provides Activate()

create neighborhood air_conditioners
  from AirConditioners()

create node virtual_air_conditioner from Actuator
  Function as "virtualActuator"
  Type as "airConditioner"
  Deactivate() as apply Deactivate()
                  over air_conditioners reliably
  Activate() as apply Activate()
                  over air_conditioners reliably
```

Figure 9.7.: A virtual actuator used to deactivate all the air conditioners.

threshold. Monitoring can be accomplished with a virtual sensor, which aggregates the energy consumption sensed throughout the building using a `sum()` function. Instead, performing distributed actuation without dedicated support would require an ad-hoc encoding of commands and parameters, message serialization and parsing, and a dedicated routing mechanism to address the relevant nodes without flooding the entire network.

Using virtual nodes, R3 is met easily by instantiating a virtual actuator whose operations are bound to a neighborhood including all the air conditioners, as in Figure 9.7 where `Activate` and `Deactivate` are executed over the `air_conditioners` neighborhood. In general, different operations of the same virtual actuator may operate on different neighborhoods. Note how the neighborhood template includes, besides conditions on attributes, a built-in predicate named `provides` that yields true when evaluated on a node that exports the operation given as parameter.

The optional `reliably` clause determines whether the run-time layer must guarantee the delivery of messages carrying actuation requests. This is needed in some application scenarios (e.g., firing an alarm in intrusion detection systems), but usually incurs higher message traffic. A best-effort scheme is preferable otherwise. Our run-time support provides both alternatives, as described in Section 9.3.

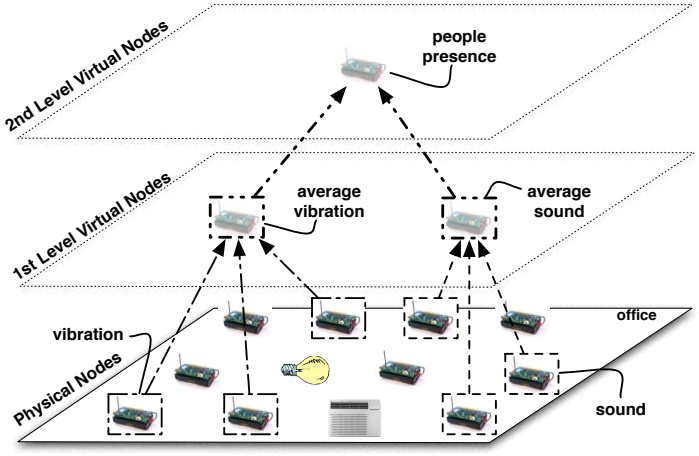


Figure 9.8.: Virtual nodes built upon other virtual nodes.

```
neighborhood template VirtualVSSensors(loc)
  with Function = "virtualSensor" and
       Location = loc and
       (Type = "avgVibration" or Type = "avgSound")

create neighborhood avg_vs_my_location
  from VirtualVSSensors(loc: myLocation())

create node virtual_presence from Sensor
  Function as "virtualSensor"
  Type as "presence"
  Location as myLocation()
  Reading as inferPresence input avg_vs_my_location
             every 120 holistic
```

Figure 9.9.: Definition of a presence sensor from (virtual) vibration and sound sensors.

9.2.4. Virtual Nodes Made of Virtual Nodes

Virtual nodes are perceived by the application just like their concrete counterparts, e.g., by accessing attributes or invoking operations. Therefore, programmers can freely mix concrete and virtual nodes, by including the latter in logical neighborhoods and use them as input (output) to higher-level virtual sensors (actuators). For instance, inferring the presence of

people as required in R1 and R2 can be achieved by creating a virtual presence sensor from the (virtual) vibration and sounds sensors reporting the average readings, leading to the hierarchy in Figure 9.8. As illustrated in Figure 9.9, the programmer is not required to learn new concepts, as the SPIDEY constructs described so far are already sufficient to accomplish the corresponding definition. The only difference is the use of the `Function` and `Type` attributes in the neighborhood template, here used to distinguish the relevant virtual sensors from the physical nodes.

9.3. Virtual Nodes in Practice

Our prototype targets the nesC [28] programming language and the TinyOS operating system [42]. Section 2.5.1 provided a concise introduction to nesC. Here we illustrate how virtual nodes are provided in this language, along with the most relevant characteristics of our run-time support.

9.3.1. Virtual Nodes Language Support

We designed virtual nodes to integrate seamlessly with the nesC programming model, by bringing the duality between commands and events to a distributed setting. nesC commands allow the application to interact with lower-level components. Similarly, virtual actuators allow commands to flow downward towards the nodes controlling the relevant actuators. Therefore, the interface used to access a virtual actuator lists a command for each operation bound to a logical neighborhood. In the case of Figure 9.7, this yields:

```
interface virtual_air_conditioner_if {
  command result_t Deactivate();
  command result_t Activate();
}
```

Hence, distributed actuation is achieved transparently *as if* the actuator were directly attached to the node.

Dually, nesC events make sensed data flow upward from low-level components to the application. Similarly, virtual sensors allow data sensed by real nodes to flow upward towards the defining node. The data are therefore made available to the application as periodic nesC events, as in the following interface for the virtual sensor defined in Figure 9.5:

```
interface virtual_avg_vibration_if {
  event result_t Reading(int value);
}
```

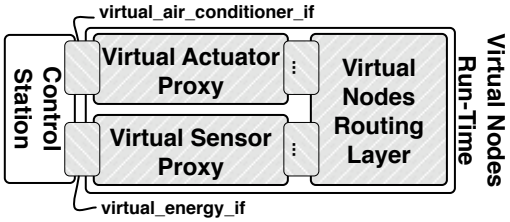


Figure 9.10.: Component configuration on the control station. White components are developed by the programmer, gray ones are automatically generated or belong to our run-time support.

where the `Reading()` event is signaled every 30 seconds, according to the `every` clause we specified.

The processing behind the interfaces is automatically generated by our SPIDEY preprocessor as a set of proxy components, bridging the application and our run-time support. For instance, Figure 9.10 shows the component configuration on the control station node. The `ControlStation` component, *entirely* depicted in Figure 9.11, constitutes the *only* application code written directly by the programmer. This code uses a virtual sensor returning the overall energy consumption in the building (`vs_energy_if`), and a virtual actuator to deactivate the air conditioners when needed (`va_air_conditioners_if`). A handful of nesC lines is sufficient, as most of the complexity is hidden in our customizable run-time, described next.

9.3.2. Run-Time Support

Existing communication protocols for WSNs are only partially suited for implementing virtual nodes. Therefore, we designed a dedicated communication layer in support of our abstraction. This partially leverages off the Logical Neighborhoods routing layer, described in Chapter 7, but it also includes novel mechanisms to address the requirements germane to this particular approach.

Virtual Actuators. Virtual actuators require a *one-to-many* flow of information where messages carrying commands are pushed to a subset of nodes. The Logical Neighborhood routing layer, designed to ensure communication from a node to all the members of a logical neighborhood, provides a foundation towards this goal. Our SPIDEY preprocessor gener-

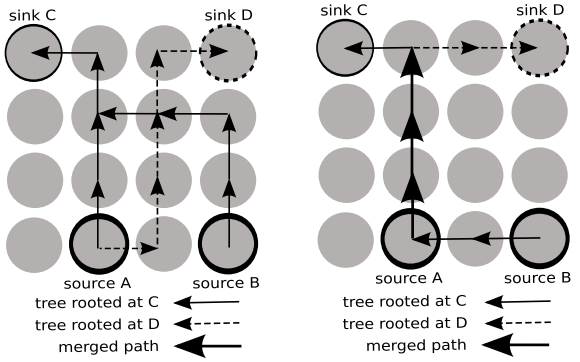
```
includes VirtualNodes;
module ControlStation {
  uses interface virtual_energy_if;
  uses interface virtual_air_conditioner_if;
}
implementation {
  uint8_t isActive = OFF;
  event result_t virtual_energy_if.Reading(int value){
    if (value > ENERGY_THRESHOLD
        && isActive == LOW_POWER) {
      call virtual_air_conditioner_if.Deactivate();
      isActive == OFF;
    } else if (value <= ENERGY_THRESHOLD
               && isActive == OFF){
      call virtual_air_conditioner_if.Activate();
      isActive == LOW_POWER;
    }
    return SUCCESS;
  }
}
```

Figure 9.11.: Complete nesC code for the control station.

ates an appropriate encoding of commands and parameters, which are then packed as application messages dealt with by this routing protocol. Moreover, we extend our original solution with a simple reliability scheme based on implicit acknowledgments, similar to other solutions in WSNs [171], as guaranteed message delivery is required by the `reliably` clause shown in Section 9.2.3.

Virtual Sensors. Unlike virtual actuators, virtual sensors exhibit a *many-to-many* communication pattern: the same subset of nodes (sources) funnels data towards multiple destinations (sinks). In our application, vibration and sound sensors report their readings to both the node controlling the lights and the air conditioner in a room. To accomplish this, we may re-use well-established techniques for data collection, e.g., by building a routing tree rooted at each sink [37].

However, similar techniques are inefficient with multiple sinks: as shown in Figure 9.12(a), different trees are built independently, and a sensible amount of resources is wasted, e.g., by splitting routes (hence duplicating messages) too early. We tackle this problem of multi-source to multi-sink routing with a novel communication scheme that, as illustrated in Figure 9.12(b), maximizes the overlapping among routing trees to decrease the amount of redundant information transmitted. This solution, initially mo-



(a) Two trees rooted at the two sinks are built independently.

(b) Two branches of the trees in Figure 9.12(a) are merged.

Figure 9.12.: Multi-source, multi-sink communication.

tivated by the need for supporting virtual sensors, enjoys wider applicability beyond our programming abstractions, e.g., in case multiple applications are running on different sinks. In this respect, the next chapter presents a mathematical formulation of the multi-source to multi-sink problem, an in-depth description of the mechanisms of our distributed protocol, and an extensive evaluation of the protocol itself.

The above solution still requires modifications to support in-network processing. In our run-time, each sampled data carries a tag (generated by the SPIDEY preprocessor) that determines the function responsible for its processing, e.g., `average()`. When such data is received as part of an application message, the function identified by the tag is executed, and the content of the message possibly changed based on the aggregation semantics. Moreover, our preprocessor also determines which protocol implementation must be linked against which application code, based on the characteristics of the functions employed as specified by the programmer, and on the nodes types found in the application.

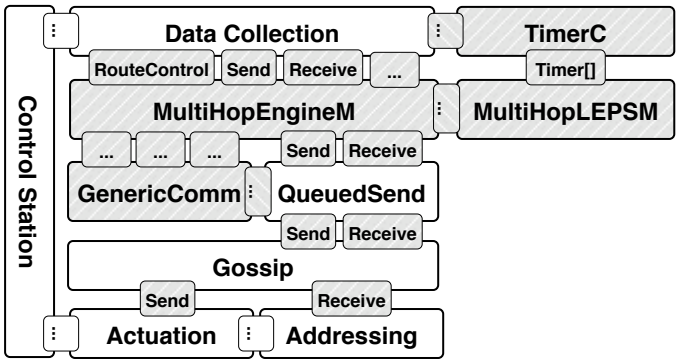


Figure 9.13.: Component configuration on the control station node using plain-TinyOS.

9.4. Evaluation

We evaluate *quantitatively* the benefits that virtual nodes bring to application development and system performance. As we mentioned in Section 9.1, and further discuss in Section 9.5, no existing platform explicitly addresses the application scenarios we consider. In the absence of alternatives, we compare against nesC/TinyOS, by far the most common programming platform for WSNs.

9.4.1. Benefits to the Programmer

The use of virtual nodes impacts beneficially both the design and the implementation phases.

Design. With respect to the design we obtained with virtual nodes, the equivalent implementation of our reference application in plain TinyOS is much more cumbersome. By comparing the plain-TinyOS component configuration shown in Figure 9.13 against Figure 9.10, it is immediate to note how virtual nodes yield a much cleaner and reusable design. Due to the lack of appropriate support in TinyOS, we had to implement lots of functionality not directly related to the application, e.g., an `Addressing` component to determine whether a node is to perform a given action. Further, to the best of our knowledge, no general purpose routing protocol exists to deliver messages to arbitrary subsets of nodes. Therefore, we developed a

`Gossip` component based on probabilistic forwarding of messages [175], as similar solutions are known to provide good performance with reasonable overhead.

It is also worth noting how Figure 9.13 is the result of several design iterations. Initially, we intended to re-use the standard TinyOS multi-hop protocol [176] for data collection. However, due to the need of supporting other concerns (e.g., actuation) the planned design failed. Most often, this was due to run-time memory overflows that we could not solve given the limited memory budget of WSN devices and the intrinsic difficulty in tuning the various protocol parameters. To address this issue, we had to dissect the TinyOS multi-hop protocol to investigate if some optimizations were possible and, luckily, we found a way to share the queueing mechanism between multi-hop routing and gossip. In doing so, however, we had to modify or rewrite about 40% of the code, dramatically increasing the development time. Instead, we did not experience similar issues with virtual nodes: the design in Figure 9.10 worked without further modifications.

Implementation. The cleaner design enabled by virtual nodes is reflected in simpler code within individual components. Typically, the code complexity of WSN implementations is evaluated by analyzing them as state machines [21, 63, 64], as processing is usually expressed as state transitions triggered by external events (e.g., a sensor reading above threshold). Specifically, the number of *explicit states* is regarded as the main measure of complexity [21]. For instance, an explicit state is one that describes the current condition of an air conditioner with a value among `on`, `low-power`, and `off`. The higher the number of states, the more difficult is to express state transitions [64], and the more complex and error-prone are the implementations. Counting the number of states in TinyOS implementations is straightforward, as they are typically stored in global variables inside components, such as `isActive` in Figure 9.11.

Figure 9.14 reports the metric above for some node types in our application. As the plain-TinyOS implementation deals with communication directly, it requires several states, for instance, to keep track of whether the radio is busy transmitting. These are not needed in our approach, as all distribution concerns are hidden from the programmer. Further, TinyOS requires variables to coordinate the gathering of data from the sensor devices with their reporting to the actuators. These variables are easily source of race conditions [28], as they are modified concurrently across components. Again, we do not need them because our preprocessor automatically (and

Node Type	Explicit states			Lines of code			% of hand-written code		
	Virtual nodes	Plain TinyOS	Reduction	Virtual nodes	Plain TinyOS	Reduction	Virtual nodes	Plain TinyOS	Reduction
Control Station	2	18	88.8%	57	1032	94.4%	12.8%	48.2%	73.4%
Air Conditioner	2	16	87.5%	118	972	87.8%	14.5%	49.1%	70.4%
Sensors (any)	0	8	100%	84	672	87.5%	5.7%	41.2%	86.1%

Figure 9.14.: Comparing a virtual node-based implementation against plain TinyOS. (SPIDEY specifications are counted as lines of code).

safely) generates the corresponding functionality. Remarkably, the only state variables needed with virtual nodes are related to the very application processing, e.g., the `isActive` variable in Figure 9.11 keeping track of the current state of a set of air conditioners.

Figure 9.14 also evidences how the reduction in the number of explicit states causes a 90% decrease in lines of code, on the average. Indeed, fewer state transitions are to be expressed, and much less bookkeeping code is needed. Simpler implementations foster highly reusable components: as long as the control logic remains unmodified, the `ControlStation` in Figure 9.11 can be reused *as is*, e.g., to shut down the air conditioners on a floor, instead of those in the entire building. TinyOS provides much less decoupling, and similar modifications require changes in several places (e.g., the `Control`, `Actuation`, and `Addressing` components in Figure 9.13).

The rightmost column in Figure 9.14 reports the *fraction of hand-written code* w.r.t. the total code deployed. In a sense, this captures the expressive power of the programming abstractions. Using virtual nodes the programmer writes only a small fraction of the final running code, as most of it is generated by our preprocessor. TinyOS abstractions, instead, are not expressive enough and require a substantial programming effort, as illustrated earlier by Figure 9.13.

Finally, the size of the binary code loaded on the real nodes is comparable. The control station node represents the worst case, with a binary image of 43.5 Kbytes using virtual nodes, against the 41 Kbytes using TinyOS.

9.4.2. System Performance

WSN nodes are typically battery-powered. Hence, system *lifetime* is regarded as the main performance metric in WSNs [2]. To evaluate it, we

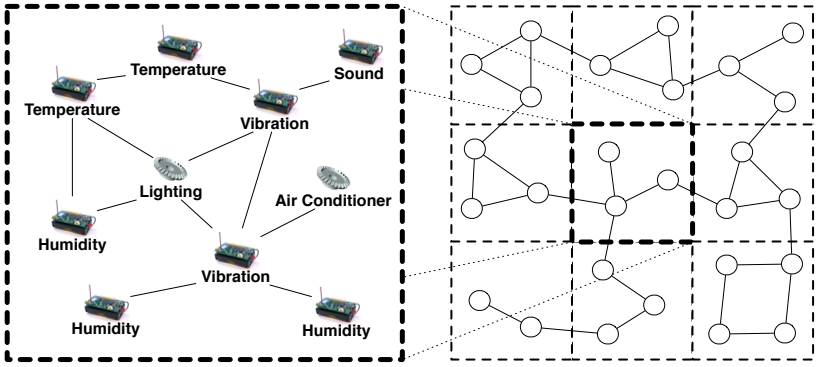


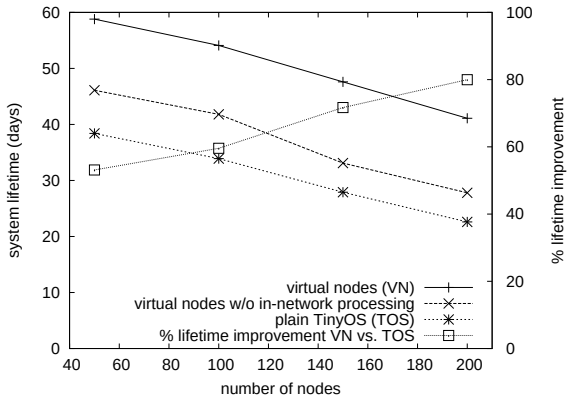
Figure 9.15.: Sample topology used in experiments.

conducted experiments using Avrora [157], a cycle-accurate emulator for the popular MICA2 node [72]. Unlike other simulators (e.g., TOSSIM [124]), Avrora's fine-grained emulation considers all aspects concurring to energy consumption, from radio communication to CPU processing.

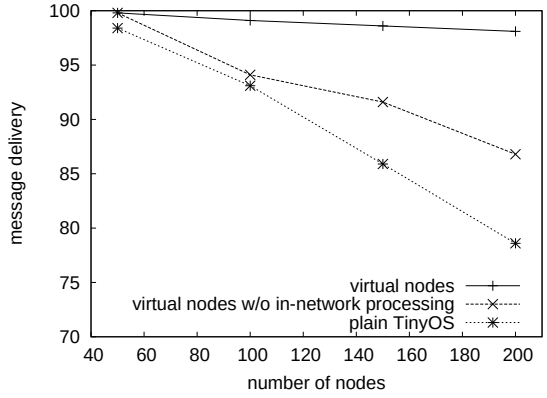
Settings and Metrics. A variable number of nodes is randomly placed in a connected topology, as in Figure 9.15. The field is divided in smaller subareas representing single locations in the building. Each location contains an average of 10 nodes, with at least one sensor node per type, and two actuator nodes as air conditioner and light controller. Additionally, a single actuator is placed randomly, acting as the control station. On the average, each node has 4 neighbors. For each combination of parameters, 10 different topologies are generated and the results averaged.

Each sensor generates a reading every 30 s, while a controller sends a command to its target actuators every 5 min. We set the forwarding probability of the plain-TinyOS `Gossip` component to 0.75, a value sufficient to deliver the commands to operate the actuators. For virtual nodes, we use the reliability scheme mentioned in Section 9.3.2. The maximum packet size is 60 bytes for virtual nodes, and 29 bytes for plain TinyOS. Virtual nodes require longer messages due to increased control information. We use the standard TinyOS MAC layer.

To evaluate the *system lifetime*, each node is given an initial energy budget comparable to a pair of AA batteries. The experiments stop when the first network partition preventing the application to close the control loop



(a) System lifetime.



(b) Data delivery ratio.

Figure 9.16.: Virtual nodes performance.

occurs, e.g., when only one path to an actuator remains, and a node dies on that path. In addition, we also evaluate the *delivery ratio* of messages flowing from the sensor nodes to controllers. This gives a measure of the level of service provided by the running system. If messages are lost on this path, the information available at a controller node may not be accurate.

Results. Figure 9.16 shows the results of our experiments. Our dedicated run-time support drastically increases the system lifetime w.r.t. a plain-TinyOS implementation, up to a 80% factor. Moreover, more than half of the gain is due to the automatic customization made by our preprocessor. Indeed, the ability to perform in-network processing (e.g., while computing

the average vibration readings based on the code in Figure 9.5) significantly reduces the amount of data that needs to be transmitted, which bears great influence on energy. Without in-network processing, the system lifetime improves "only" by a factor of 35%, on the average.

Analogous observations hold for the delivery ratio, shown in Figure 9.16(b). The reduction in network traffic due to in-network processing corresponds to less congestion on the wireless medium, which yields fewer packet collisions. Consequently, packet delivery increases, and our solution scales much better w.r.t. the same routing scheme without in-network processing and the standard TinyOS routing.

In principle, similar in-network processing mechanisms can be coded by the programmer on a per-application basis, at the cost of hampering their reuse. Instead, virtual nodes harmonize the expressiveness and flexibility provided by the programming abstraction with the effectiveness of the underlying run-time, empowering programmers with the tools for developing reusable and efficient implementations.

9.5. Related Work

We conciliated expressiveness and efficiency by co-designing the language constructs and the supporting distributed protocols. Similar approaches are Directed Diffusion [37] and TinyDB [17]. In the former, data is named by attribute-value pairs and the user poses queries by expressing constraints on the data of interest. At the network level, this is implemented by flooding a user's query throughout the whole system, and setting up gradients along the reverse path to the query source. Instead, TinyDB offers a database-like interface, whereby the user retrieves the data of interest by expressing queries with a subset of SQL. A tree topology is exploited for routing, with several optimizations to avoid querying sensors that would not provide useful data.

Our work departs from the above, in that we focus on *actuation* in addition to sensing. The solutions above are indeed ill-suited to cope with actuation, being essentially designed for data collection. As a consequence, our run-time must account for multiple data consumers (e.g., multiple actuators receiving data from the same sensors), as opposed to the many-to-one pattern in TinyDB and Directed Diffusion.

Although ours is an integrated solution, some of the individual problems we tackled in this work have been widely researched.

Programming Abstractions. Constructs for programming groups of resource-constrained devices have been proposed in [102,109]. These solutions, however, target specific applications where processing exhibits physical locality, e.g., object tracking. Virtual nodes straightforwardly encompass these scenarios, while providing more powerful constructs enabling a *logical* notion of proximity, rather than physical.

Notions of scoping similar to Logical Neighborhoods have been proposed in [14,21]. In these cases, there is no programming support to express the very application processing, as opposed to our solution where virtual nodes leverage off logical neighborhoods to describe the application operation at a higher level of abstraction. Moreover, [21] only targets 1-hop neighborhoods, whereas [14] essentially requires the implementation of a dedicated routing support for each different type of region defined.

Data Aggregation. TAG [130] and TinyDB [17] provide data aggregation constructs. Their programming model, however, is inherently data-centric, lacking constructs to deal with actuators. Again, this prevents their use to implement control systems like the ones we target.

At the routing layer, several solutions have been proposed. For instance, the work in [177] studied the placement of aggregation operators to minimize network traffic. Data aggregation in the presence of multiple, mobile sinks is investigated in [178]. The problem we consider here is different from the ones above. We do not take mobility into account, as we target systems deployed in controlled environments, e.g., buildings. Moreover, we consider scenarios with both multiple sources and multiple sinks. However, we are investigating if some of the techniques above can be borrowed and adapted to our goals.

Data-Centric Routing. Looking at the vast literature in WSN routing, it is easy to recognize how most solutions aim at optimizing communication from multiple sources to a single sink [33]. However, the scenarios we target inherently calls for routing solutions to report to multiple sinks. Similar problems have been investigated in [179, 180]. As in our scheme, in [179] broadcast transmissions are used to let nodes collect information on alternative routes. However, the sources are not aware of each other, thus missing the opportunity to share paths towards receivers. The solution in [180] adjusts the sensing rate at different nodes to eliminate the redundancy in the data gathered. Instead, we do not assume the ability to influence the source behaviors.

10. Routing from Multiple Sources to Multiple Sinks

Early deployments of WSNs were based on a a single sink collecting data from a number of sources. Recently, however, scenarios where multiple sources must communicate with multiple sinks are increasingly being proposed, e.g., to deal with actuator nodes. The resulting many-to-many *communication pattern is difficult to address using state-of-the-art WSN routing protocols. Existing solutions for single-sink routing are straightforwardly inefficient, as the* many-to-one *interactions they foster is ill-suited to the requirements at stake. Likewise, multicast protocols for wireless networks do not capture some of distinctive traits of the scenarios considered. Indeed, they usually foster* one-to-many *interactions that miss the opportunity of making sources collaborate with each other.*

In this chapter, we present a novel routing scheme in support of many-to-many interactions in WSNs. We first study the problem from a theoretical perspective, using a mathematical formulation inspired to the multi-commodity network design problem. We derive an optimal solution that, albeit based on global knowledge, provides us with a theoretical lower bound to evaluate decentralized solutions against. We then illustrate an adaptive, distributed protocol that minimizes the number of message exchanged while balancing the routing load. Adaptation is driven by two orthogonal metrics: the routing quality of a node, dictated by topological information, and its expected lifetime. Our evaluation shows how our protocol performance lies within 10% from the optimal solution, and yields a 75% increase in network lifetime. A preliminary version of this work appeared in [30].

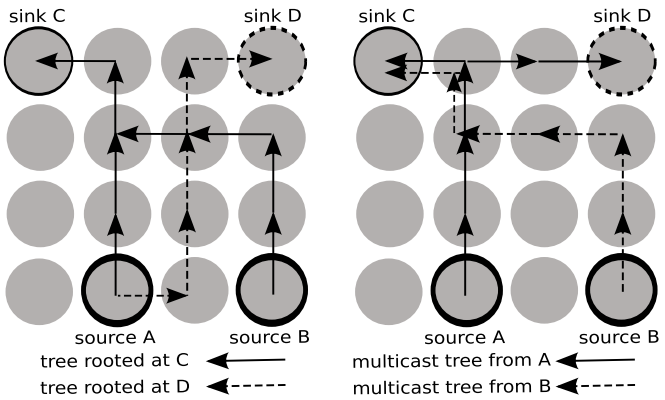
10.1. Motivation

Early deployments of wireless sensor networks (WSNs) were based on a single-sink architecture, e.g., as in habitat monitoring applications [22]. Recent developments, however, increasingly call for scenarios where the

data sensed fro multiple sources must be delivered to multiple sinks. For instance, similar communication patterns are needed in support of high-level programming constructs for sense-and-react scenarios, such as the virtual node abstraction described in Chapter 9. In this case, the physical nodes in the system need to communicate their data to multiple receivers, where a different processing is performed. However, the need for a similar communication pattern arises in also other situations, e.g., when the same WSN is serving multiple applications, each running on distinct devices. Moreover, multiple sinks are increasingly required to implement advanced applications. For instance, data collection is evolving into complex in-network data mining [181]. In these applications, the mining process is distributed across the nodes in the system, each collecting readings from different sets of data sources.

Unfortunately, existing protocols and algorithms are ill-suited to cope with scenarios where data must be reported from multiple sources to multiple sinks, i.e., in a *many-to-many* fashion. Most routing protocols in WSNs focus on reporting data from multiple sources to a single sink, addressing the needs of *many-to-one* communication. To address multi-sink scenarios, they simply *replicate* the routing infrastructure. For instance, the well-known Directed Diffusion protocol [37] sets up a tree along which sources report their data to the single sink. Dealing with multiple sinks involves setting up a separate, independent tree for each sink—a rather inefficient solution. Multicast routing protocols, on the other hand, are also an imperfect match for the scenarios at stake. Usually, they implement a *one-to-many* communication pattern that straightforwardly misses the opportunity of making sources collaborate among themselves. For instance, data aggregation or fusion techniques cannot be applied.

To see why this is a problem, consider the sample scenario with two sources and two sinks illustrated in Figure 10.1(a). Node A reports data to both sinks, whereas node B only transmits to sink C. To achieve multi-hop communication, a traditional tree-based routing protocol would build two *independent* trees rooted at the two sinks, e.g., by flooding a control message from each sink and having each node remember the reverse path to the sink, as in [37]. This base solution involves 13 nodes in routing. Moreover, to report to the two sinks node A is forced to duplicate its data right at the first hop. Differently, Figure 10.1(b) depicts how a multicast routing protocol would address the scenario considered above. Although node A does not duplicate its data right at the first hop, readings originated



(a) Two trees rooted at the two sinks are built independently.

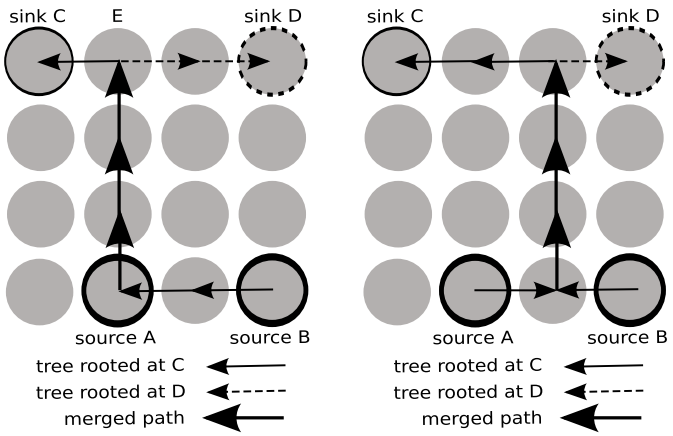
(b) Multicast trees for the scenario in Figure 10.1(a).

Figure 10.1.: A sample multi-source to multi-sink scenario.

from node A and node B are still routed independently. For instance, this makes the system unable to take advantage of data aggregation techniques when applicable, as data from different sensors travel separately.

10.2. Contribution

In this chapter, our goal is to support efficient routing from multiple sources to multiple sinks. In this respect, Figure 10.2(a) illustrates a better solution to the scenario in Figure 10.1(a), based on the scheme we describe in the rest of the chapter, obtained by adapting the routing topology to minimize the number of nodes involved in routing messages. The two parallel branches starting from node A have been merged in a single one, and node B leverages off this merged path instead of relying on an independent one. As a consequence, only 9 nodes are involved in routing. By reducing the number of nodes involved, we decrease the amount of redundant information flowing in the network, and duplicate data only if and when strictly necessary. This increases the system life-time, and reduces the contention on the wireless medium and packet collisions, therefore ultimately increasing reliability. Moreover, the readings coming from the two sources can be either aggregated using some application-specific processing along the



(a) Two branches of the trees in Figure 10.1(a) are merged.

(b) The merged path transitions to a different set of nodes to balance the load.

Figure 10.2.: An efficient solution to routing from multiple sources to multiple sinks.

merged path, or simply packed in a single physical message to reduce the per-reading header cost.

Nevertheless, an undesirable side-effect of the above solution may be to deplete the energy available at nodes on the merged path more quickly than in the rest of the system. This may yield an uneven degradation of performance, to the point of partitioning the network even though some nodes would still be able to operate. To address this issue, we employ a novel scheme to balance the routing load among the nodes possibly involved. Our solution "juggles" with the routes whenever alternative paths exists that may guarantee a longer system lifetime. For instance, our solution allows the routing topology shown in Figure 10.2(a) to eventually morph into the one in Figure 10.2(b). The latter configuration has equal cost w.r.t. the former in terms of nodes involved in routing, yet it leverages off a different set of devices along the merged path, thus achieving a better overall balancing. At a later time, however, the system may return to the configuration in Figure 10.2(a) if deemed convenient to improve the system lifetime w.r.t. the current situation.

Our scheme enhances the traditional single-sink tree-based solution, thus

enabling easy integration of our solution into existing routing schemes, e.g., [37]. Therefore, we assume the presence of a very basic routing infrastructure made of separate trees connecting the sources to the corresponding sinks. In this case, a *single path* connecting a given source to each sink is always established. This is a common approach in WSNs, motivated by the reduction in network traffic w.r.t. a solution exploiting multiple paths from a source to the same sink. Furthermore, we do not make any assumption about the pairing of sources and sinks, as it is indeed determined by the initial tree structure. Given this setting, our objective is *to enable efficient routing from the sources to the corresponding sinks by minimizing the number of nodes involved and maximizing the system lifetime.*

To achieve our goal, our contribution is twofold:

- We present a theoretical model of the problem, inspired to the multicommodity network design problem [182, 183]. Indeed, graph algorithms usually employed to study multicast protocols are not applicable in our case, as the metric we minimize is the number of nodes employed for routing by considering all the sources at once. Differently, algorithms such as minimal spanning trees consider a single source at a time. Thanks to our formulation, we reuse available results and tools for integer programming to compute the theoretical optimal topology for our routing problem. The model and optimal solution are illustrated in Section 10.3. This technique, however, assumes global knowledge of the system topology and is therefore derived in an off-line, centralized fashion, impractical for real WSN deployments. Nevertheless, this theoretical result is valuable for providing a lower bound against which to compare more practical and decentralized solutions.

- We present and evaluate our own decentralized solution, based on a periodic adaptation of sink-rooted trees. The adaptation consists of selecting a different neighbor as the parent towards a given sink. The decision to adapt is taken locally by a node and is based on two metrics: i) a *routing quality* figure based on topological information, and ii) the *expected lifetime* of a node. Our adaptive protocol, whose details are illustrated in Section 10.4, is simple enough to be easily implemented on resource-scarce WSN devices. At the same time, as shown in Section 10.5, it is able to improve the system lifetime of about 75% w.r.t. the base solution with independent trees, a result close to the theoretical optimum we derive in Section 10.3.

We conclude the chapter with a survey of related efforts in Section 10.6.

10.3. System Model and Optimal Solution

In this section we provide a mathematical characterization of our problem. Besides providing a formal foundation for the results presented here, in this section we show how our model can be used to derive the optimal topology for our routing problem, using tools for mathematical programming.

System Model. Our model is inspired to the multi-commodity network design problem [182]. In the most common formulation of this problem, we are given a directed graph (e.g., representing a road network) with node set $\mathcal{N}$ and arc set $\mathcal{A}$, and a set of commodities $\mathcal{C}$ (e.g., a set of physical goods). The goal is to route each commodity $k \in \mathcal{C}$ from a set of sources $O(k) \subseteq \mathcal{N}$ to a set of destinations $D(k) \subseteq \mathcal{N}$, by minimizing a given metric.

We can straightforwardly model a WSN as a directed graph whose node set $\mathcal{N}$ is composed of the WSN devices, and whose arc set $\mathcal{A}$ is obtained by setting an arc (i, j) between two nodes i and j when the latter is within the communication range of the former. Without loss of generality, as shown in [183], a commodity can be assumed to flow from a single source to a single destination. In this case, since commodities generated from the same source and directed to the same destination follow the same route, one can state a one-to-one mapping between the route connecting any source-sink pair $(o(k), d(k))$, and any commodity k.

With this notion of network, we can capture message routing with a set of decision variables:

$$r_{i,j}^{k} = \begin{cases} 1 & \text{if the route for the source-sink pair } k \text{ contains arc } (i,j) \\ 0 & \text{otherwise} \end{cases} \tag{10.1}$$

A value assignment $\forall (i, j) \in \mathcal{A}$ to these variables formally represents the route messages follow from the source $o(k)$ to the sink $d(k)$.

Metric. Usually, the metric of interest in the multi-commodity network design problem is the number of arcs exploited for routing. For instance, in modeling a transportation system the number of arcs used represent the single trips required to transport a physical good between two locations. Applying the same metric in WSNs, however, does not capture the broadcast nature of the wireless medium. For instance, compare Figure 10.3 against Figure 10.2(a). In case the objective is to minimize the number of

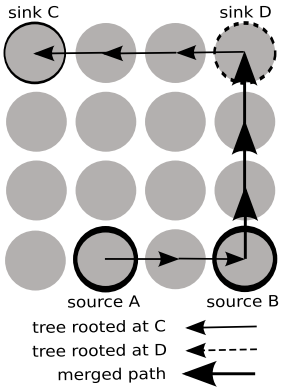


Figure 10.3.: A routing topology where all transmissions are pair-wise.

network links exploited for routing, both solutions look optimal. Nevertheless, the configuration shown in Figure 10.2(a) is preferable in a broadcast network, since node E can forward data to different destinations using a single broadcast transmission. As the minimum number of broadcast transmissions involved corresponds to the number of nodes along the route, in our model we minimize the number of nodes instead of network links. To this end, we capture the fact that node i is involved in *at least one* source-sink route as:

$$u_i = \begin{cases} 1 & \text{if } \exists k \in \mathcal{C}, j \in \mathcal{N} \mid r_{i,j}^k = 1 \\ 0 & \text{otherwise} \end{cases} \tag{10.2}$$

and define our metric of interest as:

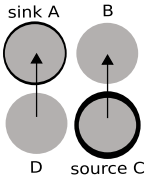
$$\mathit{NodesInvolved}(\mathcal{C}, \mathcal{A}) = \sum_{(i) \in \mathcal{N}} u_i \tag{10.3}$$

Based on the above, our goal consists of finding the optimal set of routes used to deliver data messages from sources to sinks. Formally:

> *Goal*: to find the value assignment of $r_{i,j}^k, \forall k \in \mathcal{C}, \forall (i,j) \in \mathcal{A}$ that minimizes the value of $\mathit{NodesInvolved}(\mathcal{C}, \mathcal{A})$.

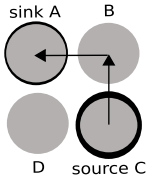
The relation between $r_{i,j}^k$ and u_i defined in (10.2) captures the essence of the problem, as well as the rationale of our distributed solution, presented next. Indeed, to minimize *NodesInvolved* one should aim at reusing as

Variable	Value
$r_{C,B}^{C,A}$	1
$r_{D,A}^{C,A}$	1
Remaining $r_{i,j}^{C,A}$	0



(a) An assignment and topology representing non-consistent routes for a commodity flowing along the source-sink path (C, A). Node B and D do not obey to (10.4).

Variable	Value
$r_{C,B}^{C,A}$	1
$r_{B,A}^{C,A}$	1
Remaining $r_{i,j}^{C,A}$	0



(b) An assignment and topology representing meaningful routes for a commodity flowing along the source-sink path (C, A). Constraint (10.4) holds for every node.

Figure 10.4.: Sample assignments for $r_{i,j}^{C,A}$.

much as possible nodes along the routes serving other source-sink pairs, i.e., for which the cost u_i is already paid. Minimizing the nodes involved in routing, in turn, corresponds to *maximizing the overlapping* among source-sink paths. In Section 10.4 we present a protocol for achieving this goal.

Although this formalization of the problem is simple and general, alternatives exist and are discussed in Section 10.6.

Finding the Optimal Solution. Based on the model we just presented, we can derive an optimal solution using techniques of mathematical programming, provided that we specify the constraints to be satisfied by any meaningful solution. We first require that $r_{i,j}^k$ and u_i are integer, binary variables and that the following relation holds among them:

$$\forall (i,j) \in \mathcal{A}, \forall k \in \mathcal{C}, \quad r_{i,j}^k \leq u_i$$

In our case, these constraints are satisfied by construction through (10.1) and (10.2).

Most importantly, we state the requirement that the assignment to $r_{i,j}^k$ contains a connected, end-to-end path for each source-sink pair k. This can be expressed by requiring every node different from the source $o(k)$ and the

sink $d(k)$ to "preserve" the message, i.e.:

$$\forall i \in \mathcal{N}, \forall k \in \mathcal{C}, \quad \sum_{m:(i,m)\in\mathcal{A}} r_{i,m}^k - \sum_{n:(n,i)\in\mathcal{A}} r_{n,i}^k = \begin{cases} 1 & \text{if } i = o(k) \\ -1 & \text{if } i = d(k) \\ 0 & \text{otherwise} \end{cases} \tag{10.4}$$

The previous expression is similar to a network flow conservation equation, and effectively imposes the existence of a multi-hop route from each source to every sink. Figure 10.4 illustrates the concept in the case of a single source-sink pair. The solution in Figure 10.4(a) is not acceptable, as the message originated at C and directed to A is lost at node B and suddenly reappears at node D. Indeed, the constraint in (10.4) does not hold for node B and D, as its left-hand side evaluates to -1 when $i = B$ and to 1 for $i = D$, and neither node is an origin or destination for the source-sink path. Conversely, the solution in Figure 10.4(b) is perfectly meaningful: a connected, multi-hop path from the source to the sink exists, and indeed the constraint in (10.4) holds for every node.

With this formulation, the problem of finding the optimal assignment that satisfies our goal can be solved straightforwardly using well-established techniques and tools from mathematical programming. These techniques require global knowledge of the system state and are computationally expensive, and therefore impractical for WSNs. For this reason, we devised a distributed scheme that relies only on local (i.e., within the 1-hop neighborhood) knowledge, and can be implemented on resource-constrained devices. We return to the theoretical optimal solution in Section 10.5, where we show how it is efficiently approximated by the distributed solution, discussed next.

10.4. A Distributed Solution

As we discussed in Section 10.2, in our solution the goal of minimizing the number of nodes involved in routing and that of balancing the load are played in an integrated manner. In this section, we first illustrate the mutual interplay between the solutions we devised to achieve this goal, and then detail the single mechanisms.

(a) High routing quality neighbor with short lifetime.

(b) Medium routing quality neighbor with longer lifetime.

Figure 10.5.: Sample interplay between routing quality and expected lifetime.

10.4.1. Protocol Overview

We assume that the initial state of the system is such that a tree exists for each sink, connecting it to all the relevant sources. These sink-rooted trees are easily built using mechanisms available in the literature, e.g., along the reverse path of interest propagation as in Directed Diffusion [37]. These mechanisms are designed to build each tree independently of the others, and therefore do not guarantee any property regarding their overlapping.

As already mentioned, minimizing the nodes involved in routing can be achieved by maximizing the overlapping of the paths used to route data from a given source to a given sink. In our solution, adapting the routes to achieve this goal is based on topological information about the neighbor nodes, e.g., how many overlapping source-sink paths it is currently serving, piggybacked on application messages and overheard during transmission. This control information is fed into a routing metric $R(n, s)$ that yields a measure of the quality of a neighbor n as the parent towards a sink s, and evaluated each time a node has data to send to s.

To balance the load, we periodically evaluate the expected lifetime $T(n)$ at every node n. The specific technique used for this purpose is described in Section 10.4.3. This quantity is multiplied by the value of $R(n, s)$ to obtain a time-extended quality metric $Q(n, s) = R(n, s) \cdot T(n)$. Indeed, as intuitively shown in Figure 10.5, given a sink s Q provides a measure of the routing metric for a specific neighbor n extended over time. The neighbor whose time-extended quality metric Q is shown in Figure 10.5(a) has high

routing quality, yet it will not be able to serve for a long period of time. Differently, the neighbor whose time-extended quality metric Q is depicted in Figure 10.5(b) has lower routing quality. However, it is able to serve for a longer period of time, and is therefore preferable w.r.t. the former.

The time-extended quality metric Q is used at each node to manipulate the source-sink paths by changing the neighbor serving as its parent towards a given sink. Changing the current parent to a different neighbor n occurs when $Q(n, s)$ is maximum among all neighbors for sink s. If so, the node simply begins forwarding data to the new parent. Note that alternating among many different routes is the most natural way to balance the load. Our solutions achieves precisely this behavior, due to varying values of $T(n)$ over time that, in turn, influence the value of Q. Nevertheless, switching to a different parent does not incur in any additional cost. This operation is indeed managed without additional control messages by using a timeout.

10.4.2. Computing the Routing Quality

In principle, the routing quality metric R can be designed to rely on various quantities. Here, we present and evaluate an instantiation of our protocol where our routing quality metric relies on:

- $dist(j, s)$, the distance (in hops) from a node j to a given sink s, as determined by the initial interest propagation;
- $paths(j)$, the number of source-sink paths passing through a given node j, i.e., using the notation in Section 10.3:

$$paths(j) = \sum_{k \in \mathcal{C}} r^k_{i,j} \quad (i, j) \in \mathcal{A}$$

- $sinks(j)$, the number of sinks a given node j currently serves.

The distance between a neighbor and a sink is of fundamental importance in increasing reliability and reducing overhead. Indeed, the higher the number of nodes traversed by a message, the higher the probability to lose data due to unreliable transmission, and the higher the overall computational and communication cost paid to deliver the message end-to-end.

The rationale behind the choice of the other two quantities can be visualized with the help of Figure 10.6. In the network shown, a source Z needs to send data to the sink S, and to do so it routes messages upstream through

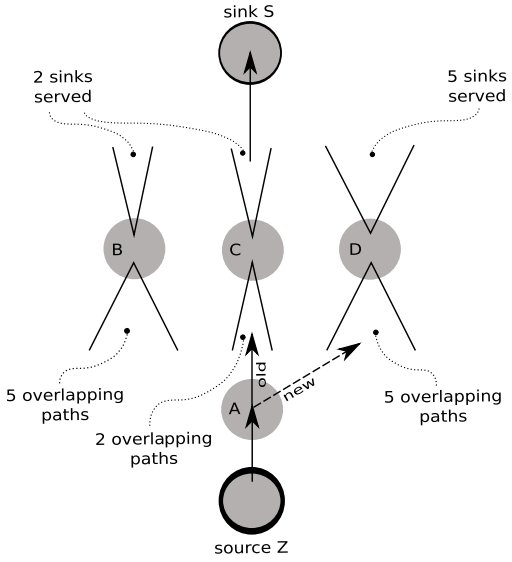


Figure 10.6.: An abstract view of a WSN with multiple sources and multiple sinks. Source Z generates data to be delivered to sink S, routed through node A. Besides Z, node A is a neighbor of B, C, and D. At node A, the current parent towards S is C. However, a better choice is represented by D, since it enjoys the highest number of overlapping paths and served sinks among A's neighbors.

its neighbor A. Node A, in turn, has three neighbors B, C, and D, with C being the current parent in the tree rooted at S. Nevertheless, the figure also shows how both B and D are currently traversed by more source-sink paths than node C. Therefore, if A were to choose either of these neighbors as the new parent towards S, they would guarantee higher *overlapping* than in the current situation—which is exactly our goal. Finally, the figure also shows that D is serving more sinks than node B. Therefore, with respect to B, D is more likely[1] to be already reporting readings to S, possibly on behalf of other sources. If this is actually the case, choosing D leads to reusing an "already open" path towards S, therefore further increasing the

[1] As we know only the number $sinks(j)$ of sinks served by j we cannot be sure that S is really among them. To obviate to the problem, we could propagate the *identifier* of the sinks served instead of their number. However, as shown in Section 10.5, the latter already yields good performance and generates much less overhead.

Field Name	Description
neighborId	The identifier of the neighbor relative to this entry.
dist	An associative array containing, for each sink in the system, its distance from *neighborId*.
paths	The number of different source-sink paths currently passing through *neighborId*.
sinks	The number of sinks served through *neighborId*, possibly along a multi-hop path.

Figure 10.7.: Information used to compute the routing quality metric for a neighbor node.

overlapping of source-sink paths at no additional cost. Therefore, node D is the highest routing quality neighbor of A.

Here, we designed q to be a linear combination of the three quantities above:

$$q(n,s) ::= \delta \cdot dist(n,s) + \alpha_1 \cdot paths(n) + \alpha_2 \cdot sinks(n) \tag{10.5}$$

where $\delta, \alpha_1, \alpha_2$ are tuning parameters of the protocol. Again, the shape of the function R and its constituents can in principle be different. Although the results presented in Section 10.5 with the routing quality metric in (10.5) are already very positive, investigating the impact of alternative definitions of R is in our immediate research agenda.

To compute $R(n,s)$ for a given neighbor n and sink s, a node must first determine the three constituents $dist(n,s), paths(n)$, and $sinks(n)$. These are evaluated by relying on a data structure maintained by each node. Figure 10.7 shows the data structure fields for a single neighbor. Note how the various fields are maintained differently. The value of the field *neighborId* serves as the key to index the data structure. The content of *dist* is determined from the messages flooded by the sink either during the tree setup phase, or in successive flooding operations performed to keep this information up-to-date with respect to nodes joining or failing. The values of $paths(n)$ and $sinks(n)$ are instead derived by the node through overhearing of messages sent by n. These messages piggyback the control information above, which can then be used to update the data structure in Figure 10.7. Note how the overhead due to this additional control information is very small: only two integer values are needed.

10.4.3. Computing the Expected Lifetime

Evaluating the expected lifetime of WSN devices is a challenge per se, as it depends on various factors such as the amount of processing generated by the application, the network traffic, and the characteristics of the hardware employed. Here we propose a simple mechanism to compute this metric. This proved to be sufficiently accurate for our purposes, as we discuss in Section 10.5.2. Nonetheless, our approach decouples this aspect from the rest of the protocol, allowing for the use of alternative approaches, e.g., more sophisticated models taking into account the non-linear behavior of commercially available batteries [184, 185].

In our solution, each node periodically evaluates its remaining energy. Some commonly used WSN nodes are able to provide this value through dedicated hardware probes, e.g., [186]. Otherwise, most of the WSN nodes are able to read the current battery voltage, which can be used to compute the remaining energy based on information about the current draw as reported in the node data sheets. The difference between two consecutive readings represents the energy consumption E_i during the i-th time interval.

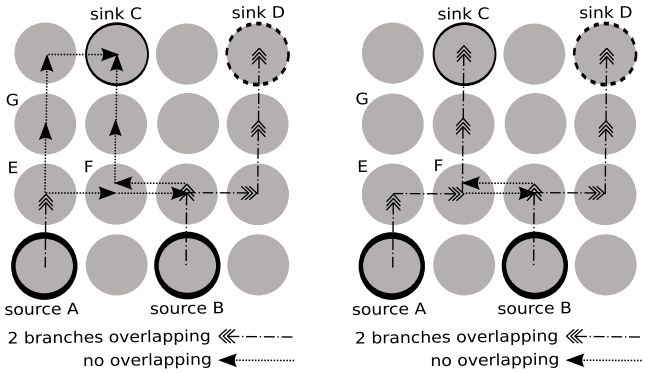
The N most recent E_i are fed as input to an exponential moving average (EMA) of type:

$$\overline{E_i} = \alpha \cdot E_{i-1} + (1 - \alpha) \cdot \overline{E_{i-1}} \tag{10.6}$$

We choose an EMA as it reacts faster to recent changes than other types of moving averages, thus allowing the system to adapt quickly to sudden events affecting the overall system state, e.g., whenever a node dies. To account for the limited memory budget of WSN nodes, we express the smoothing factor α as suggested in [187], namely, in terms of the N measurements that we allow in memory:

$$\alpha = \frac{2}{N + 1} \tag{10.7}$$

The above formulation indeed gives more accurate results when the moving average spans a limited time frame. $\overline{E_i}$ provides us with an estimate of the current energy drawing of the node. Using this value and the current energy budget, we can compute $T(n)$ as the number of time intervals the node is going to survive given the current situation.



(a) Initial configuration.

(b) Node E changes its parent from G to F.

Figure 10.8.: A sample adaptation process.

Field Name	**Value**
neighborId	**G**
dist	$\{C = 2, D = 4\}$
paths	1
sinks	1
neighborId	**F**
dist	$\{C = 2, D = 4\}$
paths	2
sinks	2

Figure 10.9.: Data stored at node E in the situation depicted in Figure 10.8(a).

10.4.4. Putting All Together

Figure 10.8 illustrates a sample adaptation process. For the sake of the example, we focus on node E and sink C, and we assume $T(F) = 6$, $T(G) = 5$, and $\delta = \alpha_1 = \alpha_2 = 1$ in (10.5). With these values, node E evaluates $R(n, s)$ towards sink C for its two neighbors F and G. Figure 10.9 shows the content of the data structures in Figure 10.7 for F and G. The evaluation returns $R(G, C) = 2 + 1 + 1 = 4$ and $R(F, C) = 2 + 2 + 2 = 6$. Therefore, E computes $Q(G, C) = 4 \cdot 5 = 20$ and $Q(F, C) = 6 \cdot 6 = 36$, thus recognizing F as the best next-hop towards C. Based on this, E changes its parent to F, as depicted in Figure 10.8(b). The immediate

benefit of this change can be easily seen by computing the number nodes involved: the network in Figure 10.8(a) uses 12 nodes, against the 10 nodes of Figure 10.8(b).

To break ties between the current parent and a new one, a node always selects the latter, as it is guaranteed to enjoy a higher value of R after becoming a parent. Indeed, at least the number of source-sink paths passing through it increases by one. In selecting the new parent, the only additional constraint is to not select as a new parent a neighbor whose distance from a sink is greater than that of the selecting node. Without this constraint, a node could potentially select one of its children as the new parent, hence creating a routing loop.

Finally, our distributed protocol is complemented by a simple scheme for packing multiple readings in the same network message. To this end, each node maintains a buffer for each neighbor, limited by the number of readings allowed in a message. Upon receiving a reading from another node, the reading is inserted in the buffer for the neighbor on the route to the target sink. When the buffer for a given neighbor is full (or upon expiration of a timeout) a message is created and forwarded to the neighbor. This simple scheme decreases the per-reading header cost and helps reducing collisions, since buffers are likely to become full at different times and therefore messages are going to be reasonably spread in time. In principle, the same packing scheme can be used without our adaptation protocol. However, its impact is greater in the presence of adaptation, since the latter guarantees a higher degree of overlapping among trees, with more readings being funneled through the same links.

10.5. Evaluation

In this section, we assess the effectiveness of our protocol using cycle-accurate emulation of WSN devices. Specifically, we implemented our protocol on top of TinyOS [42], and used the Avrora [157] emulator to study its dynamics. The latter allows for fine-grained emulation of the popular MICA2 platform [72], and also includes a detailed energy model to emulate its power consumption [188].

In all our tests, we employ the standard TinyOS MAC layer for MICA2 nodes. A single sensor reading is represented by a 32-bit integer value, while the message size at the MAC layer is 52 bytes, possibly containing up to 8 sensor readings. The sources generate one message per minute.

Hereafter, the time period between two successive readings generated from the same source is termed *epoch*. Initially, all nodes are provided with an energy budget equivalent to a pair of commercially available AA batteries. All experiments are repeated 20 times, and the results are averaged.

We first report about experiments in a regular grid, where each node can communicate with its four neighbors. This choice simplifies the interpretation of results by removing the bias induced by random deployments, while also well modeling some of the settings we target, e.g., indoor WSN deployments for control and monitoring [27]. To this end, the nodes are placed 25 meters apart with a communication range of 40 meters. Moreover, we also evaluated the performance of our protocol in deployments with a random topology, characterized by a pre-specified average number of neighbors for each node. With respect to the grid deployment, these scenarios allow us to assess the impact of the connectivity degree on our results, as well as to evaluate our protocol in more unstructured scenarios (e.g., modeling outdoor WSNs deployments).

The initial tree is built by flooding the system with a "tree construction" message sent by every sink. Each node keeps track of messages received from the same sink, and stores the identifier of the neighbor along which the message was received with the least number of traversed hops. This way, the initial tree is built by minimizing the length of the path connecting each source to every sink. This base tree is also used, without any additional modification, as point of comparison against our solution. Indeed, it essentially provides a baseline, representative of protocols that build independent trees (e.g., Directed Diffusion [37]), against which we show the benefits of our adaptive scheme. The same "tree construction" message is periodically re-sent to possibly update a node's distance from the sink.

For what concerns the protocol parameters, we verified experimentally that the combined contribution of α_1 and α_2 yields the best results in minimizing the number of nodes involved in routing. Therefore, we set $\langle \alpha_1 = 1, \alpha_2 = 1 \rangle$. Nonetheless, a more thorough investigation of how the value of these parameters affect the routing performance is in our immediate research agenda. As for the distance from sinks, we discussed in Section 10.4.2 how its contribution is key in achieving a good ratio of delivered readings. Differently from the two quantities above, the lower is this value, the better (closer) is the neighbor located w.r.t. a given sink. For this reason, we always set $\delta = -2$ throughout all the test runs, so that neighbors closer to the considered sink are preferred over neighbors farther

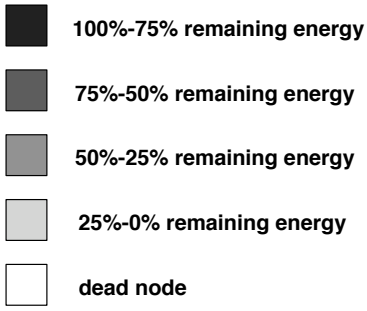
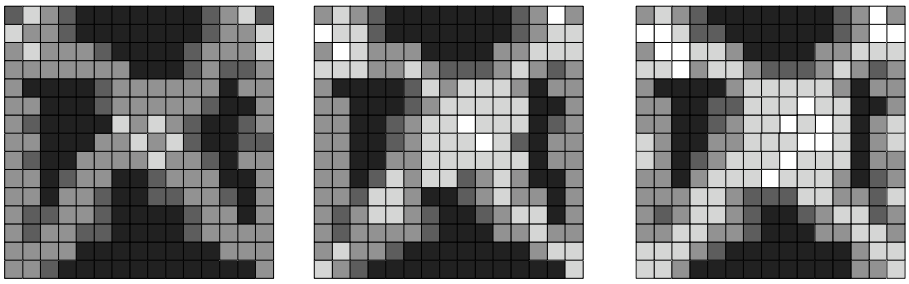


Figure 10.10.: Classes of nodes depending on remaining energy.



(a) Snapshot after 1000 epochs.

(b) Snapshot after 1250 epochs.

(c) Epoch 1339: one of the sink is disconnected from the network.

Figure 10.11.: Energy consumption over time using independent trees (225 nodes in the system).

away. As we verified experimentally, this value provides a good trade-off w.r.t. the other parameters.

Our evaluation is split in two parts. First, we study our protocol in a synthetic scenario to assess whether its behavior matches our design criteria. Next, we quantitatively evaluate its performance in different scenarios and along several metrics.

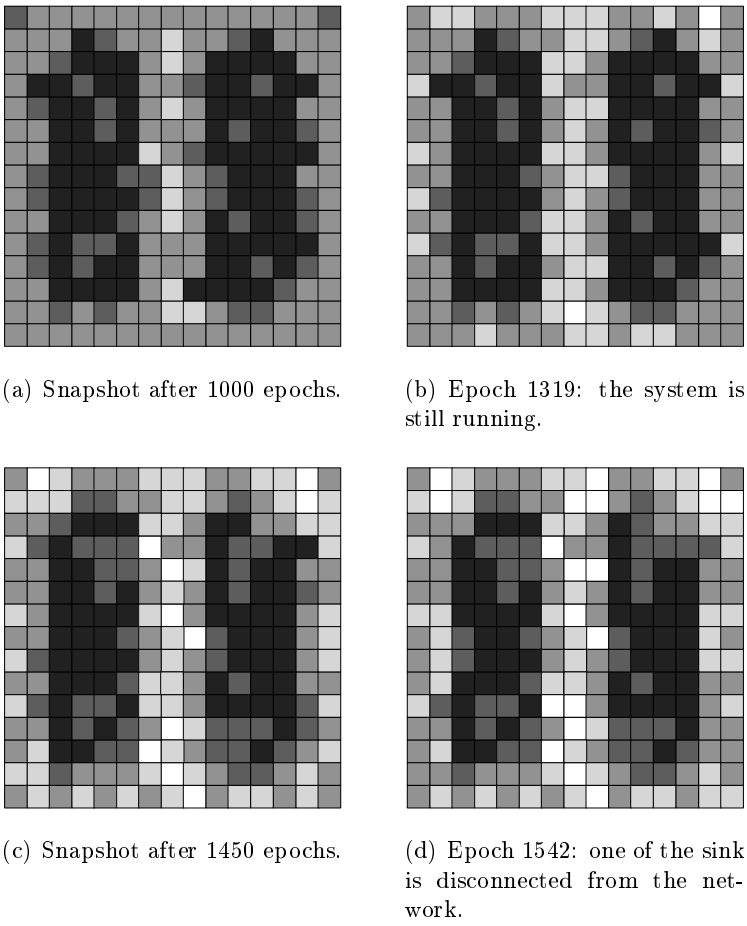
10.5.1. Analyzing the Protocol Behavior

To precisely analyze the behavior of our solution, we run experiments where we keep track of a node's remaining energy over time. In the charts presented hereafter, we distinguish among five classes of nodes depending on their remaining energy, as illustrated in Figure 10.10. To better interpret the results, we used a synthetic scenario with only two sources and two sinks, placed at the opposite corners of a grid topology. This essentially serves to study our protocol behavior as the system evolves. For the same reason, however, the quantitative results obtained here are not indicative of our protocol performance, which is instead evaluated in Section 10.5.2.

Results. Figure 10.11 depicts different snapshots of the system state using the base protocol, i.e., a base tree construction with neither path merging nor load balancing. As the figures illustrate, two source-sink paths cross in the middle of the topology. Nodes in that area are therefore exploited for routing towards two different sinks, and consume energy more rapidly than others. The nodes closest to the two sinks are similarly exploited, as they lie where the two paths leading to the same sink converge. Consequently, at some point nodes start failing in the middle of the topology as well as around the two sinks, until one of them is completely disconnected from its sources, as shown in Figure 10.12(c). Although the chart shows the case with 225 nodes in the system, we observed about the same patterns with different system scales.

On the other hand, Figure 10.12 illustrates the behavior of our protocol using path merging, yet still without load balancing. Here, it is evident how maximizing the overlapping of source-sink paths yields the formation of a sort of "backbone" in the middle of the system. Nodes along this merged path consume energy more quickly than others. They are indeed required to send larger messages whenever multiple readings can be packed in the same physical packet, or to serve multiple source-sink pairs when the paths are not merged. The latter situation can temporarily occur even though no nodes have died yet. For instance, if a tree construction message is lost, a node may receive the tree update from different upstream nodes at different times, thus changing its distance from the sink. In turn, this modifies the node's routing quality as perceived by its neighbors. They may therefore decide to change their parent, thus momentarily separating previously merged paths.

Interestingly, Figure 10.12 also shows a few nodes outside the backbone



(a) Snapshot after 1000 epochs.

(b) Epoch 1319: the system is still running.

(c) Snapshot after 1450 epochs.

(d) Epoch 1542: one of the sink is disconnected from the network.

Figure 10.12.: Energy consumption over time using our routing solution with no load balancing.

consuming a small amount of energy, and then conserving their remaining power until the end of the experiment. These devices, better highlighted in Figure 10.13, correspond to the nodes used by the base tree, as it can be observed by comparing Figure 10.13 against Figure 10.11(a). Without the load balancing scheme, these nodes are indeed involved in routing only initially, until the adaptation mechanism converges on the merged path.

Despite our protocol insists on the nodes in the middle of the topology,

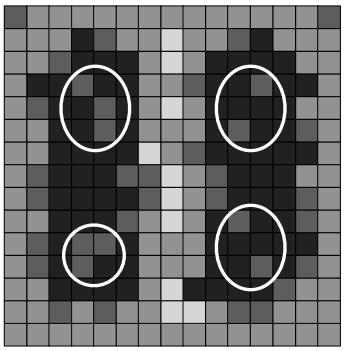
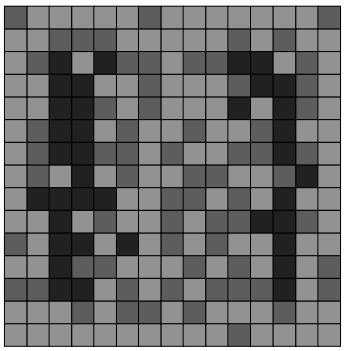


Figure 10.13.: Highlighting the nodes in Figure 10.12(a) that were involved in the initial tree.

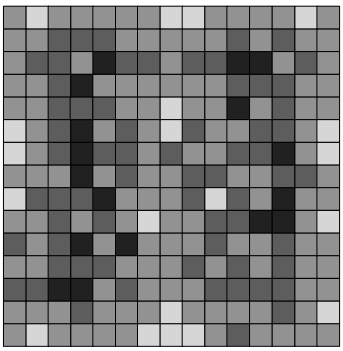
the overall system performance is already improved. Using the base protocol, one of the sinks is disconnected from its sources after 1339 epochs. This happens because of nodes dying close to the sink itself, as in Figure 10.11(c). Even if it may seem our protocol cannot address this issue, by inspecting the logs of our experiments we verified that a partial merging occurs also among the nodes around the sinks as the two paths from the sources come closer. This reduces the number of physical messages these nodes need to send, thus increasing their lifetime. Indeed, using our protocol at the same epoch both sinks are still connected to both sources, as shown in Figure 10.12(b), and only two nodes died throughout the system.

Figure 10.14 depicts the dynamics of our complete protocol, i.e., using adaptation driven by both routing quality and expected lifetime. A comparison of these snapshots against Figure 10.11 and 10.12 provides immediate indications on the effectiveness of our load balancing scheme. The "backbone" effect is much less evident. Moreover, the number of nodes consuming at least 25% or 50% of their available energy remarkably increases over time. This is due to the ability of our solution to distribute the routing effort evenly, hence involving different nodes at different times.

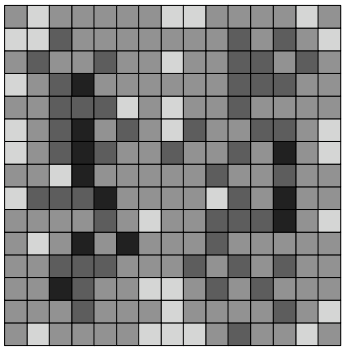
More specifically, as illustrated in Figure 10.14(a), after 1000 epochs no node is yet under 25% of remaining energy. After 1339 epochs, as shown in Figure 10.14(b), the system is still running with no failed nodes, whereas at the same epoch Figure 10.11(c) shows the base protocol already with a sink completely disconnected from its sources. A similar reasoning holds



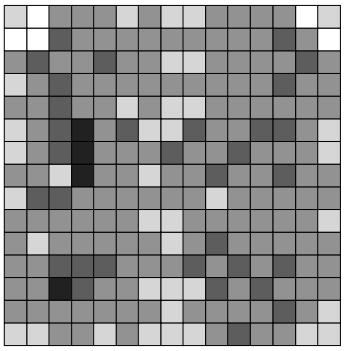
(a) Snapshot after 1000 epochs.



(b) Epoch 1339: the system is still running with no failed nodes.



(c) Snapshot after 1542 epochs.



(d) Epoch 1699: one of the sink is disconnected from the network.

Figure 10.14.: Energy consumption over time using our complete protocol.

between Figure 10.14(c) and Figure 10.12(d). In the latter, our protocol without load balancing yields a partition between one of the sink and the rest of the system. At the time of the first network partition, shown in Figure 10.12(d), almost all nodes spent at least 50% of their available energy. Notably, the partition still occurs because of some nodes dying around one of the sink. However, this happens far later in time, as our scheme is able to merge the paths around the sinks and wisely alternate among these critical nodes. Obtaining a similar behavior was precisely our

chief design objective, which we therefore deem accomplished based on the results presented in this section. The impact of these mechanisms on the overall system performance is quantitatively assessed next.

10.5.2. Performance Characterization

In this section, we report on experiments comparing the performance of our solution against the base mechanism without adaptation as well as the optimal solution identified in Section 10.3. To compute the latter, we used the GLPK [189] solver to determine the ideal topology connecting sources to sinks, given their respective placement in the system and the constraints defined in Section 10.3. These solutions provide the two extremes for our evaluation: we indeed demonstrate that our adaptation strategy provides remarkable benefits w.r.t. mainstream tree-based solutions, and that its effectiveness approaches the theoretical optimum.

As for the modeling of sources and sinks, each scenario is set so that 10% of the nodes are data sources, whereas we vary the number of sinks involved to study how our protocol handles a variable number of source-sink paths. To investigate quantitatively our protocol performance, we measured the following quantities:

- the *ratio of readings delivered* to the sinks over those sent;
- the *system lifetime*, computed by stopping the experiments when the last source-sink paths becomes interrupted. This indeed corresponds to the point in time when the application is no longer able to process any data;
- the *number of nodes exploited*, i.e., the metric we aim at minimizing to obtain more efficient routes connecting sources to sinks.

Results. We first focus on grid topologies. As for the ratio of messages delivered to the sinks, our protocol improves of about 15% w.r.t. the base solution, mostly irrespective of whether the load balancing scheme is used. Though a similar result may seem marginal, the system lifetime improves drastically using our approach, as illustrated in Figure 10.15. The path merging mechanism alone increases the system lifetime of about 50%. In combination with the load balancing scheme, our complete scheme achieves improvements around 75%, on average. Similarly to what we observed in Section 10.5.1, we verified that most of the network partitions are due to

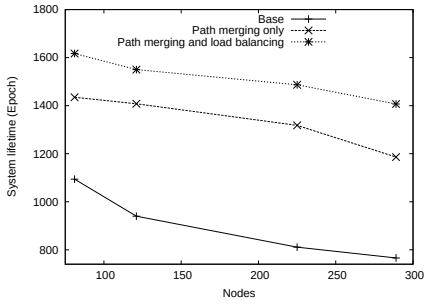
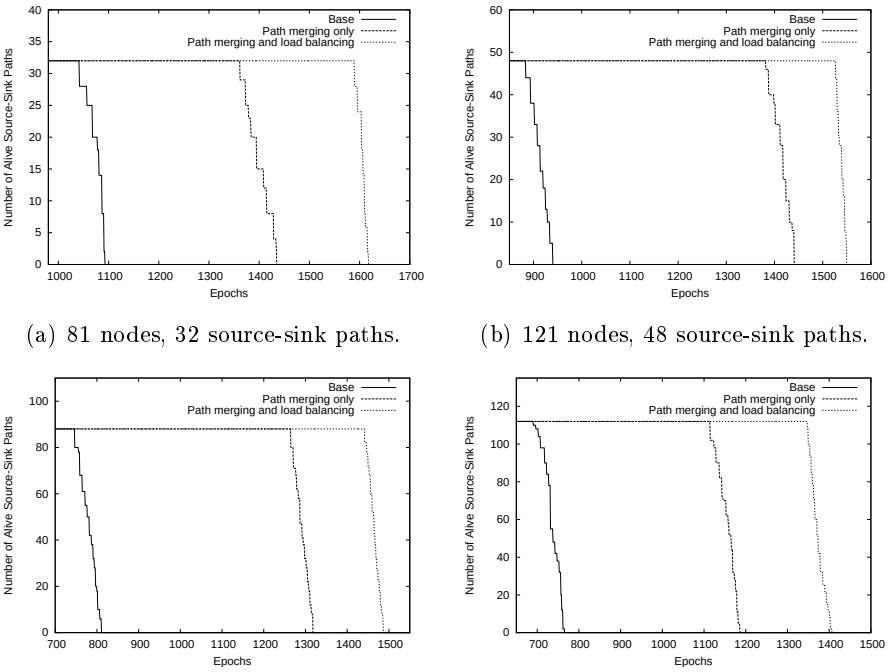


Figure 10.15.: System lifetime vs. number of nodes, in a system with 4 sinks.

nodes around the sinks depleting their battery. Nonetheless, our protocol exploits these nodes much better than the base protocol, thanks to the two complementary mechanism we designed. One on hand, the path merging technique decreases the number of physical messages these nodes need to transmit. On the other hand, the load balancing mechanism achieves a sort of fair load allocation among these nodes, exploiting their resources evenly.

To find out how the system behaves during the additional running time allowed by our solution, Figure 10.16 provides a finer-grained analysis of the system evolution over time. Regardless of which solution is employed and the system scale, the number of active source-sink paths always decreases abruptly, as soon as some nodes around a sink prevent communication towards some part of the system. However, our scheme pushes the moment in time when this occurs much farther. Hence, during the time that our solution allows in addition to the base protocol, the system effectively operates to its full capabilities.

The above results are enabled by the combination of path merging and load balancing. To study the effectiveness of the former, Figure 10.17 reports the number of nodes involved in routing using our adaptive protocol compared to the base solution, before any node fails. In addition, we also report the minimum number of nodes needed to connect sources to sinks computed using the model in Section 10.3, i.e., in a centralized manner and with global knowledge of the system topology. As the chart illustrate, our solution drastically improves over the base solution. Moreover, it always lies within 10% from the theoretical minimum, yet it does *not* require any apriori knowledge of the system topology. These results hold both against



(a) 81 nodes, 32 source-sink paths.

(b) 121 nodes, 48 source-sink paths.

(c) 225 nodes, 88 source-sink paths.

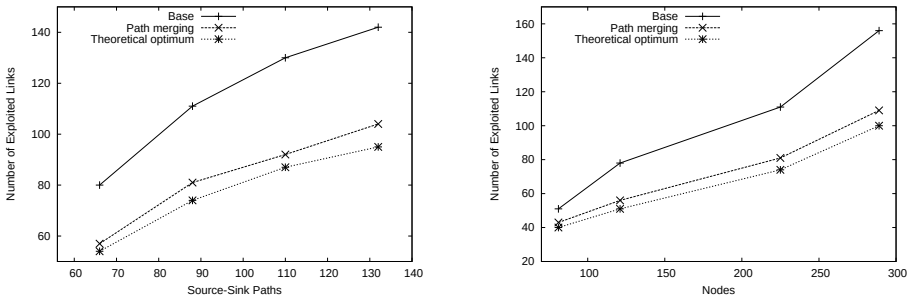
(d) 289 nodes, 112 source-sink paths.

Figure 10.16.: Grid topology: number of active source-sink paths over time vs. nodes (4 sinks).

a variable number of source-sink paths, as highlighted in Figure 10.17(a), and w.r.t. the number of nodes in the system, as Figure 10.17(b) illustrates.

Instead, to assess the contribution of our load balancing scheme we also measured the energy remaining at every node when the experiment stops. Figure 10.18 illustrates this metric against the system scale. Interestingly, using the path merging scheme alone this quantity ends up being almost the same as in the base protocol. Differently, a node's remaining energy using the load balancing scheme is much lower w.r.t. the previous cases. In addition, the variance of the results also decreases, Consequently, we maintain that the contribution brought by this mechanism to the system lifetime comes from spreading the routing load more wisely, so that a higher number of nodes eventually participate in routing.

Next, we repeated the experiments above by generating random topolo-



(a) Nodes involved against number of source-sink paths (225 nodes).

(b) Nodes involved against system size (4 sinks).

Figure 10.17.: Grid topology: nodes involved in routing.

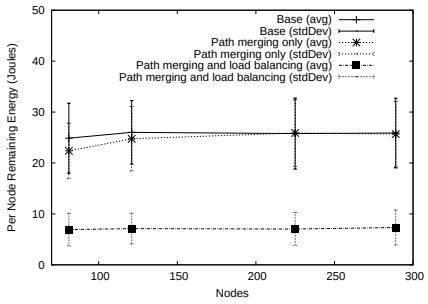


Figure 10.18.: Grid topology: per node remaining energy when simulation stops.

gies with a pre-specified average number of neighbors per node. The results and conclusions we discussed above still hold in these alternative settings, thus assessing the effectiveness of our solution also in more unstructured scenarios. In random topologies, however, we observed again how the system lifetime is ultimately dictated by the running time of nodes around the sinks. To better investigate this aspect, we run an additional set of experiments where the node location is decided semi-randomly. Specifically, we divide the physical space in squared sub-areas whose side is 200 meters long. In each sub-area A, we deploy a set $N(A)$ of nodes, and impose their

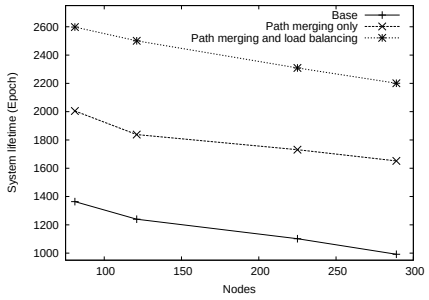


Figure 10.19.: System lifetime vs. number of nodes, in a system with 4 sinks.

density in A be driven by the following formula:

$$\frac{|N(A)|}{A} = \sum_{s \in S} \sum_{n \in N(A)} \frac{K}{dist(n,s)^2} \tag{10.8}$$

where S is the set of sinks used in a given experiment, $dist(n_1, n_2)$ returns the geographical distance between node n_1 and n_2, and K is a constant large enough to yield a connected topology. Intuitively, the above formula forces our topology generator to deploy more nodes around the sinks, while decreasing their density exponentially as nodes are placed farther.

Interestingly, generating the topology as described above amplifies the improvements of our solution w.r.t. the base protocol. In particular, the gains due to the load balancing scheme become larger, as illustrated in Figure 10.19. This evidences once again how the aforementioned scheme wisely exploits the available resources. Indeed, if more nodes are deployed around the sinks, our protocol can push even farther in time the moment where one of the sink becomes disconnected, by simply alternating among more nodes for routing. This occurs with no modification to internal mechanisms of the protocol, that automatically adapts to the particular topology at hand.

10.6. Related Work

The model we presented in Section 10.3 is derived from the large body of literature in operational research and network design. Our choice of the multi-commodity network design problem as a modeling framework is motivated by the generality it allows in pairing sources and sinks. In contrast,

modeling the same problem as a p-source minimum routing cost spanning tree [190] or a Steiner minimal tree [191] would force us to consider *every* node (or source, respectively) to be a sink as well. At the same time, the model we presented here is a simple instance of the multi-commodity network design problem. More sophisticated formulations exist, e.g., taking into account the capacity of network links [192]. In this case, when the capacities along a path are exhausted, alternative, parallel paths are used to share the traffic load, therefore activating more links. However, in WSNs it is difficult to evaluate precisely the actual bandwidth available, due to contention of the wireless medium, collisions and unreliable transmissions [193]. Moreover, these issues are amplified as the number of links used to route messages increase. Therefore, we believe these formulations are not suited for the wireless setting.

For what concerns distributed solutions, it is safe to say that most research in sensor network focuses on optimizing communication from multiple sources to a single sink, as witnessed by the vast amount of literature on the subject [33]. As we already mentioned, these approaches cannot provide efficient solutions to support programming abstractions geared towards sense-and-react scenarios, which inherently call for routing solutions to report to multiple receivers.

In [194] the authors propose mechanisms to build sink-rooted trees incrementally, to perform data aggregation and in-network processing. A path from a single source to the sink is first built, and then shared by other, nearby sources. In this sense, their approach resembles our rational of minimizing the number of nodes exploited to reduce the network traffic. However, their solution is geared to single-sink scenarios, and their results are barely comparable to ours, as they are obtained in simulation using a MAC layer derived from IEEE 802.11. Devising mechanisms to combine the two techniques could provide further benefits, and is a topic worth further investigation.

The work in [179] addresses the problem of routing from a single source to multiple sinks. Common to our approach is the use of broadcast transmissions to let nodes collect information on alternative routes. However, the adaptation in [179] is performed based on *long-range* information (e.g., the number of hops from a node to the different sinks). As this information may not be immediately available, the algorithm starts with a worst case estimation and randomly tries different routes, including those deemed less favorable. When the information gathered during this exploration phase is

not modified for a given number of iterations, the algorithm switches to a stable phase where the discovered routes are used. The adaptation mechanism we proposed in this work is instead based mainly on *local* information that is immediately available (e.g., the number of source-sink paths passing through a node), Moreover, our algorithm is basically self-stabilizing, and does not require distinct phases of operation.

Some researchers addressed the problem of routing from multiple sensors to mobile sinks, focusing on mechanisms to deal with frequent location updates. To this end, in [195] a two-level grid structure is proactively built by the sources. This identifies a reduced subset of nodes responsible for storing information about the sink position, and to which location updates are sent. Conversely, in [196] a stationary sensor node builds a tree on behalf of one or more mobile sinks. These remain linked to this node until they move too far away, at which point they are forced to select a different stationary node. In-network data processing in the presence of mobile sinks is also considered in [178], where a tree is built by a master sink and then shared by slave sinks. Local repair strategies are employed to adjust the tree according to sink mobility. Differently from our approach, in these works sink mobility is the distinctive feature of the target scenario, and the proposed solutions are aimed at reducing the overhead due to it. In contrast, we concentrate on optimizing the source-sink paths, as this is key to improve the system lifetime in our target scenarios, actually less dynamic. In doing so we make only minimal assumptions about the node capabilities (i.e., the ability to overhear messages sent by neighbors), while all the aforementioned proposals require nodes to be aware of their geographical position, exploited for routing.

Instead, the work in [180] introduces an algorithm targeting monitoring applications for achieving energy-efficient routing to multiple sinks. The optimizations proposed are centered around the ability to adjust the sensing rate at different nodes, eliminating the redundancy in the data gathered while preserving the ability to reconstruct the corresponding phenomenon. Instead, we do not assume the ability to influence the source behaviors. Conversely, common to our approach is the problem formulation based on integer linear programming. The authors then map this formulation to a distributed search algorithm based on subgradient optimization, executed in a decentralized fashion. However, they do not provide any insights on the processing overhead this solution would impose on real, resource constrained nodes. We use instead the model presented in Section 10.3 as a

theoretical bound for careful analysis of a lightweight, distributed solution straightforwardly implementable on WSN devices.

Finally, other works have focused on the opportunity to employ multiple sinks not to meet an application requirement, but as a mechanism to increase the system lifetime. For instance, the work in [197] investigates the design problem related to optimally locating multiple sinks in the sensor field, so as to achieve a pre-specified operational time. In this case, even if multiple sinks are present, these simply act as cluster-heads, with each sensor node reporting to only one of them. Similarly, the proposal in [198] studies the problem of selecting, at each node, one of the many sinks present in the system to minimize the overall energy expenditures. Clearly, this is a different problem w.r.t. ours, where the multiple sinks actually represent different system actors, that need to simultaneously gather sensor data for potentially different tasks.

11. Enabling Scoping in Sensor Network Macroprogramming

As we discussed in Chapter 2, WSN programming solutions providing system-centric computation or global view communication are usually termed as macroprogramming approaches. Compare to mainstream programming frameworks, e.g., nesC [28], these usually provide higher-level abstractions to programmers, thus simplifying the development task. Existing approaches in this field, however, mostly assume homogeneous scenarios where a single, system-wide task is to be accomplished. As a result, they are unable to express interactions limited to specific parts of the system, which are commonly found in sense-and-react applications.

To address this issue, in this chapter we again leverage off Logical Neighborhoods as a building-block to serve higher-level abstractions, similarly to the approach we pursued in Chapter 9. Specifically, here we exploit Logical Neighborhoods to provide a flexible notion of scoping in the context of a sensor network macroprogramming framework. The resulting approach enables the specification of complex interactions among system partitions, thus greatly simplifying the development process. Moreover, this is not detrimental to performance: our approach results reasonably close to an optimal solution computed with global system knowledge, while exhibiting a 70% gain w.r.t. baseline solutions. The work presented here appeared in [12, 13, 31].

11.1. Introduction

Early deployments of wireless sensor networks (WSNs) focused on a single, system-wide goal, and featured fairly simple architectures. For instance, habitat monitoring [199] can be implemented using mostly *homogeneous* nodes, each running the *same* application code. Existing macroprogramming solutions well adapts to similar requirements.

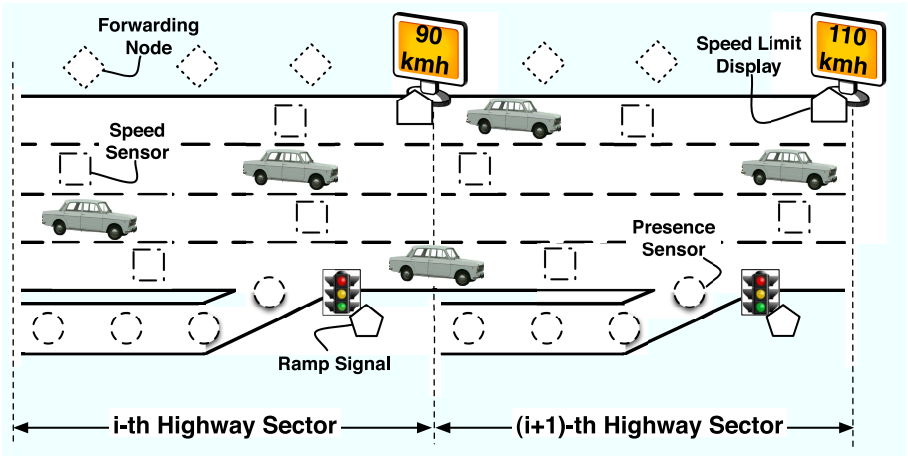


Figure 11.1.: Traffic management scenario.

Nevertheless, the recent advent of more powerful sensor nodes is enabling the use of WSNs in increasingly sophisticated settings, from smart spaces [26] to monitoring and control in buildings [27]. These applications often involve *heterogeneous* nodes equipped with actuators to influence the environment, and their ultimate goal is usually obtained by composing *multiple*, collaborating activities. These characteristics, however, make available approaches ill-suited to develop similar applications.

Reference Scenario. Consider, for instance, a *highway traffic monitoring and control* application, a field where WSNs have gained increasing attention [200]. Various techniques exist to influence the vehicle movements (e.g., to minimize pollution and fuel consumption), that use solutions such as speed signaling and ramp metering [201]. The former aims to control the behavior of traffic by suggesting appropriate speeds, while the latter influences traffic by controlling access to the highway. In these fields, different proposals exist to optimize goals such as pollution and fuel consumption [98]. These systems are often logically divided into disjoint *sectors*, with each sector usually being controlled depending on the current status of the *same* and *neighboring* sectors.

A sample highway scenario is depicted in Figure 11.1, where a sector is identified by a single ramp leading to the highway, i.e., it spans the portion of highway from a ramp to the following. The system has five main components: *i) speed sensors* installed on the highway lanes to measure and

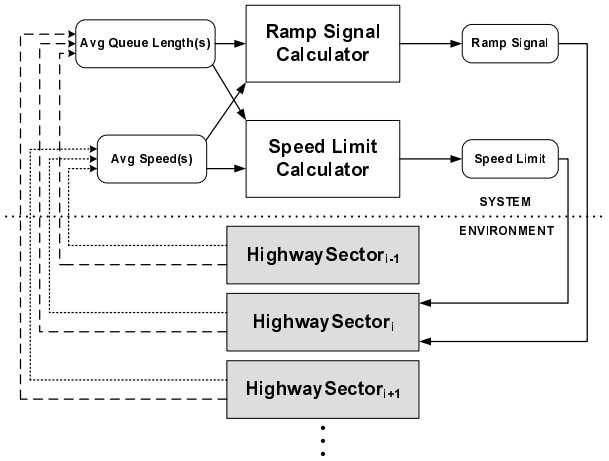


Figure 11.2.: Data processing in traffic management.

report the speeds of vehicles, *ii) presence sensors* installed on the highway ramps to report the presence of vehicles, *iii) speed limit displays* installed one per highway sector to inform of the recommended speed limit, *iv) ramp signals* installed one per highway ramp to allow or disallow cars onto the highway, and *v) forwarding nodes* installed on the road side at regular intervals to enable wireless communication between the various nodes.

Figure 11.2 illustrates, from a high-level perspective, the various stages of data processing in the application. Data is collected from the sensing devices and processed to derive *aggregate measures* —the average speed of vehicles in a highway sector or the average queue length on a ramp. This information is fed as input to an algorithm determining the *actions* to achieve the system objectives, e.g., maximize the flow of vehicles on the highway. These actions are then communicated to the ramp signals and speed limit displays. The specific algorithms employed depend on the goals and metrics of interests.

Need for Scoping. As illustrated in Figure 11.2, multiple concurrent activities must be performed to achieve the overall application goal, in our case, regulating the vehicles' speed and access to the highway. Each of these activities can be decomposed into several, inter-dependent steps where the outputs of one step are fed as input to the following one. Since nodes have different capabilities, each such step must be ultimately mapped to a different system partition that includes only nodes with specific charac-

teristics. As a result, each processing step can be regarded as mapping the inputs obtained from a specific *subset of nodes* to a *different subset of nodes*. Therefore, the programmer must not only identify the different *scopes* based on the application requirements, but, more importantly, express non-trivial interactions among them.

Scoping in Macroprogramming. Most of the existing macroprogramming frameworks provide little or no support for scoping. For instance, in EnviroSuite [102], contexts are defined with conditional statements to create a mapping between software objects and real-world elements, e.g., a moving target. Contexts determine a scope including a set of physically connected nodes with no intermediate hops outside the partition. Albeit sufficient for applications exhibiting spatial locality, such notion cannot be used to address, e.g., a set of geographically sparse actuators, as in our reference application.

TinyDB [17] offers a database interface to WSNs where users submit queries specified with a dialect of SQL. A notion of query scoping is present whereby queries are not delivered to nodes that cannot provide useful data. However, this does not emerge at the programming level, as the span of a query is ultimately dictated by the current sensor readings, and not by application-specified requirements.

The work in [114] targets shared, multi-user sensor networks, and exports a strongly-typed, functional language to express processing. Sensors are named via URI relative to the host they are connected to. Still, programmers are not provided with dedicated constructs to specify interactions among logically-defined system partitions, e.g., to direct a given output from a highway sector to the adjacent ones.

Kairos [32] and Pleiades [15] propose a macroprogramming model inspired by parallel architectures. Developers express the application behavior by writing or reading variables at nodes, iterating on the 1-hop neighbors, and addressing arbitrary nodes. Regiment [16] is a functional macroprogramming language based on the notion of region stream: a spatially distributed, time-varying collection of node states. These are taken as input to functions used to express the application processing. In these cases, no generic construct is provided to express interactions among arbitrary subsets of nodes.

In conclusion, most of the existing approaches target focus on specific classes of applications [102], or do not provide scoping as a first-class programming construct [17]. These characteristics drastically limit their ap-

plicability in the scenarios we target.

Contribution. To address the above issues, in this chapter we leverage off Logical Neighborhoods to empower an existing macroprogramming framework with the ability to express interactions among programmer-defined subsets of nodes. To this end, we make the following contributions:

- **Programming Constructs for Scoping in Macroprogramming.** In Section 11.3 we illustrate language constructs enabling the specification of complex interactions among *application-defined scopes*. The addition of scoping to macroprogramming provides application developers with a *logical* layer on top of the underlying physical system, abstracting away the physical location of data. This greatly simplifies the programming activity, thus speeding up the development process. To illustrate our ideas, we enable scope-based interactions in ATaG [108], a macroprogramming framework. Nonetheless, our techniques can, in principle, be incorporated also in alternative macroprogramming approaches.

- **Compiler and Run-time Support for Scoping.** We demonstrate the *feasibility* of our approach by developing a complete development framework in support of the resulting programming model. In this respect, Section 11.4 illustrates the compilation process used to map the new macroprogramming constructs to the API provided by a dedicated, node-level run-time. Next, Section 11.5 discusses code metrics gathered on the implementation of our reference application, as well as simulation results obtained by running the actual code resulting from the compilation process. Our results show that the ease of programming brought by our approach does not come at the cost of degraded system performance. These present significant improvements w.r.t baseline solutions, and scalability properties similar to optimal solutions computed with global system knowledge.

The next section briefly illustrates the ATaG programming model, providing the foundations needed for the rest of the work. For a comparison between ATaG and alternative macroprogramming models, the reader is referred to Chapter 2.

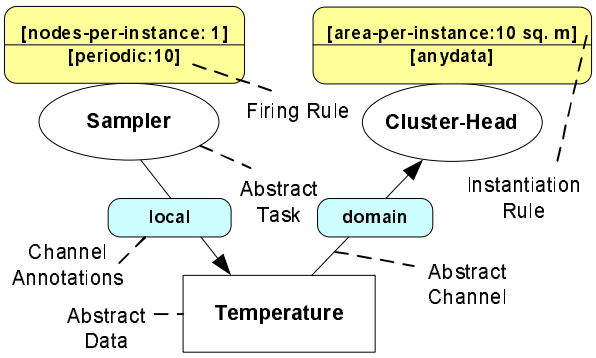


Figure 11.3.: A sample ATaG program.

11.2. The ATaG Programming Model

The Abstract Task Graph [108] (ATaG) is a macroprogramming framework providing a mixed *declarative-imperative* approach. The notions of *abstract task* and *abstract data item* are at the core of ATaG's programming model. A task is a logical entity encapsulating the processing of one or more data items, which represent the information. The flow of information between tasks is defined in terms of their input/output relations. To achieve this, *abstract channels* are used to connect each data item to the tasks that *produce* or *consume* it.

Figure 11.3 illustrates a sample ATaG program specifying a cluster-based, data gathering application. Sensors within a cluster take periodic temperature readings, which are then collected by the corresponding cluster-head. The former behavior is encoded in the *Sampler* task, while the latter is represented by *Cluster-Head*. The *Temperature* data item is connected to both tasks using a channel originating from *Sampler*, and a channel directed to *Cluster-Head*.

Tasks are annotated with *firing* and *instantiation rules*. The former specify when the processing in a task must be triggered. In our example, the *Sampler* task is triggered every 10 seconds according to the `periodic` rule. The *Cluster-Head* fires whenever at least one data item is available on *any* of its incoming channels, in accordance with its `any-data` firing rule. The instantiation rules govern the placement of tasks on real nodes. The `nodes-per-instance:1` construct requires the task to be instantiated once on every node. On the other hand, the `area-per-instance` construct used

for *Cluster-Head* entails partitioning the geographical space according to the given parameter, and deploying *one* instance of the task per partition.

Abstract channels are annotated to express the *interest* of a task in a data item. In our example, the *Sampler* task generates data items of type *Temperature* kept `local` to the node where they have been generated. The *Cluster-Head* uses the `domain` annotation to gather data from the temperature sensors in its cluster, which binds to the system partitioning obtained by `area-per-instance` and connects the tasks running in the same partition.

The code within a task is the only imperative part in an ATaG program. To express data exchange between tasks in the imperative code, programmers are provided with the abstraction of a *shared data pool*, where each task can *output* data, or be *notified* when some data of interest is available. Dedicated APIs are provided for this.

11.3. Scoping in a Macroprogramming Language

In this section, we describe how we bring scoping in macroprogramming by augmenting the ATaG model. We first illustrate how subsets of nodes are specified, and then discuss the novel programming constructs we introduced using an ATaG-based implementation of our reference application as example.

11.3.1. Determining Scopes

Subset of nodes can be determined in several ways. In this work, we take a simplistic yet general approach, and identify the nodes in a given subset as those satisfying a *membership function* $f_s(i)$, where s is a scope and i is a node. The boolean output of f returns whether i belongs to scope s or not. In turn, the actual definition of f_s is obtained as the composition of atomic boolean predicates on the nodes characteristics (called *node attributes* hereafter). As an example, $f_s(i) ::= \mathit{isInSector}(1, i) \wedge \mathit{hasSpeedSensor}(i)$ identifies the subset of nodes equipped with a speed sensor and deployed in the first highway sector.

The boolean predicates are automatically generated by an additional tool we developed that essentially inspects the attributes attached to nodes, and presents a list of predicates to the programmers who only need to compose them in the desired way. With this approach, it is quite natural to determine the desired scopes. In turn, node attributes can be straightforwardly

generated in a variety of means, e.g., from third-party meta-data describing the characteristics of a specific hardware platform [202].

11.3.2. Scoping in ATaG

To enable interactions between scopes, we need to modify primarily two aspects in the ATaG programming model: *task placement* and *data exchange* between tasks. The former express the scopes *where* processing will take place, whereas the latter describe the *interactions* among scopes.

Task Placement. From the application perspective, higher expressivity in task allocation is motivated by the need of mapping a specific processing to nodes equipped with the required sensing/acting devices, or those present in specific regions. For instance, a task designed to operate the ramp signal must be instantiated on a node having that particular device attached. However, we need only one task to compute the average speed for each highway sector, so we need to identify the different sectors uniquely. This has been achieved with revised *instantiation rules*, that give application programmers the ability to map tasks to application-defined subsets of nodes, e.g., all the nodes deployed in the same highway sector.

Data Exchange. Albeit necessary, the above additions do not yet enable the description of interactions between scopes. For instance, in our scenario the speed limit is decided based on the information sensed in three neighboring highway sectors. To achieve this, we should not only identify the speed sensors deployed in three consecutive sectors, but also deliver their sensed data to the nodes where a task computing the speed limit has been instantiated. To achieve this level of expressivity, we define new *channel interests* in ATaG, so that application programmers can specify the task interests by referring to logical properties of data, regardless of their physical location.

11.3.3. ATaG Constructs for Scoping

The syntax and use of the scoping constructs are shown in Figure 11.4, where we illustrate an ATaG implementation of our reference application. All the application information is represented as ATaG data items. The actual algorithm determining the actuation part is encapsulated in two tasks: *SpeedLimitCalculator* and *RampSignalCalculator*, whose inputs are the data produced by tasks deriving the average measures. Once the re-

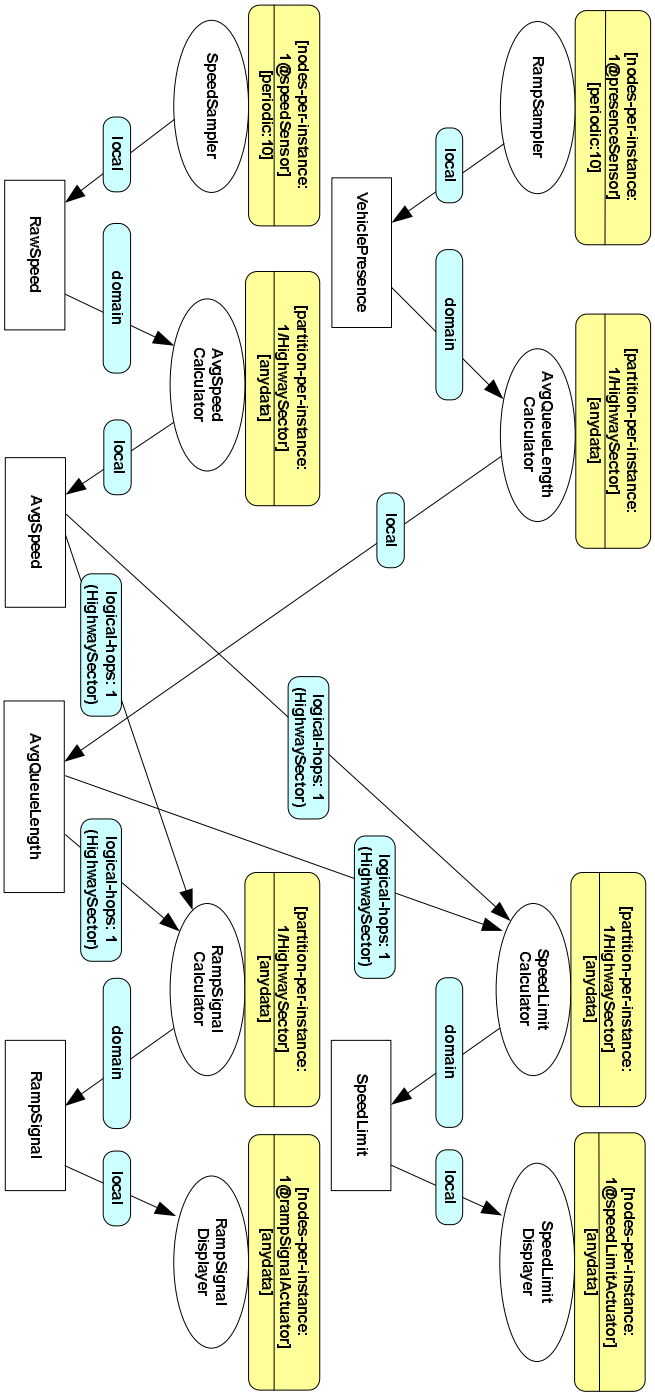


Figure 11.4.: The ATaG program for the traffic management application.

```
<task name="SpeedSampler">
  <instantiationrule>
    <nodes-per-instance
      number="1"
      scopePredicate="hasSpeedSensor"/>
  </instantiationrule>
</task>
```

Figure 11.5.: XML declaration for `@speedSensor` in Figure 11.4.

quired actuation is determined, the corresponding information is given as input to the tasks operating displays and ramp signals. As described next, only three additional constructs are needed to describe the interactions required in our reference application. Still, their *combination* enables the specification of complex communication patterns otherwise hard (or impossible) to describe.

Instantiating Multiple Tasks in a Scope. The *SpeedSampler* task is in charge of gathering the raw data from a speed sensor on a ramp leading to the highway. Therefore, it must run on a node equipped with the corresponding sensing device. To express this requirement, the `nodes-per-instance:1@speedSensor` construct is used, where `@speedSensor` is a placeholder for $f_{speedSensor}(i) ::= hasSpeedSensor(i)$. In our current prototype, this is specified using a simple XML file, shown in Figure 11.5[1]. Similar constructs are used for *RampSampler*, *SpeedLimitDisplayer*, and *RampSignalDisplayer*.

Instantiating a Single Task in a Scope. The *AvgSpeedCalculator* task takes as input the raw data coming from the speed sensors in a sector, and derives the average speed of vehicles in the same sector. Therefore, we need such a task to be instantiated once per sector. To express this, the `partition-per-instance:1/HighwaySector` construct is used. Again, `HighwaySector` is a placeholder for a membership function that identifies all the nodes in a specific sector. The compiler generates all possible values of the corresponding node attribute —that describes the sector where a node is placed in the highway— and requires the task to be instantiated on one node in each sector only.

[1]It is not our intention to force the programmer to write XML directly, we instead envision these specification to be auto-generated by an integrated development environment.

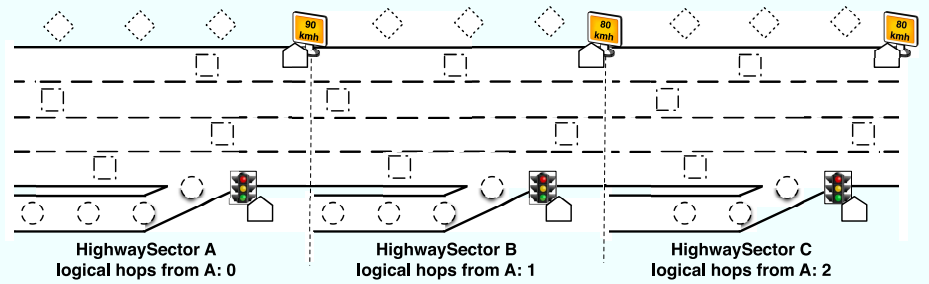


Figure 11.6.: Logical hops over the `HighwaySector` attribute.

Inter-Task Communication. To bind tasks running in the same `HighwaySector`, the `domain` annotation can still be used. However, this time it binds to the system partitioning obtained through the `partition-per-instance` instantiation rule. Differently from `area-per-instance`, this rule determines the different partitions at a logical level, by considering the node attributes instead of the geographical location.

More generally, the construct `logical-hops:1(HighwaySector)` connecting, e.g., the *AvgSpeedCalculator* to both the *SpeedLimitCalculator* and the *RampSignalCalculator* is used to push a data item to a different highway sector. It represents a number of hops counted not on the physical network links, but in terms of how many system partitions (derived from the attribute given in parenthesis) can be crossed. Figure 11.6 illustrates the concept graphically. Given the partitioning induced by the `HighwaySector` attribute, requiring one logical hop on that attribute means, for an *AvgSpeedCalculator* task, to push a data item to the same, immediately preceding and following highway sectors. Note how the semantics of specifying zero hops is not to cross any partition, i.e., to push data to the same partition where the data item originated. In this sense, the `domain` construct actually constitutes a particular case of the more general `logical-hops` construct.

Dynamic Scopes. In this example we define only static scopes, i.e., we use predicates over attributes that do not vary with time. However, the resulting programming model does not prevent, in principle, the definition of scopes involving time-varying properties of the nodes. For instance, one may specify a predicate $isSensingCar(i)$, that holds when a presence sensor is detecting a car nearby. However, it is not clear what would be the semantics of involving such a predicate in, e.g., a task instantiation rule.

Should the task be moved to another node when the condition no longer holds? If not, should the task be suspended and keep the previous state when the condition holds again, or should it just reboot? At a first sight, supporting dynamic scopes may make the programming model unnecessarily complicated, and the final application behavior hard to predict. For this reason, we are currently investigating the application scenarios that may need such a feature, and the semantics required in each case.

11.4. System Support

Our prototype system leverages off the Java2ME [203] language and APIs to describe the imperative part of an ATaG program, and targets the SunSpot sensor platform [77] as underlying hardware platform. Nonetheless, any imperative language can be used instead of Java, as long as it employs a threaded execution model, e.g., the C language on top of the Contiki OS [56].

11.4.1. Compilation

To generate the node-level code from the ATaG specifications, we implemented a dedicated compiler, whose characteristics and performance are illustrated in [31]. The compiler takes as input the ATaG program and information on the attributes attached to the nodes in the final deployment. Compilation starts by deciding the specific node where each task will be running. This is accomplished by looking at the instantiation rules specified in ATaG, and matching them against the node attributes.

When more than one choice for instantiating a task is available, as in the case of `partition-per-instance`, the compiler should place the tasks to minimize some metrics of interests (e.g., network traffic). This problem is orthogonal w.r.t. the support of scoping constructs, since it can be considered as an instance of a graph embedding problem. We are currently working on this aspect as an independent direction of research [31]. Here, instead, we intend to assess the performance of our run-time support to scopes in isolation, without the influence of smart compilation techniques. Therefore, we take a simplistic approach, and assign tasks to nodes randomly when these are not tied to the nodes' capabilities.

After tasks are bound to nodes, the compiler determines the program *data paths*. These are logical addresses identifying the location of tasks

that should actually receive a data item once output by another task. Consider, for instance, the data exchange between *AvgSpeedCalculator* and either *SpeedLimitCalculator* or *RampSignalCalculator* in Figure 11.4. In this case, the data path for an *AvgSpeed* data item includes all the nodes satisfying two specific constraints: i) they are assigned *SpeedLimitCalculator* or *RampSignalCalculator*, and ii) they are deployed either in the same sector where *AvgSpeedCalculator* is running, or in one of the adjacent sectors. Notably, this can still be captured as a scope according to the specification we introduced in Section 11.3.1. Indeed, consider for instance an *AvgSpeedCalculator* task deployed in sector 5. The subset of nodes where the data item should be delivered can be described as:

$$\begin{aligned} f_{AvgSpeed}(i) \quad ::= \quad & (isInSector(4,i) \vee \\ & isInSector(5,j) \vee \\ & isInSector(6,j)) \wedge \\ & (isSpeedLimitCalculator(i) \vee \\ & isRampSignalCalculator(i)) \end{aligned}$$

where the former conjunct refers to an attribute describing where a node has been placed, whereas the latter conjunct predicates over the assignment of tasks to nodes.

Based on the above observation, the compiler looks at the scopes defined in the application, and generates further scope definitions to identify the data paths. Specifically, for each data item, the compiler creates the corresponding data paths by combining the channel annotations between the producer and consumer tasks with the scopes mentioned on the task instantiation rules. These are used either to determine the target system partition (as done for the highway sector in the example), or to identify the receiver node based on the task it is running.

11.4.2. Node-level Run-time

The node-level code output by the ATaG compiler is designed to run atop a supporting run-time hiding the underlying, platform-specific details. Figure 11.7 depicts the architecture of our run-time system [111]. The functionality is divided into a set of modules to facilitate customization to various deployments. support our run-time layer already provides.

The *ATaGManager* stores the declarative portion of the user-specified ATaG program that is relevant to the particular node. This information includes task annotations such as firing rule and I/O dependencies, and the

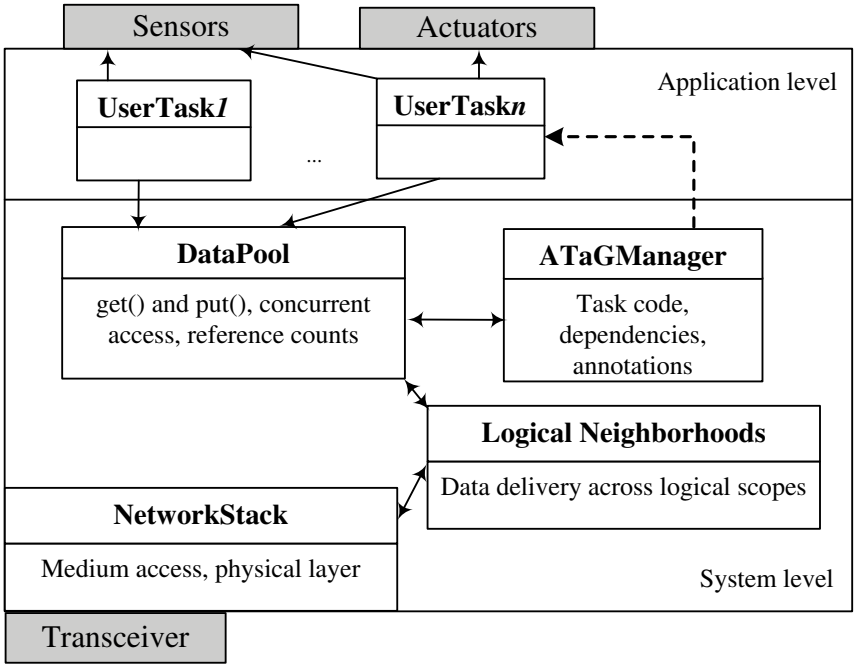


Figure 11.7.: The ATaG node-level run-time.

annotations of input and output channels associated with the data items that are produced or consumed by tasks on the node. The *DataPool* is responsible for managing all instances of abstract data items produced or consumed at the node.

As communication support, we re-used the mechanisms of *Logical Neighborhoods* described in Chapter 6 and 7 to deliver data to the nodes satisfying a given scope specification. In this work, we use the node attributes involved in the definition of at least one data path as logical properties of the nodes, and the data paths themselves as neighborhood definitions[2]. The ATaG node-level run-time leverages off the Logical Neighborhoods communication API, described in Section 6.3, to distribute the data items output by tasks. In particular, the inputs to this module include the data items

[2]The mapping from data paths to neighborhood definitions is straightforward, and omitted here for brevity.

and the scope specifications those are addressed to. Note that only a few modifications were required to embed Logical Neighborhoods as a module within the ATaG run-time, thus assessing the flexibility and generality of the Logical Neighborhoods approach.

Also thanks to the characteristics of the Logical Neighborhood routing scheme, described in Chapter 7, our run-time layer does *not* require the data paths to be evaluated at compile-time. Conversely, every time a data item is output by a task, our run-time re-evaluates the corresponding scope definitions. Interestingly, this readily provides support for dynamic scopes and migrating tasks. Indeed, to support these features, our approach does not require modifications to the scope definitions output by the compiler. For instance, if the node running *SpeedLimitCalculator* changes at run-time, every scope including $isSpeedLimitCalculator(i)$ will simply evaluate to a different subset of nodes the next time a data item is output by *AvgSpeedCalculator*. As already mentioned, however, the aforementioned functionality have deep implications on the language semantics. For instance, what happens if no node is available to accept a task willing to migrate? We are actively studying how to address these issues in the programming model, leveraging off the support our run-time layer already provides.

11.5. Evaluation

One of the issues in devising high-level programming models for WSNs is to provide an acceptable run-time performance. Indeed, the inability to reach the lowest possible levels in the protocol stack may prevent developers from fine-tuning the final running code. In this section, we argue that our approach provides a reasonable trade-off between these two extremes, by first examining the development effort in our reference application, and then reporting on performance results gathered in simulations.

Evaluating the Programming Effort. Quantifying a developer's effort is a challenge per se, because of the lack of widely accepted methodologies and metrics. This is brought to an extreme in sensor network macroprogramming, where most of the existing metrics cannot even be applied given the early stages of the field. However, interesting insights can be gained by looking at the *fraction of code* developers write w.r.t. the entirety of code deployed on the real nodes. This captures the extent to which the application semantics is achieved by either leveraging off the mechanisms

in the node-level run-time, or *automatically* generating code. In this respect, it represents the actual added value of the programming model: the smaller is this fraction, the better the abstractions provided are assisting the programmer, thus speeding up the development process.

With our solution, a total of 51 Java classes need to be compiled to deploy our reference application on the single nodes. However, only 15 of them are the direct result of developers' effort. Furthermore, considering the actual number of lines of non-commented code, only about 12% of them are hand-written by developers, whereas the rest is either part of the run-time support, or automatically generated by our dedicated compiler. We believe these results are due to the flexibility of the scoping abstraction we enabled in the programming model. Complex interactions can indeed be specified in a fully declarative manner, with the compiler taking care of automatically generating the corresponding imperative code and the inputs for the node-level run-time.

Considering the code implementing each task, it is possible to identify a recurring pattern with only two classes needed. One represents the task itself, and contains the processing to interact with the data pool. This same class usually holds a reference to a second class containing the actual processing, e.g., to average the incoming data as in *AvgQueueLengthCalculator*. Note that all the state variables defined in these classes relate only to the application semantics, and never refer to distribution aspects. This is achieved as a result of the way communication patterns are specified in our approach: the data recipients are always determined implicitly by the definition of scopes and the interactions among them. Therefore, the programmer does not need to care about this in the actual application code.

Simulation Settings. To verify that the above advantages do not entail a degraded run-time performance, we quantitatively characterize the behavior of our reference application in a simulated scenario. We use the SWANS/Jist simulator [204], as it is able to run *unmodified* Java code on top of a simulated network. This way, we measure the performance of the same code that can be deployed on the real nodes.

The relevant simulation parameters are reported in Figure 11.8. We consider the scenario in Figure 11.1 as target network, with a highway sector 20 meters wide and 200 meters in length. We place the forwarding nodes 25 meters apart, and randomly distribute the speed sensors on the four lanes so that each of them is range of at least another speed sensor or a

Parameter Name	**Value**
Propagation Model	Two-ray Ground
Radio Model	Additive Noise
MAC Layer	CSMA
Transmission Rate	250 Kbps
Communication Range	40 meters
Message Size	47 bytes
Simulation Time	2000 secs
Number of Repetitions	30

Figure 11.8.: Simulation parameters.

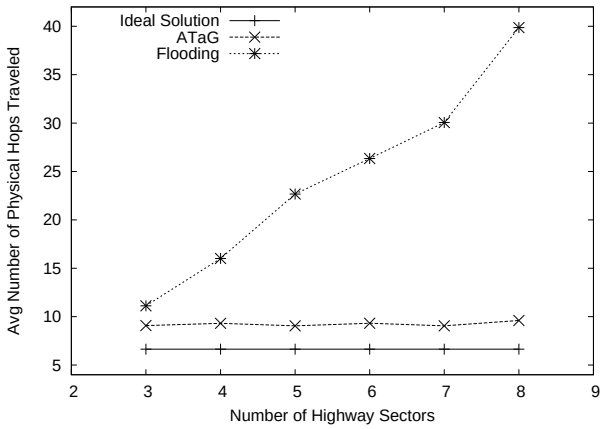
forwarding node. Similarly, the presence sensors are randomly distributed on the ramp so that each of them is in range of at least one speed sensor or another presence sensor. The node controlling the ramp signals and the speed limit displays are placed between different sectors, on opposite sides of the road. Overall, 18 nodes are deployed in each highway sector.

Note that the message rate is implicitly determined by the application itself, in particular by the firing rules for tasks. For instance, a node running an instance of *RampSampler* generates one message every 10 seconds, as its firing rule is `periodic:10`. The *AvgQueueLengthCalculator* fires for any data item received, and correspondingly outputs a new data item. Therefore, if four *RampSampler*s are in its `domain`, *AvgQueueLengthCalculator* generates a message every 2.5 seconds, on the average.

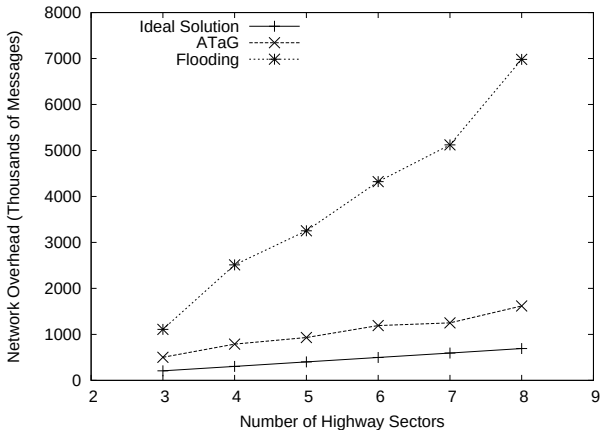
The simulation runs differ in the random seed, the location of nodes, and the assignment of tasks to nodes when the choice is not unique. As performance metrics, we consider *i)* the number of *missing actuations* on the environment, resulting from one or more *message losses* on the path from the sensing tasks to the actuation tasks, *ii)* the *network overhead*, represented as the overall number of messages sent at the physical layer, and *iii)* the average number of *physical hops* traveled by a message carrying a data item before either being discarded or delivered.

As the goal of the application developer is that of deciding actions based on data sensed, the first quantity intuitively measures the *quality of service* provided by the implemented system. Differently, as communication dominates the energy expenditures in WSNs, the second measure assesses the actual feasibility of our approach on real devices. The third measure gives insights into the trends related to communication cost, describing where communication takes place. As independent variable, we choose to vary the number of highway sectors, as this indirectly dictates the system scale.

For comparison, we compute the aforementioned metrics for an *optimal*



(a) Average number of physical hops traveled.



(b) Network overhead.

Figure 11.9.: Reference application performance.

solution minimizing the network overhead, based on global knowledge of the target network. We first identify the optimal task placement given the expected network traffic, and then the minimum cost routing tree connecting a sender to all the intended recipients. The performance obtained with a pure flooding scheme are also reported as an upper bound for further comparison.

Results. Given the message generation rates discussed earlier, our solution can provide at least 96% of the actuations that would be occurring in case there were no message losses. This illustrates how the messages carrying the application data are effectively delivered to the intended recipients. Remarkably, this metric is *not* affected by a varying number of highway sectors (and is hence not shown graphically). Such a behavior demonstrates how our scoping constructs allow the application semantics to *percolate* down to the network layers. Indeed, the application processing spans at most three adjacent highway sectors, and is therefore independent of the overall number of highway sectors.

The chart in Figure 11.9(a) further confirms the above reasoning: as expected, the number of hops traveled by a message using flooding rapidly increases with the number of highway sectors. On the contrary, our solution keeps an almost constant performance in a range of settings, effectively ending up close to the theoretical minimum. Note how it is hard to achieve the same form of implicit cross-layer optimization in the absence of scoping: if the programming model does not allow interactions among application-defined scopes to be defined, it is hard to make the routing layers aware of them.

Figure 11.9(b) depicts the trends in network overhead against a varying number of highway sectors. With our solution, this metric is much closer to the optimal solution than to flooding. More importantly, the trend as the number of highway sectors increases mimics that of the optimal solution, while flooding reveals a much steeper increase. We believe this performance is reasonable, also considering tasks are placed randomly when the decision is not unique. All the metrics are indeed likely to see a dramatic improvement if the compiler placed the tasks smartly using a cost model of the underlying routing scheme. This is in our immediate research agenda.

12. Conclusion and Future Work

Recent advances in WSN technology have made it possible to build systems that not only gather data from the environment, but also affect the physical world by taking actions on it. This way, WSNs can be used as a powerful tool to bridge the physical and virtual worlds, achieving transparent integration of this technology in our everyday life.

Nonetheless, the above scenarios pose difficult challenges to developers, as they sorely miss the appropriate programming abstractions to describe the complex interactions germane to this class of applications. The solutions presented in this thesis aimed at simplifying the programmer's life in these scenarios, while also improving the system performance. Differently from previous work, we pursued this objective by *co-designing* the programming abstractions with the underlying distributed support. This way, we have been able to take advantage of the interplay between the two aspects, obtaining remarkable performance improvements w.r.t. solutions where the two problems are tackled separately.

To reach our ultimate goal, we initially identified the grand challenges that programmers must address in the scenarios we target. Then, in the first part of the thesis we aimed at understanding the characteristics of existing programming solutions, comparing them on a common ground, and identifying where and why they turn out to be ill-suited to developing applications involving actuation. This served to establish the conceptual path we followed in the rest of the thesis, providing a framework in which we cast our contributions. In addition, we believe our taxonomy of WSN programming can be considered as a contribution per se, as it provides a reference point still missing in the field.

In the second part of the thesis, we started addressing some of the aforementioned challenges by proposing solutions targeted to programming individual WSN nodes. Specifically, we first attacked the problem of providing support to programmers for reconfiguring the software running on WSN devices. The component-based programming models we designed address this issue effectively, imposing a very limited performance overhead. Next, we presented the TeenyLime middleware as a solution for providing the

abstractions needed to express coordination among heterogeneous devices. We demonstrated how TeenyLime greatly simplifies the programmers' life, almost without affecting the overall system performance.

The third part of the thesis departed from programming individual nodes, presenting the Logical Neighborhoods abstraction as a fundamental building block to identify and interact with arbitrary groups of nodes. To achieve this, we designed Spidey, a declarative language programmers can use to identify various subsets of nodes, and devised a dedicated routing scheme in support of Logical Neighborhoods. We demonstrated that co-designing the programming abstraction with the underlying distributed protocols enables remarkable performance w.r.t. re-using existing communication schemes.

In the last part of the thesis, we investigated how Logical Neighborhoods can be coupled with different system-level mechanisms and higher-level programming abstractions. As for the former, we integrated a customized version of Logical Neighborhoods with our FiGaRo component model, giving the latter the ability to specify the subsets of nodes involved in a given reconfiguration process. The integration also included a dedicated communication layer, expressly devised in support of code distribution. In this case also, taking into account the characteristics of the programming abstraction in the underlying routing support enabled great performance improvements w.r.t. alternative solutions.

As a natural extension to Logical Neighborhoods, instead, we designed the virtual node abstraction. With virtual nodes, the programmers' life is further simplified, as distribution is masked to a great extent. Once again, we designed a dedicated system support for virtual nodes, also giving our Spidey compiler the ability to customize its internals. This approach resulted in improved performance compared to traditional solutions. In addition, we also realized how our routing layer for virtual sensors enjoys wider applicability beyond our own programming abstractions. Therefore, we developed a stand-alone version to give programmers the ability to leverage off its functionality independently of our programming solutions. Finally, we explored the coupling of Logical Neighborhoods with existing macroprogramming solutions. In this case, embedding Logical Neighborhoods within the run-time layer of ATaG allowed us to sensibly raise the expressiveness of the original language. By virtue of our co-design approach, this did not affect the overall system performance, which remained reasonable compared to alternative mechanisms.

Figure 12.1.: Buonconsiglio castle in Trento (Italy).

Although we do *not* believe the contributions described in this thesis represent a definitive answer to most of the problems we identified, we also have confidence in the advancement that our work represents w.r.t. the state of the art. Therefore, in the short term we aim at further consolidating our solutions by receiving feedback from users exploiting our solutions, and by further tuning and evaluating the performance of our mechanisms, especially in real-world settings. In this latter respect, we are already exploring the use of TeenyLime for monitoring heritage buildings in Trento (Italy), illustrated in Figure 12.1. The system will monitor various phenomena related to the structural integrity of the building, e.g., its deformation over time using dedicated fiber-optic sensors. The rational behind using TeenyLime for this application lies in the need for remotely tasking the system, posed by the structural engineers we are currently collaborating with. Using TeenyLime, a similar functionality is easily implemented, as the single sensing tasks can be represented by tuples that are remotely injected in the system. The single nodes react to the presence of these tuples by starting the corresponding sensing task. The tuples are later removed when the task is accomplished.

In the long term, however, we foresee that programmers will increasingly

require stronger semantics for their distributed applications, as opposed to the best-effort operations that almost all of the existing approaches currently provide. At that point, the next challenge to address will become how to define the semantics required, and how to implement it on resource constrained devices. Many interesting research questions will therefore arise. Do we need novel semantics definitions, or can we re-apply the large body of knowledge researchers have been using in traditional distributed computing? How does this impact the complexity of the abstractions and of the underlying mechanisms? We firmly believe these issues are likely to play a pivotal role in the future of WSN programming.

Similarly, we can also expect that some high-level programming framework (or a combination thereof) will eventually dominate over the others, and research in WSN programming abstractions will come to a stable stage. Whenever this will occur, the main issue will easily become how to verify the correctness of the resulting implementations, both statically and at runtime. In this case also, we will have to address difficult challenges. How can programmer specify what is a correct execution? How can we monitor the behavior of a highly distributed system to check its correctness without interfering in its operation? These questions are currently awaiting an answer we shall all strive to provide.

Bibliography

[1] I. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "A survey on sensor networks," *IEEE Communication Mag.*, vol. 40, no. 8, 2002.

[2] I. F. Akyildiz and I. H. Kasimoglu, "Wireless sensor and actor networks: Research challenges," *Ad Hoc Networks Journal*, vol. 2, no. 4, 2004.

[3] OnWorld - Emerging Wireless Research, www.onworld.com.

[4] P. Costa, G. Coulson, R. Gold, M. Lad, C. Mascolo, L. Mottola, G. P. Picco, T. Sivaharan, N. Weerasinghe, and S. Zachariadis, "The RUNES middleware for networked embedded systems and its application in a disaster management scenario," in *Proc. of the* 5^{th} *Int. Conf. on Pervasive Communications (PERCOM)*, 2007.

[5] P. Costa, G. Coulson, C. Mascolo, L. Mottola, G. P. Picco, and S. Zachariadis, "A reconfigurable component-based middleware for networked embedded systems," *Int. Journal of Wireless Information Networks*, vol. 14, no. 2, 2007.

[6] L. Mottola, G. P. Picco, and A. Amjad, "Fine-grained software reconfiguration in wireless sensor networks," in *Proc. of* 5^{th} *European Conf. on Wireless Sensor Networks (EWSN)*, 2008.

[7] P. Costa, L. Mottola, A. L. Murphy, and G. P. Picco, "TeenyLime: Transiently shared tuple space middleware for wireless sensor networks," in *Proc. of the* 1^{st} *Int. Wkshp. on Middleware for Sensor Networks (MidSens)*, 2006.

[8] ——, "Programming wireless sensor networks with the TeenyLime middleware," in *Proc. of the* 8^{th} *ACM/USENIX Int. Middleware Conf.*, 2007.

[9] L. Mottola and G. P. Picco, "Programming wireless sensor networks with Logical Neighborhoods," in *Proc. of the* 1^{st} *Int. Conf. on Integrated Internet Ad hoc and Sensor Networks (InterSense)*, 2006.

[10] ——, "Logical Neighborhoods: A programming abstraction for wireless sensor networks," in *Proc. of the* 2^{nd} *Int. Conf. on Distributed Computing on Sensor Systems (DCOSS)*, 2006.

[11] P. Ciciriello, L. Mottola, and G.P. Picco, "Building virtual sensors and actuator over Logical Neighborhoods," in *Proc. of the* 1^{st} *ACM Int. Wkshp. on Middleware for Sensor Networks (MidSens06 - colocated with ACM/USENIX Middleware)*, 2006.

[12] A. Pathak, L. Mottola, A. Bakshi, V. K. Prasanna, and G. P. Picco, "Expressing sensor network interaction patterns using data-driven macroprogramming," in *Proc. of the* 3^{rd} *Int. Wkshp. on Sensor Networks and Systems for Pervasive Computing (PerSens - colocated with IEEE PERCOM)*, 2007.

[13] L. Mottola, A. Pathak, A. Bakshi, G. P. Picco, and V. K. Prasanna, "Enabling scope-based interactions in sensor network macroprogramming," in *Proc. of the the* 4^{th} *Int. Conf. on Mobile Ad-Hoc and Sensor Systems (MASS)*, 2007.

[14] M. Welsh and G. Mainland, "Programming sensor networks using abstract regions," in *Proc. of* 1^{st} *Symp. on Networked Systems Design and Implementation (NSDI)*, 2004.

[15] N. Kothari, R. Gummadi, T. Millstein, and R. Govindan, "Reliable and efficient programming abstractions for wireless sensor networks," in *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2007.

[16] R. Newton, G. Morrisett, and M. Welsh, "The Regiment macroprogramming system," in *Proc. of the* 6^{th} *Int. Conf. on Information Processing in Sensor Networks (IPSN)*, 2007.

[17] S. Madden, M.J. Franklin, J.M. Hellerstein, and W. Hong, "TinyDB: an acquisitional query processing system for sensor networks," *ACM Trans. Database Syst.*, vol. 30, no. 1, 2005.

[18] C.-L. Fok, G.-C. Roman, and C. Lu, "Rapid development and flexible deployment of adaptive wireless sensor network applications," in *Proc. of the* 25^{th} *Int. Conf. on Distributed Computing Systems (ICDCS)*, 2005.

[19] S. Li, Y. Lin, S.H. Son, J.A. Stankovic, and Y. Wei, "Event detection services using data service middleware in distributed sensor networks," *Telecommunication Systems*, vol. 26, no. 2, 2004.

[20] C. Frank and K. Römer, "Algorithms for generic role assignment in wireless sensor networks," in *Proc. of the* 3^{rd} *ACM Conf. on Embedded Networked Sensor Systems (SENSYS)*, 2005.

[21] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler, "Hood: a neighborhood abstraction for sensor networks," in *Proc. of* 2^{nd} *Int. Conf. on Mobile Systems, Applications, and Services (MOBISYS)*, 2004.

[22] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson, "Wireless sensor networks for habitat monitoring," in *Proc. of the* 1^{st} *ACM Int. Wkshp. on Wireless Sensor Networks and Applications (WSNA)*, 2002.

[23] P. Juang, H. Oki, Y. Wang, M. Martonosi, L. S. Peh, and D. Rubenstein, "Energy-efficient computing for wildlife tracking: Design tradeoffs and early experiences with ZebraNet," *SIGPLAN Not.*, vol. 37, no. 10, 2002.

[24] J. Burrell, T. Brooke, and R. Beckwith, "Vineyard computing: sensor networks in agricultural production," *IEEE Pervasive Computing*, vol. 3, no. 1, 2004.

[25] K. Römer and F. Mattern, "The design space of wireless sensor networks," *IEEE Wireless Communications*, vol. 11, no. 6, 2004.

[26] E. Petriu, N. Georganas, D. Petriu, D. Makrakis, and V. Groza, "Sensor-based information appliances," *IEEE Instrumentation and Measurement Mag.*, vol. 3, 2000.

[27] A. Deshpande, C. Guestrin, and S. Madden, "Resource-aware wireless sensor-actuator networks," *IEEE Data Engineering*, vol. 28, no. 1, 2005.

[28] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC language: A holistic approach to networked embedded systems," in *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2003.

[29] D. Gelernter, "Generative communication in Linda," *ACM Computing Surveys*, vol. 7, no. 1, 1985.

[30] P. Ciciriello, L. Mottola, and G. P. Picco, "Efficient routing from multiple sources to multiple sinks in wireless sensor networks," in *Proc. of* 4^{th} *European Conf. on Wireless Sensor Networks (EWSN)*, 2007.

[31] A. Pathak, L. Mottola, A. Bakshi, G. P. Picco, and V. K. Prasanna, "A compilation framework for macroprogramming networked sensors," in *Proc. of the the* 3^{rd} *Int. Conf. on Distributed Computing on Sensor Systems (DCOSS)*, 2007.

[32] R. Gummadi, O. Gnawali, and R. Govindan, "Macro-programming wireless sensor networks using Kairos," in *Proc. of the* 1^{st} *Int. Conf. on Distributed Computing in Sensor Systems (DCOSS)*, 2005.

[33] J. Al-Karaki and A. E. Kamal, "Routing techiniques in wireless sensor networks: A survey," *IEEE Wireless Communications*, vol. 11, no. 6, 2004.

[34] K. Langendoen and N. Reijers, "Distributed localization in wireless sensor networks: A quantitative comparison," *Computer Networks*, vol. 43, no. 4, 2003.

[35] J. Elson and K. Roemer, "Wireless sensor networks: A new regime for time synchronization," *SIGCOMM Comput. Commun. Rev.*, vol. 33, no. 1, 2003.

[36] B. Sundararaman, U. Buy, and A. D. Kshemkalyani, "Clock synchronization for wireless sensor networks: A survey," *Ad Hoc Networks*, vol. 3, no. 3, 2005.

[37] C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heidemann, and F. Silva, "Directed Diffusion for wireless sensor networking," *IEEE/ACM Trans. Networking*, vol. 11, no. 1, 2003.

[38] S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S. Shenker, "GHT: A geographic hash table for data-centric storage," in *Proc. of the* 1*st Int. Workshp. on Wireless Sensor Networks and Applications (WSNA)*, 2002.

[39] L. Luo, C. Huand, T. Abdelzaher, and J. Stankovic, "EnviroStore: A cooperative storage system for disconnected operation in sensor networks," in *Proc. of the* 26^{th} *Int. Conf. on Computer Communications (INFOCOM)*, 2007.

[40] T. Liu and M. Martonosi, "Impala: A middleware system for managing autonomic, parallel sensor systems," in *Proc. of the* 9^{th} *SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2003.

[41] P. J. Marrón, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, and K. Rothermel, "FlexCup: A flexible and efficient code update mechanism for sensor networks," in *Proc. of the* 3^{rd} *European Workshop on Wireless Sensor Networks (EWSN)*, 2006.

[42] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," in *ASPLOS-IX: Proc. of the* 9^{th} *Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2000.

[43] N. Reijers and K. Langendoen, "Efficient code distribution in wireless sensor networks," in *Proc. of the* 2^{nd} *Int. Conf. on Wireless Sensor Networks and Applications (WSNA)*, 2003.

[44] J. Koshy and R. Pandey, "Remote incremental linking for energy-efficient reprogramming of sensor networks," in *Proc. of* 2^{rd} *European Workshop on Wireless Sensor Networks (EWSN)*, 2005.

[45] P. Levis, N. Patel, D. Culler, and S. Shenker, "Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks," in *Proc. of the* 1^{st} *Conf. on Networked Systems Design and Implementation (NSDI)*, 2004.

[46] J. W. Hui and D. Culler, "The dynamic behavior of a data dissemination protocol for network programming at scale," in *Proc. of* 2^{th} *Int. Conf. on Embedded Networked Sensor Systems (SENSYS)*, 2004.

[47] S. Kulkarni and L. Wang, "MNP: Multihop network reprogramming service for sensor networks," in *Proc. of the* 25^{th} *Int. Conf. on Distributed Computing Systems (ICDCS)*, 2005.

[48] V. Naik, A. Arora, P. Sinha, and H. Zhang, "Sprinkler: A reliable and energy efficient data dissemination service for wireless embedded devices," in *Proc. of the* 26^{th} *International Real-Time Systems Symposium (RTSS)*, 2005.

[49] P. Levis and D. Culler, "The Firecracker protocol," in *Proc. of the* 11^{th} *ACM SIGOPS European Workshop*, 2004.

[50] ——, "Maté: A tiny virtual machine for sensor networks," in *ASPLOS-X: Proc. of the* 10^{th} *Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2002.

[51] P. Levis, D. Gay, and D. Culler, "Active sensor networks," in *Proc. of the* 2^{nd} *Symposium on Networked Systems Design and Implementation (NSDI)*, 2005.

[52] J. Koshy and R. Pandey, "VM*: synthesizing scalable runtime environments for sensor networks," in *Proc. of* 3^{th} *Int. Conf. on Embedded Networked Sensor Systems (SENSYS)*, 2005.

[53] R. Müller, G. Alonso, and D. Kossmann, "A virtual machine for sensor networks," in *Proc. of the EuroSys Conf.*, 2007.

[54] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White, "Java on the bare metal of wireless sensor devices: the SQUAWK Java virtual machine," in *Proc. of the* 2^{nd} *Int. Conf. on Virtual Execution Environments (VEE)*, 2006.

[55] J. Polastre, J. Hui, P. Levis, J. Zhao, D. Culler, S. Shenker, and I. Stoica, "A unifying link abstraction for wireless sensor networks," in *Proc. of the* 3^{rd} *Int. Conf. on Embedded Networked Sensor Systems (SENSYS)*, 2005.

[56] A. Dunkels, B. Grönvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *Proc. of* 1^{st} *Wkshp. on Embedded Networked Sensors*, 2004.

[57] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava, "A dynamic operating system for sensor nodes," in *Proc. of the* 3^{rd} *Int. Conf. on Mobile systems, applications, and services (MOBISYS)*, 2005.

[58] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, and R. Han, "MANTIS: system support for multimodAl NeTworks of in-situ sensors," in *Proceedings of the* 2^{nd} *Int. Conf. on Wireless Sensor Networks and Applications (WSNA)*, 2003.

[59] H. Cha, S. Choi, I. Jung, H. Kim, H. Shin, J. Yoo, and C. Yoon, "RETOS: Resilient, expandable, and threaded operating system for wireless sensor networks," in *Proc. of the* 6^{th} *Int. Conf. on Information Processing in Sensor Networks (IPSN)*, 2007.

[60] A. Eswaran, A. Rowe, and R. Rajkumar, "Nano-rk: An energy-aware resource-centric rtos for sensor networks," in *Proc. of the* 26^{th} *International Real-Time Systems Symposium (RTSS)*, 2005.

[61] W. P. McCartney and N. Sridhar, "Abstractions for safe concurrent programming in networked embedded systems," in *Proceedings of the* 4^{th} *Int. Conf. on Embedded Networked Sensor Systems (SENSYS)*, 2006.

[62] C. Nitta, R. Pandey, and Y. Ramin, "Y-threads: Supporting concurrency in wireless sensor networks," in *Proc. of the* 2^{nd} *Int. Conf. on Distributed Computing on Sensor Systems (DCOSS)*, 2006.

[63] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: simplifying event-driven programming of memory-constrained embedded systems," in *Proc. of the* 4^{th} *Int. Conf. on Embedded Networked Sensor Systems (SENSYS)*, 2006.

[64] O. Kasten and K. Römer, "Beyond event handlers: Programming wireless sensors with attributed state machines," in *Proc. of the* 4^{th} *Symp. on Information Processing in Sensor Networks (IPSN)*, 2005.

[65] I. Demirkol, C. Ersoy, and F. Alagoz, "MAC protocols for wireless sensor networks: A survey," *IEEE Communications Magazine*, vol. 44, no. 4, 2006.

[66] P. Naik and K. Sivalingam, "A survey of mac protocols for sensor networks," *Wireless sensor networks*, 2004.

[67] W. Ye, J. Heidemann, and D. Estring, "An energy-efficient mac protocol for wireless sensor networks," in *Proc. of the* 21^{st} *Int. Conf. on Computer Communications (INFOCOM)*, 2002.

[68] J. Polastre, J. Hill, and D. Culler, "Versatile low power media access for wireless sensor networks," in *Proc. of the* 2^{nd} *Int. Conf. on Embedded Networked Sensor Systems (SENSYS)*, 2004.

[69] T. van Dam and K. Langendoen, "An adaptive energy-efficient mac protocol for wireless sensor networks," in *Proc. of the* 1^{st} *Conf. on Networked Sensor Systems (SENSYS)*, 2003.

[70] V. Rajendran, K. Obraczka, and J. J. Garcia-Luna-Aceves, "Energy-efficient collision-free medium access control for wireless sensor networks," in *Proc. of the* ^{1st} *Int. Conf. on Embedded Networked Sensor Systems (SENSYS)*, 2003.

[71] ——, "Energy-efficient, collision-free medium access control for wireless sensor networks," *Wirel. Netw.*, vol. 12, no. 1, 2006.

[72] Crossbow Tech., www.xbow.com.

[73] MoteIV, www.moteiv.com.

[74] Body Sensor Network Nodes, vip.doc.ic.ac.uk/bsn/index.php?article=926.

[75] BTNode, www.btnode.ethz.ch.

[76] Eyes WSN Nodes, www.eyes.eu.org.

[77] Project SunSPOT, www.sunspotworld.com.

[78] MeshNetics Tech., www.meshnetics.com.

[79] ScatterWeb Inc., www.scatterweb.com.

[80] Aduino Sensor Node Platform, www.arduino.cc.

[81] P. Baronti, P. Pillai, V. W. C. Chook, S. Chessa, A. Gotta, and Y. F. Hu, "Wireless sensor networks: A survey on the state of the art and the 802.15.4 and zigbee standards," *Comput. Commun.*, vol. 30, no. 7, 2007.

[82] EasySen, www.easysen.com.

[83] D. Hughes, P. Greenwood, G. Blair, G. Coulson, P. Grace, F. Pappenberger, F. Smith, and K. Beven, "An experiment with reflective middleware to support grid-based flood monitoring," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 4, 2007.

[84] IST CRUISE Project, "Flood detection using sensor networks," www.ist-cruise.eu/cruise/business-deck/wsns-applications/flood-detection-1.

[85] A. Arora, P. Dutta, S. Bapat, V. Kulathumani, H. Zhang, V. Naik, V. Mittal, H. Cao, M. Demirbas, M. Gouda, Y. Choi, T. Herman, S. Kulkarni, U. Arumugam, M. Nesterenko, A. Vora, and M. Miyashita, "A line in the sand: A wireless sensor network for target detection, classification, and tracking," *Comput. Networks*, vol. 46, no. 5, 2004.

[86] K. Martinez, J. K. Hart, and R. Ong, "Environmental sensor networks," *Computer*, vol. 37, no. 8, 2004.

[87] P. Padhy, R. K. Dash, K. Martinez, and N. R. Jennings, "A utility-based sensing and communication model for a glacial sensor network," in *Proc. of the* 5^{th} *Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS)*, 2006.

[88] A. Sheth, K. Tejaswi, P. Mehta, C. Parekh, R. Bansal, S. Merchant, T. Singh, U. B. Desai, C. A. Thekkath, and K. Toyama, "Senslide: A sensor network based landslide prediction system," in *Proc. of the* 3^{rd} *Int. Conf. on Embedded Networked Sensor Systems (SENSYS)*, 2005.

[89] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh, "Fidelity and yield in a volcano monitoring sensor network," in *Proc. of* 7^{th} *Symp. on Operating Systems Design and Implementation (OSDI)*, 2006.

[90] J. P. Lynch and K. J. Loh, "A summary review of wireless sensors and sensor networks for structural health monitoring," *Shock and Vibration Digest*, Mar 2006.

[91] G. Wittenburg, K. Terfloth, F. L. Villafuerte, T. Naumowicz, H. Ritter, and J. Schiller, "Fence monitoring - experimental evaluation of a use case for wireless sensor networks," in *Proc. of the* 4^{th} *European Conf. on Wireless Sensor Networks (EWSN)*, 2007.

[92] G. Simon, M. Maróti, A. Lédeczi, G. Balogh, B. Kusy, A. Nádas, G. Pap, J. Sallai, and K. Frampton, "Sensor network-based countersniper system," in *Proc. of the* 2^{nd} *Int. Conf. on Embedded Networked Sensor Systems (SENSYS)*, 2004.

[93] C. Hartung, R. Han, C. Seielstad, and S. Holbrook, "FireWxNet: A multi-tiered portable wireless system for monitoring weather conditions in wildland fire environments," in *Proc. of the* 4^{th} *Int. Conf. on Mobile Systems, Applications and Services (MOBISYS)*, 2006.

[94] K. Lorincz, D. Malan, T. Fulford-Jones, A. Nawoj, A. Clavel, V. Shnayder, G. Mainland, M. Welsh, and S. Moulton, "Sensor networks for emergency response: Challenges and opportunities," *IEEE Pervasive Computing*, vol. 3, no. 4, 2004.

[95] F. Michahelles, P. Matter, A. Schmidt, and B. Schiele, "Applying wearable sensors to avalanche rescue," *Computer and Graphics*, vol. 27, no. 6, 2003.

[96] M. Lampe and M. Strassner, "The potential of RFID for moveable asset management," in *Proc. of the Wkshp. on Ubiquitous Commerce at UbiComp*, 2003.

[97] H. Baldus, K. Klabunde, and G. Müsch, "Reliable set-up of medical body-sensor networks," in *Proc. of* 1^{st} *European Wkshp. on Wireless Sensor Networks (EWSN)*, 2004.

[98] C. Manzie, H. C. Watson, S. K. Halgamuge, and K. Lim, "On the potential for improving fuel economy using a traffic flow sensor network," in *Proc. of the Int. Conf. on Intelligent Sensing and Information Processing*, 2005.

[99] J. A. Stankovic, Q. Cao, T. Doan, L. Fang, Z. He, R. Kiran, S. Lin, S. Son, R. Stoleru, and A. Wood, "Wireless sensor networks for in-home healthcare: Potential and challenges," in *Proc. of High Confidence Medical Device Software and Systems Workshop (HCMDSS)*, 2005.

[100] M. Dermibas, "Wireless sensor networks for monitoring of large public buildings," 2005, Tech. Report, University of Buffalo. Available at www.cse.buffalo.edu/tech-reports/2005-26.pdf.

[101] K. Terfloth, G. Wittenburg, and J. Schiller, "Facts - a rule-based middleware architecture for wireless sensor networks," in *Proc. of the* 1^{st} *Int. Conf. on Communication System Software and Middleware (COMSWARE)*, 2006.

[102] L. Luo, T. F. Abdelzaher, T. He, and J. A. Stankovic, "EnviroSuite: An environmentally immersive programming framework for sensor networks," *Trans. on Embedded Computing Sys.*, vol. 5, no. 3, 2006.

[103] D. Culler, J. Hill, P. Buonadonna, R. Szewczyk, and A. Woo, "A network-centric approach to embedded software for tiny devices," in *Proc. of the* 1^{st} *Int. Wrkshp. on Embedded Software (EMSOFT)*, 2001.

[104] P. Levis *et al.*, "The emergence of networking abstractions and techniques in TinyOS," in *Proc. of* 1^{st} *Symp. on networked system design and implementation (NSDI)*, 2004.

[105] TinyOS Community Forum, "TinyOS TEP 126 - CC2420 radio stack," www.tinyos.net/tinyos-2.x/doc/html/tep126.html.

[106] J. Liu, P. Cheung, F. Zhao, and L. Guibas, "A dual-space approach to tracking and sensor management in wireless sensor networks," in *Proc. of the* 1^{st} *Int. Wrkshp. on Wireless Sensor Networks and Applications (WSNA)*, 2002.

[107] X.-Y. Li, P.-J. Wan, Y. Wang, and O. Frieder, "Sparse power efficient topology for wireless networks," in *Proc. of the* 35^{th} *Annual Hawaii International Conference on System Sciences (HICSS)*, 2002.

[108] A. Bakshi, V. K. Prasanna, J. Reich, and D. Larner, "The Abstract Task Graph: A methodology for architecture-independent programming of networked sensor systems," in *Workshop on End-to-end Sense-and-respond Systems (EESR)*, 2005.

[109] J. Liu, M. Chu, J. Reich, and F. Zhao, "State-centric programming for sensor-actuator network systems," *Pervasive Computing*, vol. 2, no. 4, 2003.

[110] R. Newton and M. Welsh, "Region streams: Functional macroprogramming for sensor networks," in *Proc. of the* 1^{st} *Int. Wkshp. on Data Management for Sensor Networks*, 2004.

[111] A. Bakshi, A. Pathak, and V. K. Prasanna, "System-level support for macroprogramming of networked sensing applications," in *Int. Conf. on Pervasive Systems and Computing (PSC)*, 2005.

[112] R. Newton, Arvind, and M. Welsh, "Building up to macroprogramming: An intermediate language for sensor networks," in *Proceedings of the* 4^{th} *Int. Symp. on Information Processing in Sensor Networks (IPSN)*, 2005.

[113] Y. Yao and J. Gehrke, "The Cougar approach to in-network query processing in sensor networks," *SIGMOD Rec.*, vol. 31, no. 3, 2002.

[114] M. J. Ocean, A. Bestavros, and A. J. Kfoury, "snBench: Programming and virtualization framework for distributed multitasking sensor networks," in *Proc. of the* 2^{nd} *Int. Conf. on Virtual Execution Environments (VEE)*, 2006.

[115] A. Boulis, C.-C. Han, and M. B. Srivastava, "Design and implementation of a framework for efficient and programmable sensor networks," in *Proc. of the* 1^{st} *Int. Conf. on Mobile Systems, Applications and Services (MOBISYS)*, 2003.

[116] A. Boulis, C.-C. Han, R. Shea, and M. B. Srivastava, "Sensorware: Programming sensor networks beyond code update and querying," *Pervasive Mob. Comput.*, vol. 3, no. 4, 2007.

[117] A. Fuggetta, G. P. Picco, and G. Vigna, "Understanding code mobility," *IEEE Trans. Softw. Eng.*, vol. 24, no. 5, 1998.

[118] P. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of Publish/Subscribe," *ACM Computing Surveys*, vol. 2, no. 35, 2003.

[119] S. Kim, S. H. Son, J. A. Stankovic, S. Li, and Y. Choi, "Safe: A data dissemination protocol for periodic updates in sensor networks," in *Proceedings of the Int. Wrkshp. on Data Distribution for Real-time Systems*, 2003.

[120] Embedded WiSeNts Project, "Embedded WiSeNts Research Roadmap," www.embedded-wisents.org/dissemination/roadmap.html.

[121] C. Frank and K. Römer, "Solving generic role assignment exactly," in *Proc. of the* 14^{th} *Int. Wkrshp. on Parallel and Distributed Real-Time Systems (WPDRTS)*, 2006.

[122] R. Gummadi, O. Gnawali, and R. Govindan, "Macro-programming wireless sensor networks using Kairos," in *Proc. of the* 1^{st} *Int. Conf. on Distributed Computing in Sensor Systems (DCOSS)*, 2005.

[123] W. B. Heinzelman, A. L. Murphy, H. S. Carvalho, and M. A. Perillo, "Middleware linking applications and networks," *IEEE Network*, vol. 18, 2004.

[124] P. Levis, N. Lee, M. Welsh, and D. Culler, "TOSSIM: accurate and scalable simulation of entire TinyOS applications," in *Proc. of* 5^{th} *Symp. on Operating Systems Design and Implementation*, 2002.

[125] NS2 simulator, "NS2 Home Page," www.isi.edu/nsnam.

[126] X. Zeng, R. Bagrodia, and M. Gerla, "Glomosim: a library for parallel simulation of large-scale wireless networks," in *Proc. of the* 12^{th} *Wrkshp. on Parallel and Distributed Simulation (PADS)*, 1998.

[127] T. Abdelzaher, B. Blum, Q. Cao, Y. Chen, D. Evans, J. George, S. George, L. Gu, T. He, S. Krishnamurthy, L. Luo, S. Son, J. Stankovic, R. Stoleru, and A. Wood, "EnviroTrack: Towards an environmental computing paradigm for distributed sensor networks," in *Proc. of the* 24^{th} *Int. Conf on Distributed Computing Systems (ICDCS)*, 2004.

[128] G. Mainland, M. Welsh, and G. Morrisett, "Flask: A language for data-driven sensor network programs," Harvard University, Tech. Rep. TR-13-06, 2006.

[129] G. Mainland, L. Kang, S. Lahaie, D. C. Parkes, and M. Welsh, "Using virtual markets to program global behavior in sensor networks," in *Proc. of the* 11^{th} *ACM SIGOPS European Wrkshp.*, 2004.

[130] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TAG: A tiny aggregation service for ad-hoc sensor networks," in *Proc. of* 1^{st} *Int. Conf. on Embedded Networked Sensor Systems (SENSYS)*, 2003.

[131] C.-C. Shen, C. Srisathapornphat, and C. Jaikaeo, "Sensor inforation networking architecture and applications," *IEEE Personal Communications*, vol. 8, no. 4, 2001.

[132] C. Borcea, C. Intanagonwiwat, P. Kang, U. Kremer, and L. Iftode, "Spatial programming using smart messages: Design and implementation," in *Proc. of the* 24^{th} *Int. Conf. on Distributed Computing Systems (ICDCS)*, 2004.

[133] K. Koumpis, L. Hanna, and S. Hailes, "Tunnels of terror," *Computing and Control Engineering*, February 2006.

[134] EU IP FP6 RUNES, "Runes Project Web Site," www.ist-runes.org.

[135] L. Mottola, G.Cugola, and G. Picco, "A self-repairing tree overlay enabling content-based routing in manets," Submitted for publication. Available at www.elet.polimi.it/upload/mottola, Politecnico di Milano, Italy, Tech. Rep., 2006.

[136] G. P. Picco, G. Cugola, and A. Murphy, "Efficient content-based event dispatching in the presence of topological reconfigurations," in *Proc. of the* 23^{rd} *Int. Conf. on Distributed Computing Systems (ICDCS03)*, 2003.

[137] C. Becker, M. Handte, G. Schiele, and K. Rothermel, "PCOM - A component system for pervasive computing," in *Proc. of the* 2^{nd} *Int. Conf. on Pervasive Computing and Communications (PERCOM)*, 2004.

[138] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt, "Run-time dynamic linking for reprogramming wireless sensor networks," in *Proc. of* 4^{th} *Int. Conf. on Embedded Networked Sensor Systems (SENSYS)*, 2006.

[139] H. Cervantes and R. Hall, "Autonomous adaptation to dynamic availability using a service-oriented component model," in *Proc. of the* 26^{th} *Int. Conf. of Software Engineering (ICSE)*, 2004.

[140] The OSGi Alliance, "The OSGi framework," www.osgi.org, 1999.

[141] A. Ferscha, M. Hechinger, R. Mayrhofer, and R.Oberhauser, "A lightweight component model for peer-to-peer applications," in *Proc. of the* 2^{nd} *Int. Wkshp. on Mobile Distributed Computing*, 2004.

[142] M. Roman and N. Islam, "Dynamically programmable and reconfigurable middleware services," in *Proc. of ACM/USENIX Middleware*, 2004.

[143] Y. Weinsberg and I. Ben-Shaul, "A programming model and system support for disconnected-aware applications on resource-constrained devices," in *Proceedings of the* 24^{th} *Int. Conf. on Software Engineering (ICSE)*, 2002.

[144] R. S. Hall, D. Heimbigner, and A. L. Wolf, "A Cooperative Approach to Support Software Deployment Using the Software Dock," in *Proc. of the Int. Conference on Software Engineering (ICSE)*, 1999.

[145] J.-P. Fassino, J.-B. Stefani, J. Lawall, and G. Muller, "THINK: A software framework for component-based operating system kernels," in *2002 USENIX Annual Technical Conference*, 2002.

[146] R. Grimm, T. Anderson, B. Bershad, and D. Wetherall, "A system architecture for pervasive computing," in *Proc. of the* 9^{th} *ACM SIGOPS European Wrkshp.*, 2000.

[147] K. Magoutis, J. Brustoloni, E. Gabber, W. Ng, and A. Silberschatz, "Building appliances out of reusable components using Pebble," in *Proc. of the* 9^{th} *ACM SIGOPS European Wrkshp.*, 2000.

[148] M. Winter, T. Genbler, A. Christoph, O. Nierstrasz, S. Ducasse, R. Wuyts, G. Arevalo, P. Muller, C. Stich, and B. Schonhage, "Components for embedded software: the PECOS approach," in *Proc. Int.*

Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES), 2002.

[149] D. Stewart, R. Volpe, and P. Khosla, "Design of dynamically reconfigurable real-time software using port-based objects," Robotics Institute, Carnegie Mellon University, Tech. Rep. CMU-RI-TR-93-11, July 1993.

[150] H. Hansson, M. Akerholm, I. Crnkovic, and M. Torngren, "SaveCCM – A component model for safety-critical real-time systems," in *IEEE Euromicro Conference*, 2004.

[151] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, "The Koala component model for consumer electronics software," *IEEE Computer*, vol. 33, no. 3, 2000.

[152] P. Grace, G. Coulson, G. Blair, B. Porter, and D. Hughes, "Dynamic reconfiguration in sensor middleware," in *Proc. of Int. Wkshp. on Middleware for Sensor Networks (MidSens)*, 2006.

[153] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar, "Next century challenges: scalable coordination in sensor networks," in *Proc. of the* 5^{th} *Int. Conf. on Mobile computing and networking (MOBICOM)*, 1999.

[154] C. Y. Wan, A. T. Campbell, and L. Krishnamurthy, "Reliable transport for sensor networks: PSFQ—Pump slowly fetch quickly paradigm," *Wireless sensor networks*, 2004.

[155] A. Rowstron, "WCL: A coordination language for geographically distributed agents," *World Wide Web Journal*, vol. 1, no. 3, 1998.

[156] TinyOS Official Source Tree. www.tinyos.net.

[157] B. L. Titzer, D. Lee, and J. Palsberg, "Avrora: Scalable sensor network simulation with precise timing," in *Proc. of the* 4^{th} *Int. Symp. on Information Processing in Sensor Networks (IPSN)*, 2005.

[158] A. L. Murphy, G. P. Picco, and G.-C. Roman, "LIME: A coordination model and middleware supporting mobility of hosts and agents," *ACM Trans. on Software Engineering and Methodology (TOSEM)*, vol. 15, no. 3, 2006.

[159] C. Curino, M. Giani, M. Giorgetta, A. Giusti, A. L. Murphy, and G. P. Picco, "Mobile data collection in sensor networks: The TinyLime middleware," *Elsevier Pervasive and Mobile Computing Journal*, vol. 4, no. 1, 2005.

[160] M. Jonsson, "Supporting context awareness with the context shadow infrastructure," in *Wkshp. on Affordable Wireless Services and Infrastructure*, 2003.

[161] B. Deb, S. Bhatnagar, and B. Nath, "ReInForM: Reliable information forwarding using multiple paths in sensor networks," in *Proc. of the* 28^{th} *IEEE Int. Conf. on Local Computer Networks*, 2003.

[162] L. Mottola and G. P. Picco, "Using Logical Neighborhoods to enable scoping in wireless sensor networks," in *Proc. of the* 3^{rd} *ACM International Middleware Doctoral Symposium (MDS - colocated with ACM/USENIX Middleware)*, 2006.

[163] ——, "Programming wireless sensor networks with Logical Neighborhoods: A road tunnel use case," in *Public Demonstration in Proc. of the the* 5^{th} *Int. Conf. on Sensor Systems (SENSYS) - Best Demo Award*, 2007.

[164] Friendly Neighborhood Spider-Man - Wikipedia, en.wikipedia.org/wiki/Friendly_Neighborhood_Spider-Man.

[165] H. Qi and P.T. Kuruganti, "The development of localized algorithms in wireless sensor networks," *Sensors Journal*, vol. 2, no. 7, 2002.

[166] L.A. Wosley, *Integer Programming*. Wiley, 1998.

[167] R. Stoleru and J. Stankovic, "Probability grid: A location estimation scheme for wireless sensor networks," in *Proc. of the* 1^{st} *Int. Conf. on Sensor and Ad-Hoc Communication and Networks (SECON)*, 2004.

[168] Y. Ni, U. Kremer, A. Stere, and L. Iftode, "Programming ad-hoc networks of mobile and resource-constrained devices," in *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2005.

[169] Q. Wang, Y. Zhu, and L. Cheng, "Reprogramming wireless sensor networks: Challenges and approaches," *IEEE Network*, vol. 20, no. 3, 2006.

[170] G. Frederickson, "Fast algorithms for shortest paths in planar graphs, with applications," *Siam J. Computing*, vol. 16, no. 6, 1987.

[171] F. Stann and J. Hiedemann, "RMST: Reliable data transport in sensor networks," in *Proc. of the* 1^{st} *Int. Wkshp. on Sensor Network Protocols and Applications*, 2003.

[172] P. J. Marrón, A. Lachenmann, D. Minder, J. Hahner, R. Sauter, and K. Rothermel, "TinyCubus: A flexible and adaptive framework sensor networks," in *Proc. of the* 2^{rd} *European Workshop on Wireless Sensor Networks (EWSN)*, 2005.

[173] J. Elson and D. Estrin, "Sensor networks: A bridge to the physical world," *Wireless sensor networks*, 2004.

[174] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh, "Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total," in *Proc. of the* 12^{th} *Int. Conf. on Data Engineering*, 1996.

[175] Z. J. Haas, J. Y. Halpern, and L. Li, "Gossip-based ad hoc routing," *IEEE/ACM Trans. Netw.*, vol. 14, no. 3, 2006.

[176] TinyOS Community Forum, "TinyOS multi-hop routing," www.tinyos.net/tinyos-1.x/doc/multihop/multihop_routing.html.

[177] B. J. Bonfils and P. Bonnet, "Adaptive and decentralized operator placement for in-network query processing," in *Proc. of* 2^{nd} *Int. Wkshp. on Information Processing in Sensor Networks (IPSN)*, 2003.

[178] K. Hwang, J. In, and D. Eom, "Distributed dynamic shared tree for minimum energy data aggregation of multiple mobile sinks in wireless sensor networks," in *Proc. of* 3^{rd} *European Wkshp. on Wireless Sensor Networks (EWSN)*, 2006.

[179] A. Egorova-Förster and A. L. Murphy, "A feedback enhanced learning approach for routing in wireless sensor networks," in *Proc. of the* 4^{th} *Workshop on Mobile Ad-Hoc Networks (WMAN)*, 2007.

[180] K. Yuen, B. Li, and B. Liang, "Distributed data gathering in multi-sink sensor networks with correlated sources," in *Proc. of* 5^{th} *Int. IFIP-TC6 Networking Conf.*, 2006.

[181] K. Römer, "Distributed mining of spatio-temporal event patterns in sensor networks," in *Proc. of the* 1^{st} *Euro-American Wkshp. on Middleware for Sensor Networks (EAWMS)*, 2006.

[182] B. Y. Wu and K.-M. Chao, *Spanning Trees and Optimization Problems*. Chapman & Hall, 2004.

[183] K. Holmberg and J. Hellstrand, "Solving the uncapacitated network design problem by a lagrangean heuristic and branch-and-bound," *Oper. Res.*, vol. 46, no. 2, 1998.

[184] S. Park, A. Savvides, and M. Srivastava, "Battery capacity measurement and analysis using lithium coin cell battery," in *Proc. of the 2001 Int. Symp. on Low Power Electronics and Design (ISPLED)*, 2001.

[185] C. Park, K. Lahiri, and A. Raghunathan, "Battery discharge characteristcs of wireless sensor nodes: An experimentla analysis," in *Proc. of the IEEE Int. Conf. on Sensor and Ad-hoc Communications and Networks (SECON)*, 2005.

[186] iMote2, www.xbow.com/Products/productdetails.aspx?sid=267.

[187] Y. Chou, *Statistical Analysis*. Holt International, 1975.

[188] O. Landsiedel, K. Wehrle, and S. Gotz, "Accurate prediction of power consumption in sensor networks," in *Proc. of the* 2^{nd} *IEEE Wrkshp. on Embedded Networked Sensors (EmNets)*, 2005.

[189] GNU Linear Programming Toolkit, www.gnu.org/software/glpk.

[190] B. Y. Wu, G. Lancia, V. Bafna, K.-M. Chao, R. Ravi, and C. Y. Tang, "A polynomial-time approximation scheme for minimum routing cost spanning trees," *SIAM J. Comput.*, vol. 29, no. 3, 2000.

[191] F. K. Hwang, D. S. Richards, and P. Winter, *The Steiner Tree Problem*. North-Holland, 1992.

[192] B. Gendron, T. G. Crainic, and A. Frangioni, "Multicommodity capacitated network design," *Telecommunications Network Planning*, pp. 1–19, 1998.

[193] G. J. Pottie and W. J. Kaiser, "Wireless integrated network sensors," *Commun. ACM*, vol. 43, no. 5, pp. 51–58, 2000.

[194] C. Intanagonwiwat, D. Estrin, R. Govindan, and J. Heidemann, "Impact of network density on data aggregation in wireless sensor networks," in *Proc. of the* 22^{th} *Int. Conf. on Distributed Computing Systems (ICDCS)*, 2002.

[195] H. Luo, F. Ye, J. Cheng, S. Lu, and L. Zhang, "TTDD: Two-tier data dissemination in large-scale wireless sensor networks," *Wireless Networks*, no. 11, 2005.

[196] H. S. Kim, T. F. Abdelzaher, and W. H. Kwon, "Minimum-energy asynchronous dissemination to mobile sinks in wireless sensor networks," in *Proc. of the* 1^{st} *Int. Conf. on Embedded networked sensor systems (SENSYS)*, 2003.

[197] E. I. Oyman and C. Ersoy, "Multiple sink network design problem in large scale wireless sensor networks," in *Proc. of* 1^{st} *Int. Conf. on Communications (ICC)*, 2004.

[198] A. Das and D. Dutta, "Data acquisition in multiple-sink sensor networks," *Mobile Computing and Communications Review*, vol. 9, no. 3, 2005.

[199] Habitat Monitoring on the Great Duck Island. `www.greatisland.net`.

[200] T. T. Hsieh, "Using sensor networks for highway and traffic applications," *IEEE Potentials*, vol. 23, no. 2, 2004.

[201] P. Kachroo and K. Ozbay, *Feedback Ramp Metering in Intelligent Transportation Systems.* Plenum Pub Corp, 2004.

[202] SensorML, vast.uah.edu/SensorML.

[203] Sun Microsystems, Sun^{TM} Java2 Micro-edition Specification, java.sun.com/javame.

[204] R. Barr, Z. J. Haas, and R. van Renesse, "JiST: An efficient approach to simulation using virtual machines," *Softw. Pract. Exper.*, vol. 35, no. 6, 2005.

Politecnico di Milano
Dipartimento di Elettronica e Informazione
Piazza Leonardo da Vinci 32 I 20133 — Milano