Aus dem Institut für Telematik
der Universität zu Lübeck

Direktor:
Prof. Dr. rer. nat. Stefan Fischer

Comprehensive Development Support for Wireless Sensor Networks

Inauguraldissertation
zur
Erlangung der Doktorwürde
der Universität zu Lübeck
– Aus der Technisch-Naturwissenschaftlichen Fakultät –

Vorgelegt von
Herrn Dipl.-Ing.(FH) Dennis Pfisterer
aus Heidelberg

Lübeck, im Oktober 2007

Erster Berichterstatter:	Prof. Dr. rer. nat. Stefan Fischer
Zweiter Berichterstatter:	Prof. Dr.-Ing. Dirk Timmermann (Universität Rostock)
Dritter Berichterstatter:	Prof. Dr. Hermann De Meer (Universität Passau)
Tag der mündlichen Prüfung:	17. Dezember 2007

Zum Druck genehmigt.

Lübeck, den 17. Dezember 2007

gez. Prof. Dr. rer. nat. Enno Hartmann
– Dekan der Technisch-Naturwissenschaftlichen Fakultät –

Vorwort

Investition in Wissen bringt die höchsten Zinsen. Dieses Sprichwort von Benjamin Franklin trifft besonders auf die vergangenen Jahre zu, die mein Leben in Form von neuem Wissen, Erfahrungen und tollen Menschen unermesslich bereichert haben. Zu aller erst möchte ich dafür meinem Doktorvater Prof. Stefan Fischer ganz herzlich danken. Danke dafür, dass er mir diese Arbeit ermöglicht hat, dass er mir die Freiräume für meine Ideen geschaffen hat und dass er auf seine ganz besondere Art der beste Mentor war, den ich mir wünschen konnte. Für die Übernahme des Koreferats bedanke ich mich ganz herzlich bei Herrn Prof. Dr.-Ing. Dirk Timmermann.

Dass ich diesen Weg überhaupt erst beschreiten konnte, verdanke ich einer ganzen Reihe von Personen, die mich bis zum heutigen Tage immer wieder unterstützt haben. Besonders meinen Eltern möchte ich für ihre immerwährende Unterstützung und die liebevolle Begleitung meines Lebens bedanken – ohne sie wäre ich nicht der Mensch, der ich heute bin. Den Grundstein meiner wissenschaftlichen Laufbahn hat das European Media Lab in Heidelberg gelegt. Für die spannende Zeit danke ich ganz besonders Prof. Siegfried Englert, Dr. Yun Ding, Dr. Ulrich Walther, Prof. Rainer Malaka, Prof. Andreas Reuter, Dr. Klaus Tschira sowie der gesamten Belegschaft des EML und der Klaus Tschira Stiftung. Mein besonderer Dank gilt auch Herrn Prof. v. Hoyningen-Huene und Prof. Körner von der FH Mannheim für die stetige Begleitung und Beratung.

Eine Arbeit wie die vorliegende entsteht nicht auf der grünen Wiese. Vielmehr ist es das Umfeld, das ihr einen Nährboden bietet und sie gedeihen lässt. Für diese Unterstützung danke ich meinen Kollegen am Institut für Betriebssysteme und Rechnerverbund sowie meinen Projektpartnern Prof. Sándor Fekete und Alexander Kröller vom Institut für Mathematische Optimierung der TU Braunschweig. Darüber hinaus vermochten meine Kollegen Carsten Buschmann, Christian Werner, Daniela Krüger, Horst Hellbrück, Birgit Schneider, Martin Lipphardt, Stefan Ransom, Axel Wegener, Dirk-Frank Schmidt, Nils Glombitza und Stephan Pöhlsen des Instituts für Telematik der Universität zu Lübeck, mir tagtäglich ein angenehmes Arbeitsumfeld zu bereiten und meine Arbeit durch viele interessante Gespräche zu bereichern. Besonders hervorheben möchte ich dabei meinen Kollegen Carsten Buschmann, mit dem ich zahllose spannende Diskussionen führte, in denen sich oft vage Ideen zu konkreten Projekten entwickelten.

Mein größtes Dankeschön gilt jedoch denjenigen, mit denen mich eine tiefe Liebe und Freundschaft verbindet: Meinen Eltern und allen meinen Freunden, die mein Leben begleiten und dafür sorgen, dass ich immer mit einem Lächeln durch das Leben gehen kann.

Zusammenfassung

Drahtlose Sensornetzwerke (engl. Wireless Sensor Networks, WSNs) sind heterogene Systeme, die aus kleinen, mit stark begrenzten Ressourcen ausgestatteten Sensorknoten sowie Gateways und Backend-Systemen bestehen. In die Umwelt eingebettet erfassen Sensorknoten physikalische Parameter (wie z.B. Temperatur, Helligkeit oder Bewegung) und leiten diese an Gateways weiter. Gateways senden die erfassten Daten über herkömmliche Netzwerke zu Backend-Systemen, wo sie schließlich weiterverarbeitet und visualisiert werden.

Die Realisierung von Anwendungen für drahtlose Sensornetzwerke ist komplex, da hierbei Herausforderungen verteilter Anwendungen als auch eingebetteter Systeme zu lösen sind. Erschwert wird dies durch die erwähnten heterogenen Komponenten, unvorhersehbare Umwelteinflüsse sowie die Größe der Netze. Um diese Herausforderungen zu meistern verwenden Entwickler typischerweise Simulationen, um Anwendungen in einer kontrollierbaren Umgebung zu testen und zu optimieren, sowie Visualisierungen von Daten, die als Displayersatz für die bildschirmlosen simulierten oder realen Sensorknoten fungieren. Aufbauend auf diese beiden Schritte wird dann die Anwendung auf Sensorknoten, Gateways und Backend-Systeme portiert.

Ziel dieser Arbeit ist es, die Simulation, Visualisierung und Implementierung von heterogenen Sensornetwerkanwendungen durch einen integrierten Entwicklungsprozess zu unterstützen. Dieser Entwicklungsprozess besteht aus den vier Komponenten Shawn, SpyGlass, Fabric und microFibre, die die vormals separaten Schritte durch neue Techniken verbessern und miteinander verbinden.

Dabei sind im Rahmen des SwarmNet Projektes zwei neue Hilfsmittel zur Simulation (Shawn, [49, 92, 127]) und Visualisierung (SpyGlass, [25, 28, 29]) von Sensornetzwerken entstanden. Während existierende, herkömmliche Simulatoren eine detailgetreue Abbildung von Phänomenen der echten Welt ermöglichen, zielt Shawn auf eine Abstraktion von dieser Detailtreue ab. Auf diese Weise unterstützt Shawn die zielgerichtete, schrittweise Entwicklung von Algorithmen und Protokollen für WSNs und erlaubt auch die Betrachtung sehr großer Szenarien mit mehr als 10^6 Sensorknoten. SpyGlass ermöglicht die Visualisierung von Daten, die von einem Sensornetzwerk, einem Simulator oder einer beliebigen anderen Quelle stammen und bildet somit das Fundament auf dem Entwickler durch Plug-Ins applikationsspezifische Visualisierungen erstellen.

Durch die Heterogenität von WSNs, die außer den Sensorknoten auch Gateways, Backend-Systeme, Simulatoren und Visualisierungsumgebungen umfassen, verfügen die einzelnen Komponenten über stark unterschiedliche Ressourcen und abweichende Programmiersprachen und -konzepte. Mangels verfügbarer Lösungen, die diese breite Spanne von Systemen abdecken, ist es gängige Pra-

xis, Anwendungen jeweils inklusive der Netzwerkfunktionalitäten manuell zu implementieren. Diese Mehrfachimplementierung erschwert die konsistente und fehlerfreie Integration von Erweiterungen und führt oftmals zu einer vereinfachten Netzwerkimplementierung, die vorhandene Optimierungspotenziale nicht ausschöpft. Es ist daher erforderlich, diese Komponenten in einen Entwicklungsprozess einzubeziehen und sich wiederholende Aufgaben zu automatisieren.

Um dies zu erreichen, schlagen wir in dieser Arbeit eine neuartige Technik namens Fabric [125, 126, 128] zur Generierung applikations- und datentypspezifischer Middleware vor. Fabric verbindet Simulation, Visualisierung und Applikationsentwicklung für heterogene WSNs, indem ein nahtloser Nachrichtenaustausch zwischen diesen Plattformen ermöglicht wird. Ein Applikationsentwickler beschreibt dazu die Datentypen einer Anwendung als XML Schema Dokument und ergänzt die einzelnen Datentypendefinitionen um Annotationen. Diese bestimmen, wie der jeweilige Datentyp von der generierten Middleware behandelt werden soll; also wie Instanzen eines Datentyps als Nutzlast von Netzwerknachrichten repräsentiert werden und nach welchen Regeln diese verarbeitet werden. So könnten Annotationen beispielsweise angeben, dass Instanzen eines speziellen Datentyps zunächst komprimiert und dann verlässlich übertragen werden sollen.

Die generierte Middleware enthält dann Code, der einen Kompressionsschritt durchführt und das Ergebnis zwischenspeichert, um im Falle einer ausbleibenden Empfangsbestätigung eine wiederholte Übertragung zu veranlassen. Anwendungen, die auf dem generierten Middleware-API basieren, arbeiten dabei ausschließlich mit gewöhnlichen Programmiersprachenkonstrukten, während die Middleware sämtliche Netzwerkfunktionalitäten durchführt. Die eigentliche Generierung einer Middleware leisten dabei Module, die von so genannten Frameworkentwicklern implementiert werden. Ein Modul ist ein hoch spezialisierter Codegenerator und steuert Code für einen bestimmten Aspekt, wie zum Beispiel Kompression oder verlässliche Übertragung, bei. Basierend auf den Eingabedaten des Applikationsentwicklers (den annotierten Datentypen und einer Zielplattformbeschreibung) liefert ein Modul eine Selbstbeschreibung zurück, die Fabric nutzt, um die an der Codegenerierung partizipierenden Module aus den Verfügbaren auszuwählen.

Der Vorteil dieses Ansatzes ist, dass die so generierte Middleware auf die Bedürfnisse der Anwendung und die Fähigkeiten der einzelnen Plattformen zugeschnitten werden kann. Wird dieser Prozess für verschiedene Zielplattformbeschreibungen wiederholt, so entstehen kompatible Middleware-Instanzen für Sensorknoten, Gateways, Backend-Systeme, Simulatoren und Visualisierungsumgebungen. Weiterhin können im Zuge der Generierung Optimierungen vorgenommen werden, die in manuellen Implementierungen meist ausgelassen werden. In dieser Arbeit wird dies exemplarisch an zwei Modulen zur Serialisierung von Datentypen aufgezeigt. Der von diesen Modulen erzeugte Code bildet dabei Instanzen von Datentypen auf die Nutzlast von Netzwerknachrichten ab und umgekehrt. In Sensornetzwerken kommen hierzu gegenwärtig zwei Ansätze zur Anwendung: Zum einen wird das Speicherabbild einer Datentypinstanz direkt

als Nutzlast von Netzwerknachrichten verwendet, zum anderen werden die einzelnen Elemente eines Datentyps manuell nach einem festgelegten Verfahren in die Nutzlast von Netzwerknachrichten kopiert. Der erste Ansatz geht implizit davon aus, dass Nachrichten nur zwischen baugleichen Geräten ausgetauscht werden und folglich die Repräsentationen im Speicher identisch sind. Somit ist ein nahtloser Nachrichtenaustausch mit Gateways und Backend-Systemen nicht problemlos möglich. Der zweite Ansatz vermag zwar die Grenzen zwischen den verschiedenen Plattformen zu überwinden, verlangt jedoch eine manuelle Implementierung dieser Abbildung auf allen heterogenen Zielplattformen. Durch die Verwendung von Fabric ist es möglich, diese Abbildung applikations- und datentypspezifisch zu automatisieren und gleichzeitig zu optimieren.

Das erste der vorgestellten Module generiert dabei Code, der unser neuartiges Serialisierungsverfahren namens microFibre [130] einsetzt, während das zweite Code erzeugt, wie er typischerweise von Hand erstellt wird. Dabei sind die von microFibre erzeugten Nutzlasten von Netwerknachrichten signifikant kürzer als solche, die von manuell implementiertem Code erzeugt werden. Existierenden, automatisierten Ansätzen wie zum Beispiel den Packed Encoding Rules von ASN.1 und Xenia ist microFibre im Hinblick auf die Kompressionsraten ebenbürtig oder gar überlegen, wobei der von microFibre erzeugte Code nur einen geringen Overhead in Form von Codegröße und Laufzeit erzeugt und so den stark begrenzten Ressourcen von Sensorknoten Rechnung trägt. Aufgrund der Tatsache, dass die Funkschnittstelle den Energiekonsum eines Sensorknotens dominiert, führt die Verkürzung von Netzwerknachrichten unter anderem zu einem geringeren Bandbreitenbedarf, einer Energieersparnis und dadurch zu einer verlängerten Laufzeit des Sensornetzwerks.

Jede der vier vorgestellten Komponenten bietet einen in sich abgeschlossenen Dienst für den Entwickler von WSN Anwendungen. Kombiniert man diese Komponenten, so entsteht ein Entwicklungsprozess, der von der Simulation und Visualisierung bis zur Generierung von Middleware für heterogene Zielplattformen reicht und so die Applikationsentwicklung in jedem Schritt mit leistungsfähigen Werkzeugen unterstützt. Um den vorgestellten Entwicklungsprozess auf eine fundierte Basis zu stellen und einem breiten Publikum zugänglich zu machen, wurde Fabric in die Entwicklungsumgebung Eclipse als Plug-In integriert. Diese nahtlose Integration ermöglicht eine Verwendung aller Werkzeuge dieses Entwicklungsprozesses aus einer weithin eingesetzten Entwicklungsumgebung heraus. Um die Anwendbarkeit des vorgestellten Entwicklungsprozesses zu untermauern, präsentieren wir sowohl experimentelle Messungen als auch ein Fallbeispiel, das den Einsatz dieses Entwicklungsprozesses zur Realisierung des WSN-Projektes *MarathonNet* im Detail aufzeigt.

Abstract

Wireless sensor networks (WSNs) are heterogeneous networks that comprise tiny, resource-constraint sensor nodes, gateways and backend systems. Embedded into the environment, sensor nodes measure ambient parameters such as temperature, motion, etc. and gateways provide the integration with traditional networks while backend systems process and visualize received data.

Application development for WSNs is complex as it unites the challenges of distributed applications and embedded programming. In addition, heterogeneity, unpredictable environmental influences and the size of the networks further complicate this situation. To master these issues, developers typically perform multiple, distinct steps: simulations support testing and optimizing applications in a controllable environment and visualizations on remote computers serve as surrogates for the display-less simulated or real sensor nodes. Finally, developers port the application to the sensor nodes, gateways and backend systems.

In this work, we propose a novel development framework that integrates and improves the formerly separate steps of simulation, visualization and application development for heterogeneous WSNs. It consists of the four components Shawn, SpyGlass, Fabric and microFibre. *Shawn* is a novel simulation tool for the design and optimization of applications prior to their real-world deployment. The level of detail provided by traditional tools that mimic the real world as closely as possible comes at a price and it is often not required. Instead, Shawn uses abstract and exchangeable models allowing users to focus on their research goal while operating at orders of magnitudes higher speeds. *SpyGlass* is a modular, extensible and platform-independent visualization environment. Unlike existing tools, it provides the infrastructure for arbitrary visualizations independent from the actual WSN platform. *Fabric* provides the link between simulation, visualization and heterogeneous WSN devices. It is a novel system for the generation of application- and data type-specific middleware for heterogeneous target platforms. Developers provide a data type definition augmented with annotations. Based on this input, multiple *modules* conjointly generate optimized middleware instances. *microFibre*, a new (de-)serialization scheme implemented as such a module, shows how considerable bandwidth savings are achieved without requiring manual optimization.

These four components are contained and valuable in themselves. Combined, they constitute a sound development framework that supports developers at all stages from an initial idea to a final WSN application. We present experimental measurements and a case study showing how this framework facilitated the realization of the real-world WSN project MarathonNet.

Contents

1. Introduction

Embedded systems are constant companions of our daily life. Innumerable of everyday applications rely on these unimposing helpers. Mostly invisible, they are the core of nearly every device ranging from washing machines to game consoles. Advances in technology have fueled the development of a new class of computing devices, so-called wireless sensor nodes. These tiny, low-power and low-cost devices comprise sensors, computational hardware and a wireless communication interface. A – potentially large – number of sensor nodes form a wireless sensor network (WSN) [45, 84]. This chapter briefly sketches the developments that have paved the way for these novel appliances, introduces properties of wireless sensor networks and concludes with a motivation and the scientific contributions of this thesis.

The development of modern computer hardware started in the 1960s with the invention of the integrated circuit (IC). This and the following decade were dominated by huge mainframe computers that filled rooms and were only affordable for a few large companies and data processing centers. The situation radically changed in the 1980s when the personal computer (PC) was introduced that evolved to the most popular computer platform nowadays. The continuous decrease in prize and size continued and notebook computers that made computing non-stationary became widely popular. Since the availability of personal digital assistants (PDAs) in the 1990s, computers are lightweight, portable and small enough to fit into a trouser's pocket.

A similar development could be observed in the networking domain. The first mainframes and home computers were sole islands of computing power. With the advent of local and wide area network technologies, they were released from their isolation and interconnected computers were the enabling technology for the new era of distributed computing. Due to the rise of the Internet and the widespread use of notebooks and PDAs, a global demand for wireless access to the Internet arose. Vastly popular wireless technologies such as GPRS, UMTS and Wireless LAN [69] make the Internet virtually omnipresent.

These two trends (miniaturization and wireless networking) have lead to cheap, tiny and low-power chips offering computational circuits as well as efficient wireless transceivers. Combined, these trends are the enabling technologies for WSNs where computing devices are no longer investment goods but a bulk commodity.

1.1. Wireless Sensor Networks

Wireless sensor networks (WSNs) are heterogeneous networks that comprise tiny, resource-constraint sensor nodes, gateways and backend systems. Embedded into the environment, sensor nodes measure ambient parameters such as temperature, motion, etc. Gateways provide the integration with traditional networks and backend systems process and visualize received data.

Individual sensor nodes are envisioned to be cheap (≈1$), tiny (≈ 1mm^3) and disposable devices with scarce computational, energy and storage resources. Figure 1.1 depicts a typical research prototype of a sensor node, as it is available today. Its major components are a CPU, sensors, I/O-interfaces, energy supply and a wireless transceiver. Form factor and price of the current generation are still an order of magnitude higher than envisioned, yet it is anticipated that even future sensor nodes will not have more resources at their disposal [134].

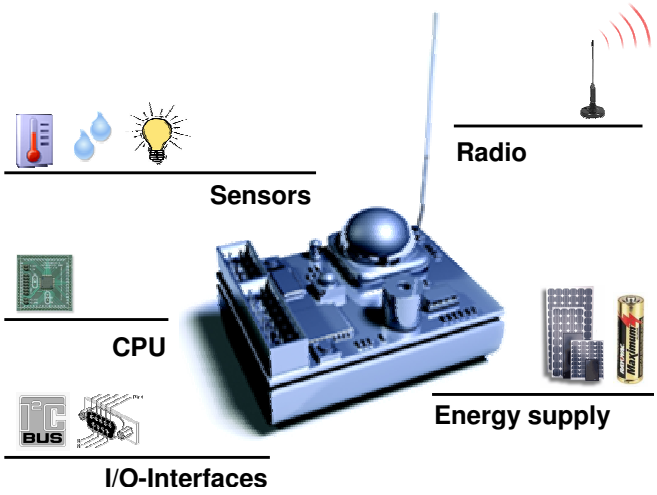


Figure 1.1.: Typical wireless sensor node

Their embedded CPU allows a processing of sensor readings, e.g., to detect and discard outliers in the readings. The wireless interface enables communication with other neighboring nodes. Thus, beyond the local processing of sensed data, the wireless interface supports a cooperative behavior of the nodes. For instance, close by nodes could harmonize their sensor readings before they are routed towards some destination. On the way to the sink, data from several sources can be aggregated to richer information entities (e.g., the average temperature of an area vs. individual temperature readings).

Figure 1.2 shows a typical WSN tracking application. Sensor nodes are randomly scattered over some area of interest where they detect motion using their sensors. Since sensors can deliver erroneous readings, multiple nearby nodes use their wireless interface to determine whether motion was observed by more than one node. If this is the case, they distribute this information augmented with

time and position of the motion detection in the network. Upon the reception of such information from multiple locations and points in time, the sensor nodes can derive motion vectors of objects that crossed the area covered by the WSN. Gateway computers, which are connected to both a sensor node and a traditional network, provide the transition of data from the WSN to backend systems and vice versa. Backend systems then visualize received data, store it in a database or convert it for the use with third party systems.

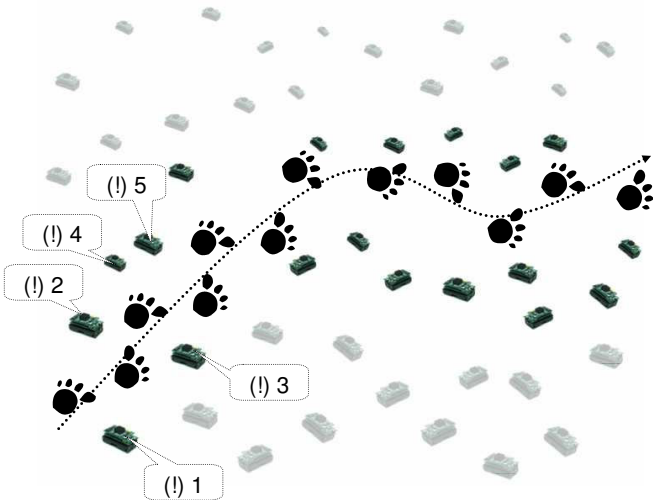


Figure 1.2.: Tracking a moving object with scattered sensor nodes

This *in-network processing* is one example of the innovative properties of WSNs that distinguishes them from traditional gauging applications. Those are designed carefully and sensing devices are installed at predetermined locations with access to infrastructure such as electricity or telecommunication networks. By contrast, WSNs are randomly scattered over potentially hostile or inaccessible terrains (e.g., by dropping them from a plane) and operate for long times without human intervention or infrastructure. After deployment, issues such as changing environmental conditions, battery exhaustion or hardware failure require that WSNs constantly adapt to the prevailing situation.

1.2. Motivation

Application development for WSNs is complex as it unites the challenges of distributed applications and embedded programming. In addition, heterogeneity, unpredictable environmental influences and the size of the networks further complicate this situation. For that reason, most traditional, generic programming techniques are not directly applicable and novel methodologies are required for coping with these challenging conditions. A typical procedure to master these issues is to perform multiple, distinct development steps: Simulations, visualizations and the implementation of the application on sensor nodes,

gateways and backend systems. Based on the state of the art, this section reveals shortcomings of existing solutions and identifies potential for improvements.

Once deployed, sensor nodes must operate unattended without human intervention for a long time. Consequently, software for WSNs must be thoroughly tested prior to real-world deployments. *Computer simulations* are a promising means for the development, implementation and optimization of algorithms and protocols before they are deployed on hardware. Existing simulation tools mimic the real world as closely as possible within a simulation environment. This fine-grained simulation is resource demanding and limits simulations to small scenarios while future scenarios anticipate networks with several thousands of nodes [44, 93]. However, this high level of detail is not required as long as the interest behavior of the system focuses on unaffected properties, e.g., when developing high-level algorithms and protocols. For that reason, novel approaches to the simulation of WSNs that support large-scale scenarios with an abstract point of view are required.

Furthermore, sensor nodes typically do not offer any kind of user interface. To reason on a node's state, a typical means is to connect to its onboard I/O-interfaces. However, determining the state of a deployed WSN by this means is virtually impossible. Hence, approaches to display the state of a network are mandatory and *visualizations* of a WSN's state are therefore a key issue to design and operate these networks. However, visualization is highly application-specific and existing visualization environments are custom-tailored for specific scenarios, hardware platforms, programming languages or a fixed set of visualization primitives. To improve this situation, a generic visualization environment that supports arbitrary visualizations and is independent from the actual hardware platform, programming language and application is required.

A property of WSNs is that they require the processing of data on heterogeneous devices. This requires that all devices are aware of the contents of payload contained in network messages. Traditional *middleware* solutions, which shield developers from this aspect, are not applicable in WSNs due to the inherent resource constraints and different programming paradigms. This often results in a manual implementation of networking code for the different devices (e.g., sensor nodes, gateways and backend systems) and software components (e.g., simulation tools and visualization environments). This is especially unfavorable because changes are an inherent companion of application development where data structures and the application's logic are constantly subject to change. As a result, an *optimization of communication aspects* is frequently omitted because it is time-consuming, laborious and error-prone to keep the implementations for the different platforms consistent manually.

Consequently, integrating simulation, visualization and middleware that supports optimized networking into a common development framework has the potential to improve the development process of WSN applications. However, until today, no widespread use of such tools can be observed. On the contrary, handcrafting WSN applications from scratch is still by far the predominant ap-

proach to WSN application development. This lack of powerful development tools artificially chokes the research progress in WSNs because developers struggle with low-level issues that are typically not in the focus of their research.

1.3. Contributions and Structure of this Work

The contribution of this work is a novel development framework for WSNs, which integrates simulation and visualization as well as the generation of application-specific, optimized middleware for heterogeneous target platforms. This framework is comprised of four components that are introduced in this thesis. On their own merits, these are contained and valuable in themselves. However, when combined into a common framework, they constitute a sound foundation to enhance the overall development process of WSN applications. The following list mentions these components along with their project names and a brief abstract of their functionality:

1. *Shawn*, a high-level and high-performance simulation tool
2. *SpyGlass*, a generic visualization environment
3. *Fabric*, a middleware synthesis framework
4. *microFibre*, a scheme for bit-length optimized payload of network messages

In Chapter 2 the reader is familiarized with essential elements that are used throughout this thesis. First, Section 2.1 provides a detailed overview on wireless sensor networks. Second, Section 2.2 presents concepts, structures and examples of the two important technologies XML and XML Schema. Third, Section 2.3 illustrates fundamental information theoretical aspects.

Chapter 3 introduces our development framework for WSNs. It starts with an overview of the state of the art in WSN application development. Based on an evaluation of this state, it motivates the need for a new kind of development support and introduces criteria for such a framework. It then presents the architecture of our approach and a brief overview of its functionality. Subsequently, Chapter 4, Chapter 5 and Chapter 6 delve into the details of the individual components Shawn, SpyGlass, *Fabric* and *microFibre*.

Chapter 4 presents *Shawn* [49, 92, 127], our novel simulation framework for WSNs with unique features for the development of algorithms, protocols and applications. Shawn does not compete with traditional simulators in the area of network stack simulation. Instead, it focuses on an abstract, repeatable and expressive approach to WSN simulation. By replacing low-level effects with abstract and exchangeable models, the simulation can be used for huge networks in reasonable time while keeping the focus on the actual research problem. The chapter concludes with measurements that compare Shawn's performance with Ns-2 and TOSSIM.

Chapter 5 introduces *SpyGlass* [25, 28, 29], our approach to a modular and extensible visualization environment for wireless sensor networks. Unlike existing tools, SpyGlass supports the visualization of WSNs independent from the actual hardware platform and programming language of the WSN application. We show how users can extend the set of visualization tasks in SpyGlass.

In Chapter 6, we propose *Fabric* [125, 126, 128], a novel middleware-synthesis system for heterogeneous WSNs. Application developers supply a high-level data type description, which is complemented with annotations that parameterize the synthesis process. So-called framework developers contribute their domain specific expertise and complement *Fabric* by implementing modules that back the annotations with code generating functionality. The key benefit of this approach is that it results in lean, custom-tailored code that matches the available resources of the different target devices. While hiding networking aspects and the complexity of distributed systems, it still offers the required flexibility because data handling can be differentiated on a per-type basis. This is demonstrated at the example of two modules that implement different schemes for data type (de-)serialization. They convert in-memory data structures from/to the payload of network messages. Based on the data type descriptions, these modules generate code optimized for either message length or footprint. The first module employs our novel scheme for bit-length optimized data type serialization called *microFibre* [130] while the second one, called *macroFibre*, generates code that resembles traditional methods. This chapter concludes with an evaluation of *microFibre*, *macroFibre* and *Fabric*.

Chapter 7 complements the description of our proposed development framework with a case study of how it has been used to realize a WSN project called *MarathonNet* [63,103,129]. We show how the use of our development framework alleviated the design, implementation and maintenance of this project. The chapter ends with an evaluation of the presented development framework using the criteria introduced in Chapter 3.

Chapter 8 concludes this thesis with a summary and presents directions for future research.

Acknowledgements

The work presented in this thesis would not have been possible without the close cooperation and the fruitful discussions with my colleagues at the Institute of Telematics and the project partners of the SwarmNet [47] project. Shawn is joint work with Alexander Kröller, Prof. Sándor P. Fekete and Prof. Stefan Fischer. The concept of SpyGlass is joint work with Prof. Stefan Fischer and Carsten Buschmann. The realization of the MarathonNet project was conducted together with Prof. Stefan Fischer, Horst Hellbrück, Martin Lipphardt and Stefan Ransom.

2. Fundamentals

This chapter acquaints the reader with fundamental aspects that are used throughout this thesis. It is divided into three parts: Section 2.1 provides an in-depth survey on sensor networks. Section 2.2 introduces XML and XML Schema technologies. Finally, Section 2.3 subsumes important concepts of information theory.

2.1. Wireless Sensor Networks

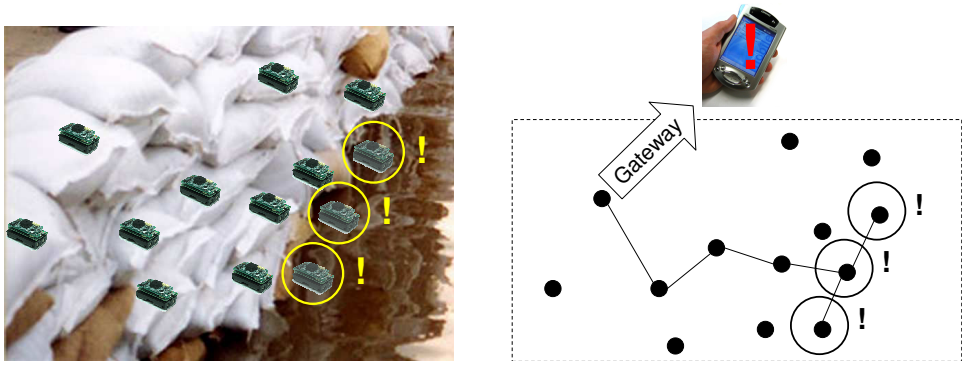
The following sections provide an overview of sensor networks. Section 2.1.1 introduces typical application scenarios, while Section 2.1.2 identifies key challenges that must be dealt with in order to make these visionary scenarios come true and to turn sensor networks into a commercial off-the-shelf product. The remaining subsections revive the topics from Section 2.1.2 and present the state of the art in important research areas. This includes networking paradigms (Section 2.1.3), hardware platforms (Section 2.1.4) as well as operating systems and programming abstractions that address the specific properties of WSNs (Section 2.1.5).

2.1.1. Application Scenarios

WSNs are beneficial in situations where human observation is hardly possible or where wired sensing systems are difficult or expensive to use. This section describes several application scenarios in which wireless sensor networks are in active use as well as application domains in which their future application is promising. Nevertheless, this represents only a small fraction of the possible applications for this new class of networked embedded systems.

Figure 2.1(a) presents a typical application for a sensor network: monitoring a dike made of sandbags for moisture penetration during a flood. The idea is to provide disaster relief teams with real-time data on the status of a dike to avoid breaches due to slowly evolving structural weaknesses. Sensor nodes are incorporated into individual sandbags where they measure the moisture penetration.

Once deployed, the nodes begin to communicate and a spontaneous, ad-hoc network is established. Local sensor readings (e.g., wet or dry) are reconciled with neighboring sensor nodes to prevent faulty readings from being propagated. After this local coordination phase, the final sensor reading is routed to one or



(a) Sensor nodes incorporated in sandbags (b) Data forwarding to a gateway

Figure 2.1.: Example WSN application

more gateway nodes (cp. Figure 2.1(b)). On the route to a sink, readings from multiple sources are further processed. This in-network processing step merges solitary local readings to a condensed data representation before this information is distributed in the network. For instance, instead of disseminating individual, potentially error-prone moisture readings, the presence of a wet area detected by a group of nearby nodes could be transmitted.

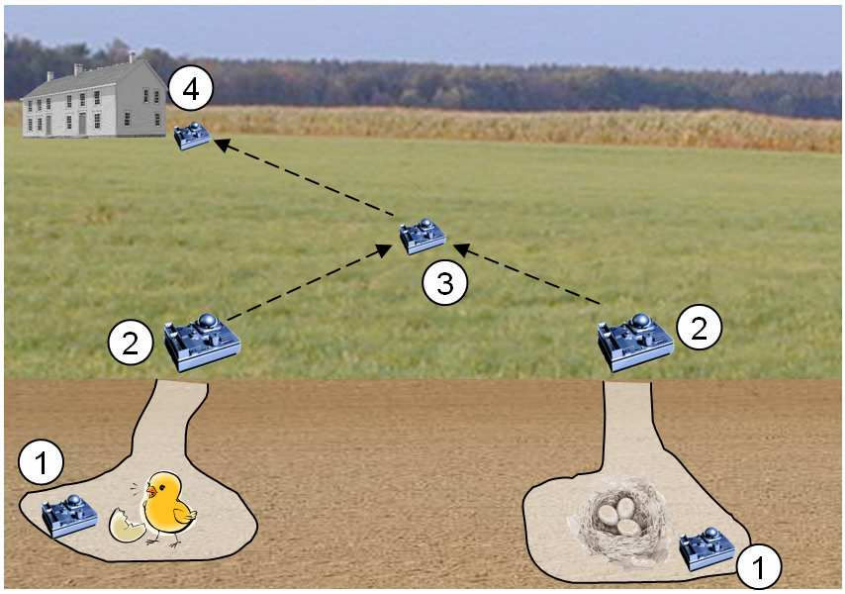


Figure 2.2.: Habitat monitoring on Great Duck Island: (1) Nodes in burrows (2) Nodes outside of burrows (3) Gateway node (4) Research station with server and satellite uplink

Another application domain is habitat monitoring. Sensor networks present a new means for zoologists and biologists to monitor the behavior of animals in real-time without human disturbance. For instance, in 2002 and 2003 re-

searchers deployed a number of sensor nodes in burrows on Great Duck Island where storm petrels were expected to breed [4, 108, 165, 166]. As depicted in Figure 2.2, one set of nodes measured the microclimate (temperature, humidity, barometric pressure and infrared radiation) inside the burrows. Another set of nodes served as weather stations providing information about the conditions outside of the burrows. A multi-hop network formed by the nodes routed the measured data to a central database.

A project called ZebraNet [198] tracks the movement and the interaction of zebras. The animals wear sensor nodes around their neck, which are equipped with GPS [85] receivers and a low-power wireless transceiver. The GPS receiver allows sensor nodes to acquire and store the position of a zebra at a certain point in time. If sensor nodes worn by animals are in the communication range of each other (e.g., while zebras are drinking water at a lake), they exchange data so that each animal stores data about all other animals it has met. Mobile base stations mounted on vehicles receive this information from passing-by animals. Behavioral scientists use this data to study the social behavior of animals over a long time without disturbing the animals in their natural surroundings.

Sensor networks have also been successfully applied in medical and health-care settings. For instance, the MarathonNet project [63, 129] uses a sensor network to monitor runners during marathon events. The runners wear special sensor nodes (so called *pacemates*) that receive heart rate signals from chest transmitter belts and forward this data – if necessary via other devices – to base stations that are deployed along the track. The base stations are connected to a central server (by wide or local area network technologies like WLAN, GPRS or a wired network) where a database stores the received values. Spectators can track the race in real-time, organizers can detect critical heart rates of individual runners and alert rescue personal and runners benefit from a post-facto analysis of their race.

The decreasing price and form factor of current microelectronic circuits allows the integration of a broad range of sensors. For instance, vehicles can be detected by monitoring changes in the magnetic field, nearby beings are noticed by passive infrared receivers and on-board cameras provide a live view of the vicinity of a node. Against this background, promising applications emerge in the telematics, security and military domains. A major strength of sensor networks is to put individual measurements into the context of time and location. For instance, instead of manually counting cars, a sensor network deployed at intersections, roundabouts or highway drives could automatically deduce traffic flows by correlating readings from multiple sensors. Alternatively, intelligent traffic lights could optimize their green phases depending on the current traffic situation. For further application scenarios, please refer to Römer and Mattern's survey paper [142].

2.1.2. Challenges

The envisioned application scenarios presented above implicitly contain a number of challenges that sensor network hardware and software must cope with. Compared to traditional networks, new techniques in several research and engineering domains are required to tackle these challenges [157].

Resource Constraints Individual sensor nodes could be as small as a few cubic millimeters. This tiny form factor imposes strict constraints on the size of sensors, memory, processor and energy supply. Furthermore, the nodes might be located in hostile or inaccessible areas and must operate for a long period, ranging from a few days to several years. In such settings, a manual replacement of energy supplies is not feasible when the nodes are in their operational area. Thus, a sensor node either needs to be supplied with energy prior to its deployment or it must harvest energy after deployment.

Harvesting energy is usually not feasible without massively violating the desired form factor. For instance, solar panels are an order of magnitude larger that the sensor node. The same holds for most other techniques that draw energy from the node's environment (e.g., from moving air or water, temperature fluctuations or vibrations). Storing energy on the node using batteries also comes with certain limitations since they consume considerable space the more energy they retain. Battery technologies such as Nickel Metal Hydride (NiMH), Lithium Ion (LiIon) or Lithium Polymer (LiPo) that are available today dominate the form factor of a sensor node and no change is expected in the foreseeable future [158, 159].

Typical capacities of standard batteries are $1000 - 3000mAh$. Given a desired network lifetime of only one year and a battery capacity of $2000mAh$, a sensor node must use less than $231\mu A$ on average to fulfill this requirement. To give an impression how challenging a long-term operation actually is, imagine a sensor node that operates in two distinct modes: One normal mode (active CPU and the radio interface in receive or transmit mode) and one sleep mode (CPU dozes and radio is turned off). Assume that in normal mode, the node requires $40mA$ and in sleep mode only $1\mu A$. Such a node could only be active for 0.57% of the time while it remains in sleep mode for 99.43% of the time.

Consequently, conserving the scarce energy resources is mandatory for any WSN application. Besides the meager energy resources of a sensor node, other parameters such as computational power and storage resources are also subject to severe limitations. These will be discussed in Section 2.1.4.

Networking and Reliability By definition, the nodes are deployed in-situ where a phenomenon is to be observed. Thus, a sensor network is not operated under laboratory conditions but in tough environmental settings. The sensor nodes are exposed to a variety of changing climatic parameters like air humidity, temperature, sunlight or radiation. A WSN may be partially covered with water

due to heavy rain, floodwater or high tide. All these unpredictable and potentially rapidly changing conditions influence the nodes in unpredictable ways.

For instance, sensors could deliver erroneous readings when they are operated outside their specified limits and the radio communication characteristics can vary due to changes in the ambient conditions. Communication links that have worked reliably may fail in nondeterministic ways. This might only affect some links and the network remains connected or it may even become temporarily partitioned. Besides these environmental influences, the network topology may also change due to node movement, e.g., when sensor nodes are attached to animals, cars or humans. Individual nodes may fail due to battery exhaustion or hardware failures and new nodes may be added to the network during another deployment phase.

Therefore, the sensor network must continuously adapt to changes, re-organize itself accordingly and detect erroneous data to ensure a maintenance-free and reliable operation. For such complex distributed applications to operate properly, the choice of networking techniques is crucial. Since wireless networking is the key ingredient of WSNs, Section 2.1.3 discusses this topic in depth.

Context A deployed sensor network must ultimately accomplish its application task, which generally includes the acquisition of values from the node's sensors. However, a sensor reading (e.g., a temperature value) alone is merely useful; it must be embedded into the context of time and position to become meaningful. Consequently, the nodes in a sensor network need to establish a common understanding of time and location. However, acquiring this important context information is difficult and demands for new approaches.

As argued by Elson and Römer [43], classical time synchronization techniques are not applicable in WSNs. Consequently, a variety of different time synchronization protocols such as *Post-facto Synchronization* [41], *Reference-Broadcast Synchronization* [42] or *Traffic Induced Control of Time And Communication* (TICTAC, [19,26]) with different optimization goals were proposed. Besides the aforementioned ones, an assortment of other algorithms (e.g., [148,160,161,176]) has been developed. This accounts for the fact that a single solution is most likely not able to fulfill all needs.

A similar situation prevails in the area of localization where traditional localization technologies that rely on a fixed infrastructure (such as GPS) are typically not suitable. For this reason, a number of techniques based on infrastructure-free approaches were developed. For an in-depth discussion on this topic, please refer to work presented by Reichenbach et al. [137,138], Buschmann et al. [18,27] or Kröller et al. [48, 91].

Application Development All above-mentioned challenges represent run-time properties of WSNs that must be considered by an application developer when designing a reliable and energy efficient sensor network at compile-time. Due

to the fundamentally different nature of sensor networks, implementing applications for WSNs demands for new paradigms in software development. One has to deal with a multitude of difficulties such as the massively distributed nature of the system, the complexity of embedded programming, the resource constraints in terms of processing capabilities and memory availability, the lack of user interfaces for debugging, energy awareness and the size of the networks.

For this reason, a lot of research specifically targets these difficulties and a number of programming tools with a varying abstraction degree have been developed. These are further discussed in Section 2.1.5.

2.1.3. Networking Paradigms

Nowadays, networking is the major driving force in computing. Many contemporary applications such as e-commerce or e-mail are impossible without the global Internet. Despite the enormous growth of computer networks, the underlying principles remained mostly the same. Applications and protocols still strictly adhere to the *Open Systems Interconnection Reference Model* (OSI) or the *TCP/IP* reference model [32, 169].

Each layer of the OSI and TCP/IP model is responsible for a specific task (e.g., end-to-end data transport or physical hardware access), provides a clearly defined service to the upper layer and uses services of the lower layer to perform its service. The traditional Internet, which is based on this architecture, has demonstrated its flexibility for several decades and is the dominating networking technology nowadays. However, many assumptions of this classic networking technique cannot be applied to WSNs and issues such as energy consumption or addressing schemes require novel solutions.

Energy Consumption Before the advent of sensor networks, it was generally assumed that listening on a network interface is a cheap operation and thus it was ready-to-receive by default. With the availability of the first sensor network prototypes, it was obvious that the radio interface is the most energy-hungry component of a sensor node [95, 152]. Minimizing its energy consumption is therefore imperative to realize long network lifetimes. To achieve this, the radio interface should be switched on only for a fraction of time.

The media access control (MAC) layer was identified as the primary research target to reduce the energy consumption of the radio interface. Many innovative MAC schemes were proposed [67, 133, 139, 195] that trade energy consumption against latency and throughput. It was quickly realized that the archetypal, layered approach to networking makes further energy savings hard to achieve. Since only the application has the definite knowledge of its functionality, it needs a fine-grained control over the hardware to optimize its energy consumption. Consequently, it has been suggested to employ *cross-layer techniques* that expose the formerly private parameters of the individual layers to the application. This allows for a full parameterization of all layers by the application but

sacrifices the benefits of the layered approach. While this approach only conserves the energy resources of each single node, other approaches try to enhance the lifetime by network-wide means such as *clustering* or *topology control*.

Clustering algorithms form groups of sensor nodes, each of which elects a *cluster-head* (CH). Non-CH nodes do not transmit data directly to nodes in neighboring clusters, but only to their cluster-head. The cluster-head is responsible for the internal organization of the cluster and for the communication with other clusters. Internally, it arranges the duty-cycle of its group-members and aggregates the received data. Externally, the CH is in charge of all communication with other clusters via their cluster-heads. Clustering algorithms have the potential to reduce traffic and to enlarge the sleep-times of individual nodes in order to reduce the energy consumption of the network [112, 178].

Topology control algorithms adapt the transmission power of individual nodes and thus reduce the energy consumption of sensor nodes while optimizing the network's topology. The resulting topology yields less interference with other nodes and hence a higher capacity of the wireless channel. See [17, 101, 122, 181, 193] for in depth discussions on this topic.

Addressing Schemes Conventional systems assign a network-wide, unique identifier to each device. This makes perfect sense in systems where a particular device is responsible to deliver some kind of service (e.g., the web server hosting http://www.wikipedia.org). This address-based communication paradigm is not easily applicable in sensor networks since it is not known where a node with a specific address will be located and which node will adopt a certain role prior to their deployment. As a consequence, traditional ID-based addressing schemes will usually fail and a sensor network must therefore employ alternative means to identify the destination of network messages [2, 60].

For instance, properties of nodes such as their location, functionality or available resources may serve as addressing or forwarding criteria. Some protocols and algorithms also make use of a geographic region to convey the destination of data packets [83,86,88,111,180,194]. Another approach is to abstain completely from explicit addressing. *Directed Diffusion* [73] or *Rumor Routing* [16] are examples for data-centric, cross-layered routing protocols where the content of a message serves as the addressing information.

Standardization Standardization of the radio interface in WSNs is becoming increasingly important as sensor networks are on the verge of commercial success. The multiplicity of different hardware platforms – and their incompatible, proprietary radio interfaces (cp. Section 2.1.4) – prevents a collaboration of devices from different vendors. As a consequence, the *IEEE 802.15 WPAN Task Group 4* [71] published the IEEE 802.15.4 standard that specifies a physical (PHY) and a medium access (MAC) layer. The PHY layer operates at 2.4GHz (250kbit/s) or at 868/915MHz (20kbit/s and 40kbit/s). The IEEE 802.15.4 standard is a step towards homogeneous and interoperable WSN hard-

ware platforms.

Since IEEE 802.15.4 only standardizes the PHY and the MAC layer, the upper layers may still use incompatible protocols. The ZigBee Alliance [201] addresses this by defining standards that build upon IEEE 802.15.4. Similar to Bluetooth [70], ZigBee specifies application profiles that define how applications interact with each other. Despite these endeavors to standardize some application scenarios, the manifold application scenarios for WSNs suggest that ZigBee will not be the preferred solution in any case. Nevertheless, it seems a reasonable choice to agree upon IEEE 802.15.4 as the common building ground for future, interoperable WSN applications. An increasing number of research institutes and companies have recently switched to an IEEE 802.15.4 compatible radio interface.

2.1.4. Sensor Network Hardware Platforms

This section gives an impression on the features and the resources of current wireless sensor nodes. It is anticipated that even future sensor nodes will not have more resources at their disposal than the current generation [134]. Hence, the here-presented platforms are sound representatives for the current and future generation of sensor nodes.

One of the first sensor network platforms was developed by researchers of the UC Berkeley. The *Mica* hardware family [33, 64, 67] was designed as a general-purpose research node allowing researchers to validate their conjectures in real-world settings. Table 2.1 summarizes some of the features of this family, which constitutes today's most widely used WSN hardware platform.

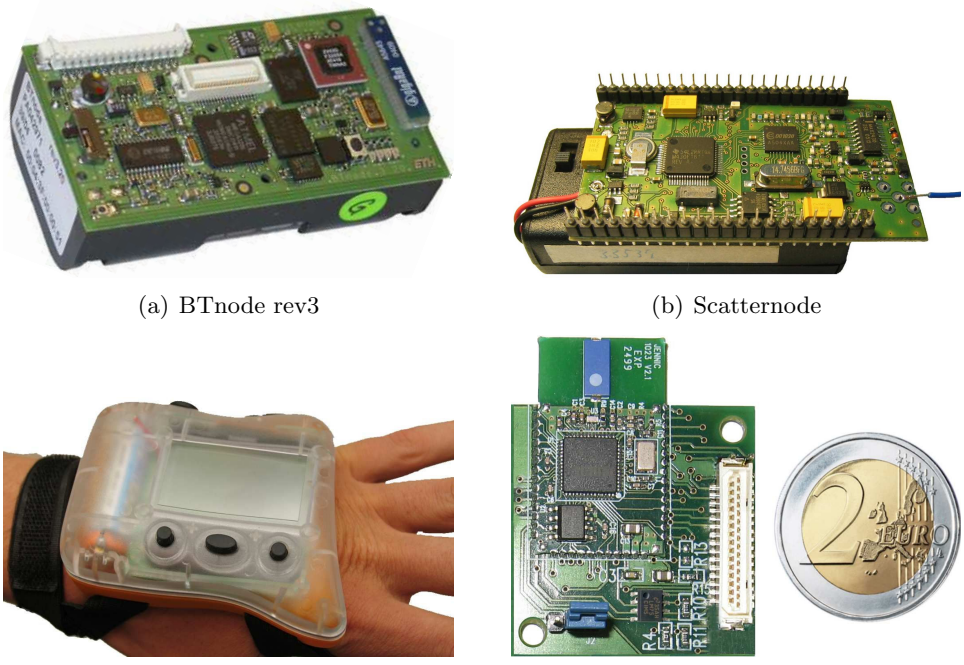
The Mica node comprises a CPU, a radio interface, volatile and non-volatile storage as well as an I/O-interface to connect additional sensors. The CPU is an 8-bit processor running at 4MHz and offering 4kBytes of RAM and 128kBytes of program memory. In addition, general-purpose I/O-pins, hardware timers, serial interfaces and an analog-digital converter are available. The radio interface operates on a single channel at 916MHz at a data rate of 40kBit/s. While the Mica is still an order of magnitude larger than envisioned, the *Spec* mote [66] has been created as a Mica compatible prototype with a form factor of only 2.5mm x 2.5mm, which demonstrates that the vision of tiny sensor nodes may turn into reality.

Despite its aptitude for research under laboratory conditions, the first Mica-series was not appropriate for real-world deployments mostly due to unreliable hardware components [108]. Hence, the second generation was designed to correct the shortcomings of the first generation. It comprises the *Mica2*, *Mica2Dot* and *MicaZ* motes. The Mica2 is the direct successor of the Mica mote and offers a better radio interface and a generally improved design [165]. The Mica2Dot is significantly smaller than the Mica2 mote at the price of slightly reduced functionality and extensibility. The MicaZ [36] replaces the proprietary radio interface of the other nodes with an IEEE 802.15.4 compatible radio. In 2004,

Year	Mica 2001	Mica2 2002	Mica2Dot 2002	Telos 2004
Microcontroller				
Program Memory (kB)	128	128	128	48
RAM (kB)	4	4	4	10
Active Power (mW)	8	33	8	3
Sleep Power (μW)	75	75	75	15
Non-volatile Storage (kB)	512	512	521	1024
Communication				
Radio	TR1000	CC1000	CC1000	CC2420
Standard	Proprietary	Proprietary	Proprietary	IEEE 802.15.4
Data rate (kbit/s)	40	38.4	38.4	250
RX power (mW)	12	29	29	38
TX power @ 0dBm (mW)	36	42	42	35
Platform				
Total active power (mW)	27	89	44	41
Extensibility	51-pin	51-pin	19-pin	16-pin
Dimensions (mm x mm)	57 x 31	58 x 32	25 x 6	65 x 31

Table 2.1.: Hardware properties of selected UC Berkeley sensor nodes

the Telos node[1] [116, 134] was introduced as the successor of the Mica2. It is a complete redesign based on the experiences of the Mica and the Mica2 series. The main objectives in the development were ultra-low-power operation, short wake-up times and the use of a standardized IEEE 802.15.4 radio.



(a) BTnode rev3 (b) Scatternode

(c) pacemate (d) iSense

Figure 2.3.: A selection of WSN hardware platforms

Apart from the UC Berkeley, different other WSN platforms have been devel-

[1]The Telos node is also known as Tmote Sky, which is equivalent to Telos Revision B.

oped such as the BTnode [11,12] from the ETH Zurich. It roughly features the same system core and radio properties as the Mica2. Besides a Mica2 compatible radio, the BTnodes offer Bluetooth [70] communication to interact with notebooks, PDAs or cell phones. Figure 2.3(a) shows the third revision of the BTnode attached to a battery pack with two AA cells.

Researchers from the FU Berlin introduced the *Embedded Sensor Board* (ESB) research and teaching platform [37,147]. It comprises a variety of sensors (passive infrared for motion detection, temperature and luminance), a proprietary radio interface and an extension interface. The ESB has evolved into a commercial version, the Scatternode [146] (cp. Figure 2.3(b)) and the newest incarnation called *Modular Sensor Board* (MSB). Newer platforms such as the iSense hardware platform [23] depicted in Figure 2.3(d) demonstrate the increasing commercial interest in WSNs and the growing standardization of the radio interface by supporting the IEEE 802.15.4 standard.

The above-mentioned sensor nodes allow for a wide range of applications and are extensible to add custom sensor-boards. Apart from these generic hardware platforms, some custom platforms were developed that optimally match the requirements of specific applications where generic platforms are not applicable. For instance, *Body Area Networks* (BANs) – a variant of WSNs – facilitate a fine grained monitoring of human physical parameters. Here, specialized sensors are necessary and ergonomics is an important acceptance factor. The BAN project *MarathonNet* project developed the *pacemate* [63,129] to monitor runners during competitions. Figure 2.3(c) depicts a pacemate worn on the back of the hand by a marathon runner. Apart from the standard features of a sensor node, it also features as display and measures the heart rate using a chest belt worn by the runner.

To summarize, one fact is common for all WSN hardware platforms: They have scarce onboard resources in terms of memory, computational resources and energy. This imposes strict limits on the applications running on single nodes.

2.1.5. Operating Systems and Programming Abstractions

Software development for traditional embedded systems is already a challenging task in itself [96]. When developing applications for WSNs, novel application scenarios, self-adaptation to ambient conditions, energy awareness and strict resource-constraints further complicate this situation. Software and programming abstractions for WSNs must therefore cope with a variety of novel requirements:

- Energy-, resource- and context-awareness
- Reliability, fault-tolerance and self-configuration
- Scale to thousands or millions of nodes
- Exchange of data with traditional networks

- Heterogeneous device architecture

Apart from these properties, many different application-specific requirements have influenced the design of existing software tools for WSNs. These requirements are often conflictive and applications must trade one property against the other. Therefore, researchers have proposed tools, architectures and alike with a varying level of abstraction.

Operating Systems One challenge is the development of operating systems (OS) that control the sensor network hardware platforms such as the ones mentioned in Section 2.1.4. Traditional embedded operating systems such as VxWorks [184], μCLinux [39] or Windows Embedded CE [113] typically target hardware platforms that are equipped with an order of magnitude superior resources. Furthermore, they offer advanced services such as preemptive multitasking, real-time support, memory protection, TCP/IP and support standards such as POSIX that manifest themselves in the footprint of the OS. Most of the offered features are neither feasible nor desired in WSNs while other important issues such as energy awareness, minimal resource consumption or cross-layer approaches are not addressed at all.

TinyOS [65, 100, 173] is the most frequently used OS for WSN in current research. It supports a variety of hardware platforms such as the complete Mica family and the Telos motes. TinyOS is an event-based system that strives to require only a minimum of resources. To realize these features, the authors proposed a novel programming language called nesC [56], which is a superset of the standard C language. Its component-based architecture enables application developers to "wire" nesC-interfaces (e.g., the radio or timer interface with the application). The nesC compiler then assembles concrete implementations of the interfaces and generates a TinyOS instance for a specific hardware platform. The application's logic itself is mostly implemented in plain C. The architecture of nesC resembles standard C++ inheritance mechanisms where application developers only know the abstract interface of a class while the actual implementation is hidden in a subclass of the interface. To realize this feature, C++ uses late binding techniques that determine the address of a concrete method at run-time. nesC performs a static, early binding at compile-time to avoid the run-time overhead of late binding at the cost of reduced run-time options.

Contiki [40] is another recognized, portable and adaptable operating system written in plain C. Comparable to TinyOS, it consists of an event-driven kernel that provides simple cooperative multitasking as well as dynamic loading and unloading of applications at runtime. It can be configured to use only a few kBytes of program and a few tens of bytes of memory. While both above-mentioned operating systems strive for portability and general applicability in WSNs, a variety of other operating systems with specialized properties and hardware affinity have been developed, e.g., Scatterware [37] for the Scatterweb nodes or BTNut [10] for the BTNodes.

By hiding most of the intricate details of the underlying hardware platform, these operating systems allow a programmer to concentrate on the application development instead of operating with registers, bus timings, etc.

Programming Abstractions Despite the level of abstraction offered by WSN operating systems, programming a sensor network remains a demanding challenge [87]. As a result, different programming abstractions have been developed that intend to ease the implementation of WSN applications. The most prominent amongst them can be classified into one or more of the following categories:

- Macro- or swarm-Programming
- Virtual Machine
- Database-like
- Middleware and Code Synthesis
- Simulation and Visualization

The key idea behind *macro- or swarm-programming* is to program the sensor network as a whole instead of writing code for single sensor nodes. A high-level behavior description is fed into the network and the single nodes derive their tasks from this global target specification. Examples for this type of programming abstraction are *SWARMS* [21,89] and *TAG* [106]. They relieve programmers from dealing with the complicated aspects of node level programming and support a platform-independent specification of the application's task. Guided by biological paradigms, an important research direction in sensor networks is to mimic the behavior of swarms. The idea is to use local rules that evict a global behavior of the network. For instance, *Generic Role Assignment* [52] uses rules to assign roles to individual sensor nodes (such as leader or gateway). *FACTS* [170] provides a rule based language to express the behavior of the network.

To address the problem of heterogeneity, *virtual machine* (VM) techniques were proposed for WSNs. Programs that target a virtual machine are inherently portable and may run on virtually any kind of hardware – if either hardware support for the VM or an interpreter is available. A common drawback of traditional VMs such as the *Java Virtual Machine* (JVM, [102]), the *.NET Common Language Infrastructure* (CLI, [74]) or Sun's KVM [163] is the footprint of the VM and the execution overhead compared to native code. Hence, dedicated virtual machines were proposed that tackle these challenges. Maté [97] realizes a simple VM on TinyOS-based nodes that allows programs to be written with only some 100 bytes in size. Because of its limited functionality, Koshy and Pandey presented VM* [90], a VM that supersedes Maté by synthesizing application-specific VMs. Programmers develop their application against a set of platform-independent interfaces. VM* analyzes the application and assembles a VM*-instance that only contains required functionality. The *application specific virtual machine* (ASVM) approach [98] is very similar and also generates specialized and lean VMs with an optimized instruction set for TinyOS

operated sensor nodes.

Database-like approaches are similar to the swarm programming techniques as they also use a high-level language to specify the behavior of the entire network. In contrast to them, the application's goal is not directly specified but queries are formulated that request specific data from the network – the WSN is treated like a distributed database. A detailed discussion is deferred until Section 6.3.

In traditional networks, *middleware* technologies have proven to be an excellent means for hiding the complex details of the underlying hardware and networking infrastructure from the application. The fundamental goals are essentially the same in WSNs but traditional middleware solutions do not match the different requirements of sensor networks. Despite the challenges that render the development of comprehensive middleware solutions for WSNs extremely difficult, middleware techniques are seen as the paramount solution for efficient sensor network application development [59,143]. Because of their superior significance for WSNs, middleware techniques are separately discussed in Chapter 6.

Simulation and *visualization* tools are of equal importance for the development and operation of WSN applications. Simulation tools support developers in optimizing and evaluating the application prior to the network's deployment on real hardware. Since sensor nodes typically do not offer any kind of user interface to the user, visualizations serve as surrogates to display the state of individual nodes or the network as a whole. Simulation frameworks are presented in Chapter 4 and visualization tools are covered in Chapter 5.

2.2. Extensible Markup Language Technologies

The unparalleled success of the Internet was fueled by the development of the Hyper-Text Markup Language (HTML) in the 1990s. HTML has paved the way for a widespread use of text markup languages. The World Wide Web Consortium (W3C) as the standardizing committee of the HTML language evolved the fundamental ideas of HTML to a more general framework. Instead of only targeting web pages, the *Extensible Markup Language* (XML) and related technologies have been standardized to support generic text markup. In the following, XML (Section 2.2.1) and XML Schema (Section 2.2.2) are introduced as far as they are relevant for this thesis. The terminology conforms to [187, 188, 191, 192].

2.2.1. XML

This section introduces the fundamentals of XML, the successor of the older and more feature-rich *Standard Generalized Markup Language* (SGML) which was frequently considered to be excessively complex. SGML's and XML's feature sets are often compared by using a Pareto distribution: While XML has only 20% of SGML's complexity, it allows the realization of 80% of the possible ap-

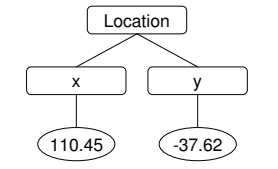
plication space of SGML. This flexibility has made XML virtually omnipresent and it is endorsed by huge companies, by many internet protocols as well as by the Open Source community. It superseded countless binary data storage formats and a huge number of applications use XML as their foremost data storage, import and export format. XML's main purpose is to provide a standard for exchanging semi-structured data in a human and machine-readable tree-like form. Semi-structured means that XML does not specify how data is represented exactly, it leaves this up to the individual application. Hence, XML defines how documents are structured but also allows unstructured parts inside the document.

Structure

An XML *document* consists of two fundamental language constructs: *markup* and *character data*. The markup is the building block that frames the character data into a tree-like structure. To put it the other way round, character data is everything that is not markup. Figure 2.4(a) shows a very basic document that represents an (x, y)-coordinate tuple. It is comprised of character data marked in bold as well as markup. Figure 2.4(b) provides the semantically equivalent tree representation of the document shown in Figure 2.4(a).

```
1 <Location>
2   <x>110.45</x>
3   <y>-37.62</y>
4 </Location>
```

(a) Textual representation



(b) Tree representation

Figure 2.4.: Simple document representing an (x, y)-coordinate tuple

The fundamental constructs that make up the markup in a document are *elements*. In Figure 2.4(b), elements are denoted by rectangles in the tree. Character data at the leaves of the tree are denoted by ovals. The tree structure of a document yields a hierarchical relation between individual elements. The element at the top of this tree is called *root element*. An element e_c that is directly contained inside another element e_p is called the *child element* of e_p and e_p is defined as the *parent element* of e_c. Elements that share the same parent element are called *siblings*. The parent and child relationship is also applied recursively and elements e_d that are the recursive children of an element e_p are called *descendants*. If e_d is the descendant of e_p, then e_p is the *ancestor* of e_d.

In the commonly used textual representation of documents (cp. Figure 2.4(a)), elements are denoted by an opening and a closing *tag*, also called *start* and *end tag*. A start tag consists of the element's name that is surrounded by < and > as in `<Location>`. Like the start tag, the end tag is encompassed by < and >

but with an additional / following the < as in </Location>.

Flanked by the start and end tag, the content of an element may occur. The content can be empty, include plain character sequences, other elements or both. If no content is present, the start and end tag may be combined to a single abbreviated tag that is functionally equivalent. This *empty element* is similar to the end tag but the slash follows the element's name instead of preceding it (e.g., <Location/>).

Apart from the element, another markup construct called *attribute* allows structuring the content of a document. These are $(name, value)$-pairs that are contained between the angle brackets of an opening tag and follow the element's name. Attributes cannot contain other attributes or elements but only character data. An element may contain more than one attribute if their names are unique within this element. Figure 2.5 shows a document containing one attribute called **unit** inside the **x** element. As a rule of thumb, they are used for a further description of the element's character content.

```
1 <Location>
2   <x unit="km">110.45</x>
3   <y>-37.62</y>
4 </Location>
```

Figure 2.5.: Using attributes in a document.

Besides elements, attributes and character data, any document may contain an optional *prolog* that must – if present – appear before the root element. It carries data that is needed for processing the following document. The prolog may start with the *XML declaration* containing the version number of the XML standard used and may contain the character encoding of the document. Figure 2.6 shows a document that has an XML prolog consisting of the XML declaration and a further *processing instruction*. Processing instructions are similar to the XML declaration but they may occur anywhere in a document where a tag may be used and must not begin with **XML** or **xml**. They are used to convey application-specific parsing instructions and they are neither part of the character data nor of the tree-like structure.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <?appspecific processing="hint" ?>
3 <Location>
4   <x unit="km">110.45</x>
5   <y>-37.62</y>
6 </Location>
```

Figure 2.6.: Document with the optional XML declaration and a processing instruction

After the prolog, the actual document content is encoded as markup and character data. As explained above, some characters (such as <, > and &) have a

particular connotation as they separate markup from character data. If these characters should occur as part of the character data, they must be escaped to void their special meaning. XML allows two distinct possibilities for escaping them: *entity references* and *CDATA sections*. Entity references may be used as substitutes for other characters. XML offers several predefined entity references: `&` (&), `<` (<), `>` (>), `&apos;` (') and `&quot;` ("). Apart from the predefined ones, numerical entities represent characters by a numerical value (e.g., `&#38;` instead of `&`). CDATA sections begin with `<![CDATA[` and end with `]]>`. Any special character in between is treated as character data. Only the > character must be escaped inside a CDATA section using `>`.

Namespaces

If these rules are consequently applied, markup and character data are easily separated. However, it may happen that the same element or attribute name must be reused with different semantics. The document shown in Figure 2.7 uses the same element name `Location` twice to express to different things: a physical and an abstract location.

```
1 <System>
2   <Location>
3     <x>110.45</x>
4     <y>-37.62</y>
5   </Location>
6   <Location>South of Luebeck</Location>
7 </System>
```

Figure 2.7.: Document with ambiguous tag names

In order to distinguish these two elements, XML allows the use of *namespaces* [187]. A namespace is identified by a *Uniform Resource Identifier* (URI, [168]). Before a namespace can be used in the document, it must be declared by assigning a short prefix to the (typically long) URI of the namespace. This declaration may be present in each starting tag and is valid until the corresponding end tag is reached. Hence, declaring a namespace in the root element of a document makes the namespace's identifier span the whole document.

A namespace declaration uses standard XML attributes to express the assignment of an identifier to a namespace URI. The attribute's name is reserved and starts with `xmlns:` followed by the namespace identifier. The value of this attribute is the URI of the namespace, and the complete namespace declaration is of the form `<tagname xmlns:prefix="URI">`. Associating a tag or attribute with a declared namespace is done by prefixing the element's or attribute's name with the namespace identifier separated by a colon. Such a composite of prefix and name is called *Qualified Name* (QName).

The use of namespaces is shown in Figure 2.8: the prefix `phys` is bound

to the namespace URI `http://www.example.com/phys` and `desc` to `http://www.example.com/desc`. Both are used to uniquely identify the previously ambiguous elements *Location*, x and y to the respective namespace. Please note that the URIs simply serve as unique identifiers and they do not necessarily point to any existing document.

```
1 <phys:System xmlns:phys="http://www.example.com/phys"
2              xmlns:desc="http://www.example.com/desc">
3   <phys:Location>
4     <phys:x>110.45</phys:x>
5     <phys:y>-37.62</phys:y>
6   </phys:Location>
7   <desc:Location>192.168.1.1</desc:Location>
8 </phys:System>
```

Figure 2.8.: Document from Figure 2.7 augmented with namespaces to avoid ambiguities

Correctness

As described above, XML defines very strict rules and regulates how a document must be structured. A document that adheres to these rules is called *well-formed*. This means that a document has only one root element, that the elements are correctly nested and that it follows all rules of the XML syntax. Figure 2.9 presents a document that is not well-formed: In line 4, the corresponding end tag for the start tag `x` is missing. Next, the end tags for the elements `y` and `Location` appear in the wrong order (line 8-9). Finally, the document has more than one root element (line 1 and 11).

```
1  <LocationList>
2    <Location>
3      <x>
4      <y>-37.62</y>
5    </Location>
6    <Location>
7      <x>-12.78</x>
8      <y>-97.10</Location>
9    </y>
10 </LocationList>
11 <AnotherRootElement/>
```

Figure 2.9.: Not well-formed XML document

The attribute well-formed is strictly syntactical and does not give any hints on the content of the document. Depending on the application, it may be essential to restrict the structure of the markup and the character data. This allows machines to process documents easily due to their well-known and structured content. To achieve this goal, well-formed documents may additionally adhere

to a specific grammar. If they are complying to such a grammar, they are called *valid*.

Quite a few of these grammar languages have been proposed in the past. A commonly used grammar language is the *Document Type Definition* (DTD) that was specified in conjunction with XML. Due to various limitations (e.g., no support for namespaces and a crude, non-expressive language) it is mostly obsolete nowadays. To fill this gap, the W3C standardized the successor of DTDs called XML Schema[2]. Roughly at the same time, other XML schema languages such as Relax NG [76] have been developed. Nevertheless, as a standard published by the renowned W3C, XML Schema is currently the predominant language.

2.2.2. XML Schema

For many applications XML is too flexible and thus too unspecific to be of any direct use. It is often handy to constrain the set of elements, tags, attributes and their contents to help machines understand the meaning of documents. XML Schema [191, 192] provides a language that allows to create dialects of XML. Using this language it is possible to define the order of elements, their relations and the structure of character data.

For example, the `Location` element depicted in Figure 2.7 may contain arbitrary, valid character data or even other markup. Understanding these generic documents with the help of machines is therefore challenging. If an XML Schema had limited the contents of the `Location` element to only numerical values, this task would have been easily accomplished. A language described by an XML Schema document is referred to as an *XML language* or *XML dialect*. A document that is well-formed and that conforms to a given XML Schema is called *valid* or *instance document*. Programs that facilitate this verification are so-called *validating parsers*.

This section presents the basic structure of XML Schema and introduces the most important building blocks that are available for developers in order to specify an XML language. An XML Schema document itself is a well-formed document that adheres to a specific grammar. The normative documents *XML Schema 1.1 Part 1: Structures* [191] and *XML Schema 1.1 Part 2: Datatypes* [192] are written in plain text to bootstrap the syntax of XML Schema. Additionally, an XML Schema that describes the XML Schema language is available[3] but not normative.

Structure

The root element of XML Schema documents is `schema`. The XML Schema standard prescribes the use of namespaces to identify all XML Schema elements

[2] Note the capital S

[3] `http://www.w3.org/2001/XMLSchema.xsd`

unambiguously. All elements that belong to the XML Schema document must be associated with the namespace `http://www.w3.org/2001/XMLSchema`.

Figure 2.10(a) depicts a simple XML Schema document. Line 1 shows the root element. Here, the prefix `xs` is assigned to the XML Schema namespace. Hence, all XML Schema tags are prefixed with `xs:` in this document.

In line 2, the `element` tag declares an element of the type `long` with the name `counter`. Element tags declare the names and the types of the elements that may occur as root elements in valid instance documents. Figure 2.10(b) depicts a valid instance document that conforms to this XML language. A single root element named `counter` contains character data representing an integer value that complies with the constraints of a type called `long`.

```
1 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
2   <xs:element name="counter" type="xs:long"/>
3 </xs:schema>
```

(a) Element declaration in XML Schema

```
1 <counter>194</counter>
```

(b) Valid instance document

Figure 2.10.: Simple XML Schema document and an exemplary instance document

As indicated by the existence of two normative documents, the core of XML Schema is divided into two distinct parts. The first one ("Structures") specifies how the markup of instance documents can be structured by using XML Schema. The second one ("Datatypes") defines how the content of character data in instance documents may be restricted. A key element of both normative documents is to provide rules describing how new data types are derived from a set of pre-defined ones. These data types are either *simple types* or *complex types*. Simple types define the possible contents for character data. Complex types encompass different simple types and define how the markup frames the character data in a specific structure.

Simple Types

XML Schema provides a set of pre-defined types, so-called *built-in types*. These are readily available in any XML Schema document. Besides the special type `anyType` all other built-in types are of simple type. These can be categorized into integer and real numbers, strings and composed types such as date and time types. The built-in types and their inheritance relationship is shown in Figure 2.11[4].

[4]By courtesy of the W3C: Copyright © 2006-02-17 World Wide Web Consortium, (Massachusetts Institute of Technology, European Research Consortium for Informatics and

The most general data type at the root of the hierarchy embraces the value space of all its child data types. When moving towards the tree's leaves, the data types are further restricted in their value space.

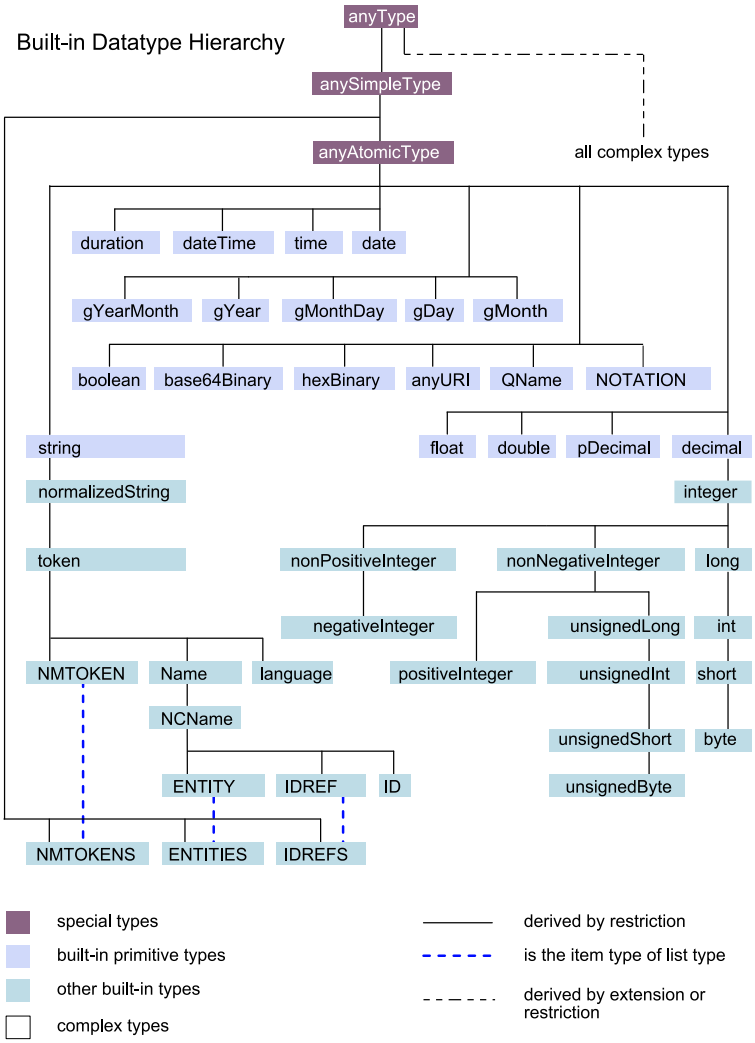


Figure 2.11.: The hierarchy of built-in data types in XML Schema

Figure 2.10(a) represents an example of how these built-in types are utilized. In this XML Schema document, the integer data type `long` is used. A document that conforms to this XML language is depicted in Figure 2.10(b).

As mentioned above, the built-in simple types serve as the basis for user-defined simple types. A simple type is derived from a built-in or another simple type by restricting the value space through so called *facets*. This feature is extremely useful, as a validating parser can already check if a given document adheres to the expected grammar. Hence, applications can largely omit error-checking

code when using such a parser.

Table 2.2 lists the available facets and their semantics. An example on how to define a simple type using facets is given in Figure 2.12. The element `smallnum` is defined as a restriction of the base type `byte` with a value space ranging from 17 to 32 by using the facets `minExclusive` and `maxInclusive`.

Facet	Description
length	Number of units of length (E.g., number of characters for strings)
minLength	Minimum number of units of length
maxLength	Minimum number of units of length
pattern	Constrains the value space to match the given regular expression
enumeration	Limits the value space to a fixed set of values
whiteSpace	Rules for whitespace treatment (preserve, replace, collapse)
minInclusive	Constrains the value space to a minimum value
maxInclusive	Constrains the value space to a maximum value
minExclusive	Constrains the value space to a minimum (excluded) value
maxExclusive	Constrains the value space to a maximum (excluded) value
totalDigits	Defines the maximum total number of decimal digits
fractionDigits	Defines the maximum number of fractional digits
minScale	Limits the arithmetic precision to a minimal power
maxScale	Limits the arithmetic precision to a maximal power

Table 2.2.: Constraining XML Schema facets

```
1  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
2    <xs:element name="smallnum">
3      <xs:simpleType>
4        <xs:restriction base="xs:byte">
5          <xs:minExclusive value="16"/>
6          <xs:maxInclusive value="32"/>
7        </xs:restriction>
8      </xs:simpleType>
9    </xs:element>
10 </xs:schema>
```

Figure 2.12.: Simple type definition by restricting the built-in data type `byte`

Complex Types

Complex types may be composed of several simple and complex types and they are used to structure the markup of a document. Thus, they define how elements are nested, which attributes may be present and how their contents may look like. XML Schema supports three different types for composing the contents of complex types: *sequence*, *choice* and *all*. In the following, these three compositions are described in detail.

A *sequence* is a composition where the order of the contained elements is fixed. Figure 2.13(a) shows how this complex type is expressed in XML Schema. Figure 2.13(b) shows a valid instance document and Figure 2.13(c) depicts

an invalid document (wrong order). A *choice* is similar to a sequence, but only one of the contained elements may be present in any given instance document. The *all* composition is somehow special since the ordering of the contained data types is arbitrary in instance documents but each element may occur at most once.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="numlist">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="abc" type="xs:float"/>
        <xs:element name="xyz" type="xs:byte"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

(a) Complex type with a sequence of two elements

```
<numlist>
 <abc>3.1415</abc>
 <xyz>-11</xyz>
</numlist>
```

(b) Valid instance document

```
<numlist>
 <xyz>-11</xyz>
 <abc>3.1415</abc>
</numlist>
```

(c) Invalid instance document (wrong order of the elements)

Figure 2.13.: Example using a sequence composition

Each element of a complex type definition may contain a specification for how often the element may occur in valid instance documents. The attributes `minOccurs` and `maxOccurs` ($0 \leq minOccurs \leq maxOccurs$) are used to specify the number of possible element occurrences. If not specified, they have a default value of one. An exception is the special value `unbounded` for the `maxOccurs` attribute that permits an infinite number of occurrences. Figure 2.14(a) presents an XML Schema that makes use of these attributes and Figure 2.14(b) shows a valid document of this XML language.

In the examples above, the simple and complex types are defined locally inside the `element` tags. Another possibility is to define them globally as children of the `schema` tag. Figure 2.15 shows the definition of a *global complex type* with the name `Location` and an element named `loc` of the type `Location`. In contrast to the previous examples, global type definitions may be repeatedly used within the schema document while local complex types are only valid inside the enclosing element. The same applies to the definition of *global simple*

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="numvalue">
    <xs:complexType>
      <xs:choice>
        <xs:element name="abc" type="xs:float"
                    minOccurs="2" maxOccurs="4"/>
        <xs:element name="xyz" type="xs:byte"/>
      </xs:choice>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

(a) XML Schema document

```
<numvalue>
 <abc>1.23</abc>
 <abc>4.56</abc>
 <xyz>3</xyz>
</numvalue>
```

(b) Valid instance document

Figure 2.14.: Example on the use of `minOccurs` and `maxOccurs`

types. These are declared using the tag `simpleType`.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="loc" type="Location"/>

  <xs:complexType name="Location">
    <xs:sequence>
      <xs:element name="x" type="xs:double"/>
      <xs:element name="y" type="xs:double"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Figure 2.15.: XML Schema global type definition

Advanced Features

Apart from element, simple and complex type declarations, XML Schema offers a variety of advanced features. In the following, two important aspects that are relevant for this work are introduced: namespace support and annotations.

As discussed in Section 2.2.1, elements and attributes in documents may have an assigned namespace. Consequently, XML Schema allows specifying the namespace of the elements described by a specific schema document. The attribute `targetNamespace` in the `schema` tag declares the namespace URI that valid instance documents belong to. To link an instance document with

its corresponding XML Schema, the attribute `schemaLocation` may be used to point to the physical location of the XML Schema document. This attribute must be assigned to the namespace URI `http://www.w3.org/2000/10/XMLSchema-instance`.

Figure 2.16(a) shows an XML Schema document that uses the `targetNamespace` attribute to specify that valid instance documents must belong to the namespace `http://www.test.com/nspace`. The instance document presented in Figure 2.16(b) assigns the prefix `nsp` to this namespace URI. The tag name `nsp:shortnum` is therefore unambiguously identified as a member of this namespace. In addition, this instance document points to the physical location of the XML Schema document by using the `schemaLocation` attribute. Validating parsers may use this information for retrieving the XML Schema document when checking whether a given document is a valid instance document.

```
1 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
2            targetNamespace="http://www.test.com/nspace">
3   <xs:element name="shortnum" type="xs:short"/>
4 </xs:schema>
```

(a) Target namespace in XML Schema

```
1 <nsp:shortnum xmlns:nsp="http://www.test.com/nspace"
2   xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance"
3   xsi:schemaLocation="http://www.test.com/test.xsd">
4   -1297
5 </nsp:shortnum>
```

(b) Valid instance document

Figure 2.16.: Use of namespaces in XML Schema

Finally, XML Schema supports the annotation of nearly all schema tags. Using this feature, application-specific data are attached directly to the tags of an XML Schema document. This is commonly used to extend the feature set of XML Schema. These annotations are divided into a human- and a machine-readable part. Figure 2.17 shows an example where both human- and machine-readable annotations are attached to the `smallnum` element declaration.

XML Schema provides the `annotation` tag that may be contained as a child in most other XML Schema tags. This may contain multiple `documentation` and/or `appinfo` tags. The `documentation` tag is intended to provide information for human readers while the `appinfo` tag serves as the frame for machine readable information. The content of both elements is arbitrary and may be comprised of well-formed markup and character data. Hence, the appinfo-tag may contain even complex documents. While these are ignored by validating parsers, they could provide important information for application-specific parsers or other tools that can process XML Schema documents.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="smallnum" type="xs:byte">
    <xs:annotation>
      <xs:appinfo>
        <Machine>readable</Machine>
      </xs:appinfo>
      <xs:documentation xml:lang="en">
        Human-readable
      </xs:documentation>
    </xs:annotation>
  </xs:element>
</xs:schema>
```

Figure 2.17.: Annotations in XML Schema documents

2.3. Information Theory

A frequent task in computer systems is to find an efficient binary representation of information entities. Common examples are the compression of files before they are backed up to tape or the compression of data in general before it is transmitted over wide area network connections. This is often beneficial to increase the efficiency of a system, e.g., to require less tapes for backup or to improve the utilization of expensive communication links. The fundamental theoretical limits for these optimizations were established by Claude E. Shannon in 1948 [149].

For this thesis, a special aspect of his theories is of major importance. Shannon's *source coding theorem* provides the limits of data compression and, as a result, a lower bound on the number of bits needed to encode a specific piece of information. Source coding therefore means eliminating redundancy and irrelevance in the stream of symbols that is emitted by a data source. It is used by our (de-)serialization scheme called *microFibre* that is introduced in Section 6.5.

Imagine a data source that emits a number of discrete symbols $X = \{x_1, \ldots, x_n\}$. This set X is called the alphabet of the source. The probability that the source emits the symbol x_i is denoted by $p(x_i)$ $(i \in \{1, \ldots, n\})$. Since the source always emits one symbol out of its alphabet, $\sum_{i=1}^{n} p(x_i) = 1$ must hold. Definition 1 provides Shannon's measure of how much information is contained in each symbol $x_i \in X$.

Definition 1 (Entropy of a symbol)

$$H(x_i) = log_2 \frac{1}{p\,(x_i)} = -log_2\,(p\,(x_i)) \tag{2.1}$$

denotes the entropy (or information content) of a symbol $x_i \in X$. The unit of $H(x_i)$ is "binary digit" or "bit".

A symbol that occurs with a high probability has therefore lower entropy than a symbol with a very low occurrence probability. Because the receiver is not

aware of the next symbol transmitted by the source, the amount of information contained in a symbol is a measure for the *uncertainty* eliminated at the receiver upon the reception of this symbol. For instance, a symbol with a probability near one only eliminates very little uncertainty because it occurs extremely frequently. A symbol with a probability near zero eliminates a high degree of uncertainty because of its rare occurrence.

Beyond the entropy of a single symbol, the entropy of the data source as a whole can be defined. This *Shannon entropy* (or the entropy of a data source) gives the average number of bits that can be used to communicate information losslessly. Definition 2 provides the formal notation of a data source's entropy.

Definition 2 (Shannon Entropy)

$$H(X) = \sum_{i=1}^{n} p(x_i)\, H(x_i) \tag{2.2}$$

$$= -\sum_{i=1}^{n} p(x_i)\, log_2\left(p(x_i)\right) \tag{2.3}$$

denotes the entropy of a data source that emits the symbols $x_i \in \{x_1, \ldots, x_n\}$ *with a probability of* $p(x_i)$ *(*$i \in \{1, \ldots, n\}$*,* $\sum_{i=1}^{n} p(x_i) = 1$*). The unit of* $H(X)$ *is "bit/symbol".*

In the special case that all symbols emitted by a specific data source are equally likely, i.e., $p(x_i) = \frac{1}{n}$, the entropy of this data source is at its maximum. Definition 3 uses Definition 2 with $p(x_i) = \frac{1}{n}$ to derive the entropy of this special data source. The resulting formula is independent of the individual (equal) probabilities and the number of bits required to represent a symbol only depends on n. Please note that the values of $H(x_i)$, $H(X)$ and $H_{max}(X)$ are typically no integer values. To actually represent a symbol in a computer system, it is necessary to round up to the next integer value, e.g. $n = \lceil H(x_i) \rceil$.

Definition 3 (Maximal Shannon Entropy)

$$H_{max}(X) = -\sum_{i=1}^{n} \frac{1}{n} log_2 \frac{1}{n} \tag{2.4}$$

$$= -n\left(\frac{1}{n} log_2 \frac{1}{n}\right) \tag{2.5}$$

$$= -log_2 \frac{1}{n} \tag{2.6}$$

$$= log_2(n) \tag{2.7}$$

denotes the entropy of a data source that emits the symbols $x_i \in \{x_1, \ldots, x_n\}$ *with an constant probability of* $p(x_i) = \frac{1}{n}$ *(*$i \in \{1, \ldots, n\}$*. The unit of* $H_{max}(X)$ *is "bit/symbol".*

In cases where not all the symbols are equally likely, so-called *entropy coding* yields the shortest possible bit-lengths. An entropy encoding assigns short

replacement symbols to very frequent source symbols and longer replacement symbols to less frequent ones. The most ubiquitously known entropy encoding is the *Huffman coding* [68]. To create such a Huffman coding, a binary tree is constructed that ultimately contains the information to derive the set of replacement symbols $R = \{r_1, \ldots, r_n\}$. R is provably optimal and no other R' can yield a lower Shannon Entropy. Algorithm 1 presents the steps in order to construct this binary tree, the so-called *Huffman Tree*.

Algorithm 1 Construction of a Huffman Tree

Require: X {Set of symbols $x_i \in \{x_1, \ldots, x_n\}$}
Require: P {Set of probabilities $p(x_i)$ $(i_1, \ldots, x_n, \sum_{i=1}^{n} p(x_i) = 1\}$
1: *vertices* := Create a vertex for each symbol x_i with the probability p_i
2: **while** $|vertices| > 1$ **do**
3: n_1 := Vertex from X with the lowest probability, remove n_1 from *vertices*
4: n_2 := Vertex from X with the lowest probability, remove n_2 from *vertices*
5: t := Create temporary vertex
6: $t.probability = n_1.probability + n_2.probability$
7: Set n_1 and n_2 as children of t, label one edge with 0 and one with 1
8: Insert t into *vertices*
9: **end while**
10: **return** first(*vertices*) {The remaining node is the root of the Huffman tree}

Provided with the set of source symbols X and the corresponding set of probabilities P, the algorithm starts by creating a set of vertices (denoted by *vertices*) using X and P. Each vertex is hereby labeled with a symbol $x_i \in X$ and its matching probability $p_i \in P$ (line 1). The remainder of the algorithm runs repeatedly until only one vertex is left in *vertices* (line 3-8). In every iteration, two nodes n_1 and n_2 with the lowest probabilities in *vertices* are removed from the set. A temporary vertex t is created and added to *vertices*. It is labeled with a temporary name and the sum of the probabilities of n_1 and n_2. Furthermore, n_1 and n_2 are attached as child vertices to this temporary node. The edges are labeled with 0 and 1 respectively. Finally, the remaining node in *vertices* is the root node of the Huffman Tree (line 10). The individual replacement symbols $r(x_i)$ for an x_i are derived by concatenating the edge labels starting from the root of the tree until the symbol x_i is found.

In the following, the process of constructing a Huffman Tree is illustrated using a simple example. Table 2.3 depicts a set of source symbols along with their assigned probabilities. It also provides the entropy of each symbol (calculated by using Definition 1). The entropy of the data source as a whole can be calculated by applying Definition 2. This gives a Shannon Entropy for this data source of $H(X) \approx 1.79 bit/symbol$. Looking at the entropy of the individual symbols, it is clearly visible that less frequent symbols have a higher value than more frequent ones.

Figure 2.18 shows the resulting Huffman Tree that is produced by running Algorithm 1. At the beginning, the nodes representing symbols D and C are removed from *vertices* because the have the lowest probabilities. They are attached as children of a newly created temporary node t_0. This process is

i	1	2	3	4
Symbol x_i	A	B	C	D
Probability p_i	$\frac{5}{11}$	$\frac{3}{11}$	$\frac{2}{11}$	$\frac{1}{11}$
$H(x_i)[bit](approx.)$	1.14	1.87	2.46	3.46

Table 2.3.: Exemplary data source

repeated until only node t_2 remains. By looking at the example, it can be seen that the tree is built bottom-up, i.e., from the leaves to the root node, since the least probable symbols have the longest path to the root.

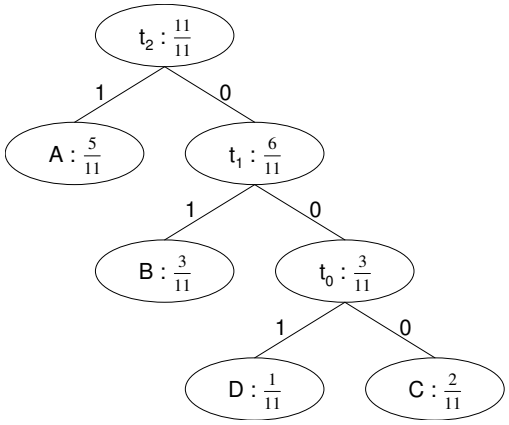


Figure 2.18.: Huffman Tree for the example shown in Table 2.3

Table 2.4 presents the resulting Huffman code that is derived from the Huffman Tree. The table contains the assigned replacement symbols r_i for each symbol x_i. These are obtained by concatenating the edge labels starting from the root of the tree until x_i is encountered. A property of Huffman codes is that they are prefix-free and uniquely decodable. This means that no r_i is the beginning of another r_j.

Symbol x_i	A	B	C	D
Replacement Symbol r_i	1	01	000	001

Table 2.4.: Resulting Huffman Code for the example shown in Table 2.3

Imagine that our exemplary data source emits the symbol stream S_1 = "A B C D A A A A A" (9 symbols). The resulting replacement code would be C_1 = "1 01 000 001 1 1 1 1 1" (14 bit)[5]. This gives an average code length $l = 14bit/9symbols \approx 1.56bit/symbol$ per symbol which is less than the above-calculated $H(X) = 1.79bit/symbol$. This is because S_1 does not exactly match

[5]The spaces are inserted for the sake of clarity, they are not part of the data

the symbol probabilities assumed in Table 2.3, for instance the symbol “A” occurs too often.

To decode the message, the exact knowledge of the Huffman Tree shown in Figure 2.18 is mandatory at the receiver. This is because the Huffman Tree itself is not unique, e.g., changing the edge labels from zero to one and vice versa yields an equally efficient, yet different Huffman Code. A frequently used technique is to prefix the replacement code with the used Huffman Tree. On the one hand, this technique allows an optimal adaptation of the Huffman Tree to the actual symbol probabilities of the data. On the other hand, it increases the length of the transmitted data. While this approach is well suited for lengthy symbol streams (e.g., when compressing files on a computer), it deteriorates the achieved compression ratio for short symbol streams (e.g., the payload of individual wireless data packets).

3. Comprehensive Development Support for WSNs

Software development for traditional embedded systems is already a challenging task in itself [96]. The unique characteristics and requirements of WSNs, which are in essence networked embedded systems, further aggravate this situation. Therefore, developing applications for these massively distributed systems with only very scarce resources requires new paradigms and development tools to counter the arising challenges. When analyzing typical real-world WSN projects, such as the ones presented in Section 2.1.1, three aspects are common to them [58, 99, 132]:

- Simulation and visualization are an integral part of WSN development
- Deployments comprise different, heterogeneous devices
- Software matures over time and changes must be anticipated

Application development for WSNs comprises a multi-stage approach and developers perform a number of sequential steps to arrive at a working application. Prior to any deployment on real sensor network hardware, simulations are conducted to ensure a correct application behavior under laboratory conditions. The outcome of the simulations is used for an evaluation of the application's performance and an optimization of the underlying algorithms and protocols. Once a satisfactory state is reached, the application is ported to the target hardware platforms including sensor nodes, gateways and backend systems. Therefore, device heterogeneity is an integral part of WSN application development. In addition, since neither simulation tools nor sensor nodes offer any user interface, application data originating from these components must be visualized by dedicated visualization code.

Apart from the challenges imposed by the implementation of simulation, visualization and applications, another issue complicates the development process. Since project specifications are subject to modification and applications evolve over time, changes are an inherent companion of the development process. Therefore, most of the above-mentioned phases are at least partially repeated multiple times. However, a common rule in project management is that changes are expensive, time consuming and error-prone the later they are introduced into the project.

For this reason, an optimized development process that supports simulation, visualization and heterogeneous, distributed application development is vital. Consequently, this chapter presents the architecture of such a development

framework for heterogeneous WSN applications.

The remainder of this chapter is structured as follows. Section 3.1 classifies standard development approaches and discusses how they are typically applied in WSN application development. Subsequently, Section 3.2 derives design criteria for a comprehensive development framework and introduces the architecture of our approach presented in this thesis.

3.1. State of the Art

Depending on the project's nature, the available resources and the experience of the development team, the chosen approach to software development varies. Figure 3.1 illustrates possible approaches as a continuum with the two oppositional approaches *manual, handcrafted code* and *model-driven engineering*. In this continuum, *code synthesis based on models* represents a compromise between both extremes.

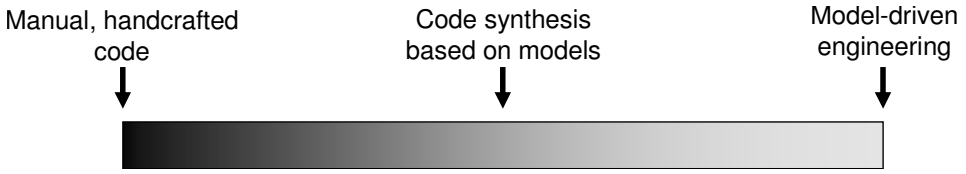


Figure 3.1.: Continuum of development approaches

In the case of *manual, handcrafted code*, the system is realized by using a high-level programming language such as Java, C, C++ or nesC. There is no intermediate modeling or abstraction phase but developers implement the application logic directly in the programming language of the target device. This process has proven advantageous for small projects with only a few developers. Being a dominant and prevailing approach to application development, a high degree of developer expertise is available in both, industry and research. In the meantime, a variety of *Integrated Development Environments* (IDEs) such as the Eclipse [172] framework are widely available and ease the development process.

However, it has been realized that the use of *code synthesis based on models* can improve the development process of complex projects. Users create graphical models that are utilized by a software tool to generate source code for different programming languages. The visual representation enables developers to gain a better understanding of complex data structures and the interdependencies of a project's components. The generated code provides method stubs that are filled with the application's logic by developers. A widespread, well-known technology in this context is the *Unified Modeling Language* (UML, [120]) and its *class* and *use case* diagrams.

While the above-mentioned approach still requires developers to implement the

application logic manually, *model driven engineering* strives for a fully automatic synthesis of the application from models. Hereby, models contain all details of the application and transform the abstract models into source code for a specific hardware platform. In contrast to model based code synthesis, developers (ideally) only modify the model without ever changing the generated source code. Probably the most commonly known representatives of this approach are the *Model Driven Architecture* (MDA, [121]) specified by the Object Management Group (OMG, [119]) and the *Eclipse Modeling Framework* (EMF, [171]) available as a plug-in for the Eclipse development environment.

In WSN application development, today's prevailing development methodology is the use of *manual, handcrafted code*. Despite the available programming abstractions (cp. Section 2.1.5 and Section 6.3), it is common practice that developers implement the application's logic directly in the programming language of the targeted sensor network hardware, transfer the compiled program onto the sensor nodes and test the deployed application. To improve this situation, developers use simulations prior to actual deployments in order to verify their protocols and algorithms. Depending on the simulation framework, this can result in a reimplementation of the application, e.g., when using Ns-2 for the simulations (implemented in C++) and TinyOS as the WSN operating system (implemented in nesC). This situation is only slightly improved by frameworks that support using the same implementation for both the WSN application and the simulation. For instance, TOSSIM [99] runs programs targeted at TinyOS hardware inside a simulated environment on standard PC equipment. While this approach combines simulation and application development, it is bound to specific sensor node hardware and therefore unable to address device heterogeneity or generic visualizations. A similar situation prevails in the area of visualization. Because the visualization of a network's state is highly specific to the individual applications, the visualization code is often implemented for one single application.

3.2. Design Goals and Architecture

Even though an ample selection of traditional model based code generation frameworks exist, writing code for sensor nodes, gateways, backend systems, simulators and visualization tools manually is still the predominant technique in WSN development. Providing tool support for the aforementioned tasks has the potential to ease and to accelerate WSN application development. However, the limited use of existing high-level frameworks motivates the need for a novel type of development support. We strongly believe that there is a demand for development tools that provide a sufficient layer of abstraction from the underlying heterogeneous hardware infrastructure in WSNs while leaving the developer in control of the overall process.

Such tools should ideally support the full span of development tasks leading ultimately to a deployable WSN application. To promote a widespread use,

they should support different WSN hardware platforms such as the ones presented in Section 2.1.4. Furthermore, this support must include infrastructure components such as gateways and backend systems while avoiding manual code duplication. We have identified five key features that a comprehensive development framework for WSNs must offer:

- Scale from resource-constraint WSN devices to more powerful gateways and high-performance backend systems
- Integrate simulation tools and visualization environments
- Encourage application evolution over its lifetime
- Support the integration with traditional networks
- Offering support for all classes of WSN applications

Consequently, we propose a development framework for WSN applications that follows these guidelines. It is comprised of the four components *Shawn*, *SpyGlass*, *Fabric* and *microFibre* that were briefly introduced in Section 1.3. While these components are contained and valuable in themselves, they unleash their full potential when they are embedded into a common development framework. Figure 3.2 presents the high-level architecture of our proposed approach. The central idea is to use *Fabric* as the integrating glue that amalgamates WSN application development, simulation and visualization into a common framework.

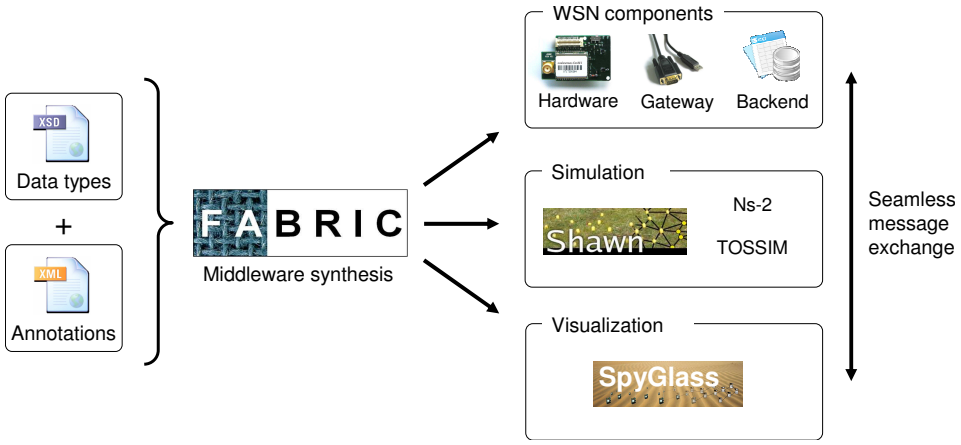


Figure 3.2.: Architecture to support a comprehensive development of WSN applications

The development process is started by defining the application's data types in a high-level language and augmenting this specification with *aspects* that define how the data type will be treated by the generated middleware (e.g., secure transmission or compact serialization). The specification is fed into *Fabric* along with a so-called *target specification* that defines properties of the generated code (such as target platform, programming language and device specific peculiarities). Invoking *Fabric* multiple times with the same annotated data types and different target specifications (e.g., for a specific WSN hardware

platform, Shawn and SpyGlass) yields compatible middleware instances that can seamlessly exchange messages.

The application developer uses the generated API to implement his application logic on top of the generated middleware instances. Compared to traditional WSN development, there is no need to propagate changes manually to the different implementations because modifications of the annotated data types manifest themselves in newly generated middleware instances automatically. This enables the application developer to concentrate entirely on the implementation of the application's logic and eases the integration of changes.

In the continuum depicted in Figure 3.1, this approach is located between *code synthesis based on models* and *model-driven engineering*. *Fabric*'s powerful type-specific annotation concept provides more services to the application developer that only method stubs. In contrast to *model-driven engineering*, *Fabric* does not generate the complete application from models. We are convinced that strict resource-constraints, device heterogeneity and fundamentally different programming paradigms such as cross-layer approaches still require manual optimizations by the application developer.

The remainder of this thesis is structured as follows. Chapter 4 introduces the WSN simulation tool Shawn and describes how the development of algorithms, protocols and applications for WSNs is supported by Shawn. Then, Chapter 5 presents the generic visualization framework SpyGlass and introduces the process of visualizing the output of simulations as well as that of real-world deployments. Next, Chapter 6 describes *Fabric* and *microFibre* in detail. Finally, Chapter 7 presents a case study and gives details on the application of these three components in a real-world project and Chapter 8 concludes with a summary and directions for future work.

4. Shawn: A Customizable Sensor Network Simulator

Software for WSNs must be thoroughly tested prior to real-world deployments since sensor nodes do not offer convenient debugging interfaces and are typically inaccessible after deployment. Furthermore, successfully designing algorithms and protocols for WSNs requires a deep understanding of these complex distributed networks. To achieve these goals, three different approaches are commonly used:

- Analytical methods
- Real-world experiments
- Computer simulations

Analytical methods are typically not well-suited to support the development of complete WSN applications. Despite their expressiveness and generality, it is difficult to grasp all details of such complex, distributed applications in a purely formal manner.

Real-world experiments are an attractive option as they are a convincing means to demonstrate that an application is able to accomplish a specific task in practice – if the technology is already available. However, due to the unpredictable environmental influences it is hard to reproduce results or to isolate sources of errors. Furthermore, real-world deployments are laborious and involve management tasks that are not directly related to the problem [63]. For this reason, they are typically limited to a few dozens of devices [165, 185], while future scenarios anticipate networks of several thousands to millions of nodes [44, 93].

Computer simulations are a promising means to tackle the task of algorithm and protocol engineering for WSNs. A number of simulation tools are available. They reproduce real-world effects inside a simulation environment including radio propagation properties and environmental influences. This mitigates required efforts for real-world deployments and may therefore help to increase their size. However, the high level of detail provided by these tools obfuscates and misses another, much more crucial issue: The large number of factors that influence the behavior of the whole network renders it nearly impossible to isolate a specific parameter of interest.

For example, consider the development of a novel routing protocol. In the case of a very low throughput, the cause of the problem is not clear at first sight, as the sources for the error are manifold: the MAC layer might be faulty; cross-

traffic from other senders could limit the available bandwidth; radio propagation properties might have changed or the routing protocol's algorithm is not yet optimal. Therefore, it is not only sufficient to simulate a high number of nodes with a high level of detail. Instead, developers require the ability to focus on the actual research problem. When designing algorithms and protocols for WSN it is important to understand the underlying structure of the network – a task that is often one level above the technical details of individual nodes and low-level effects.

There is certainly some influence of communication characteristics, e.g., because they affect transmission times, communication paths and packet loss. From the algorithm's point of view, there is no difference between a complete simulation of the physical environment (or low-level networking protocols) and the alternative approach of using well-chosen random distributions on message delay and loss. Thus, using a detailed simulation may lead to the situation where the simulator spends much processing time on producing results that are of no interest at all. By contrast, they actually hinder productive research on the algorithm.

To improve this situation, we propose a novel simulation tool called Shawn [49, 92, 127]. The central idea of Shawn is to replace low-level effects with abstract and exchangeable models so that simulations can be used for huge networks in reasonable time while keeping the focus on the actual research problem. In the following, Section 4.1 summarizes related work. Then, Section 4.2 discusses fundamental design goals of Shawn while Section 4.3 shows how these goals reflect themselves in Shawn's architecture. Finally, Section 4.4 compares Shawn's performance with two other prominent simulation tools (Ns-2 and TOSSIM).

4.1. Related Work

The range of applications for simulations is rather broad. Consequently, many different simulation tools have been developed. Each of them targets a specific application domain where it delivers best results. In the following, simulation tools frequently used in WSN research are presented. Figure 4.1 provides an overview of these tools and classifies their application area along two axes: abstraction level and the typical network sizes. Note that this does not express the maximal feasible network sizes, but rather reflects typical application domains.

Ns-2 The *Network Simulator-2* (Ns-2, [175]) is a discrete event simulator targeted at network research. Nowadays, Ns-2 is the most prominent network simulator used in WSN research [94]. It focuses on the simulation of ISO/OSI layers including energy consumption and phenomena on the physical layer. Ns-2 includes a vast repository of protocols, traffic generators and tools to simulate TCP, routing and multicast protocols over wired and wireless networks. It features detailed simulation tracing and includes the visualization tool *network animator* (nam) for later playback of the observed traffic. Support for sensor network simulations has also been integrated [117,123], including sensor models,

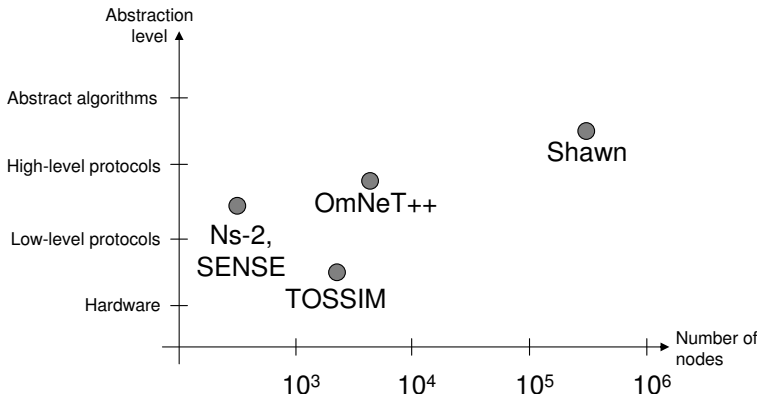


Figure 4.1.: Intended application area of Simulators

battery models, lightweight protocol stacks and scenario generation tools.

The highly detailed packet level simulations lead to a runtime behavior closely coupled with the number of packets being exchanged, making it nearly impossible to simulate large networks. Ns-2 is capable of handling up to 16,000 nodes but the detail level of its simulations render working with more than 1,000 nodes virtually impossible in terms of runtime and memory consumption.

OMNeT++ The *Objective Modular Network Testbed in C++* (OMNeT++, [177]) is an object-oriented, modular discrete event simulator. It is very similar to Ns-2 and also targets the ISO/OSI model. It can handle a few thousands of nodes and features a graphical network editor as well as a visualizer for the network and the data flow. The simulator is written in C++ and comes with a homegrown configuration language called *NED*. OMNeT's main objective is to provide a component architecture through which simulations can be composed very flexible. Components are programmed in C++ and then assembled into larger components using NED. It is free for academic use only and a commercial license is available.

SENSE The *Sensor Network Simulator and Emulator* (SENSE, [167]) is a simulator specifically developed for the simulation of sensor networks. The authors mention extensibility and reusability as the key factors they address with SENSE. Extensibility is tackled by avoiding a tight coupling of objects through a *component-port* model, which removes the interdependency of objects that is often found in object-oriented architectures. This is achieved by their proposed simulation component classifications, which are essentially interfaces, enabling the exchange of implementations without the need to change the actual code. SENSE offers different battery models, simple network and application layers and an IEEE 802.11 [69] implementation. In its current version, it provides a sequential simulation engine that can cope with around 5,000 nodes. Depending on the communication pattern of the network, this number may drop to

500. The authors plan to support parallelization of the simulations to increase the overall performance.

TOSSIM The *TinyOS mote simulator* (TOSSIM, [99]) emulates TinyOS [65, 100, 173] motes down to the bit level and is hence a platform specific simulator. It compiles code written for TinyOS to an executable file that can be run on standard PC equipment. It ships with a GUI (TinyViz), which can visualize and interact with running simulations. Recently, PowerTOSSIM [152], a power modeling extension has been integrated into TOSSIM. PowerTOSSIM models the power consumed by TinyOS applications and includes a detailed model of the power consumption of the Mica2 [33] motes. Using this technique, developers can test TinyOS applications without deploying them on real sensor network hardware. TOSSIM can handle scenarios with a few thousand virtual TinyOS nodes.

The crucial point of the above presented simulation tools is that each of them has its area of expertise in which it excels. Unfortunately, none of these areas happens to be high-level protocols and abstract algorithms in combination with the speed to handle large networks. This gap is filled by Shawn.

4.2. Design Goals

Shawn differs in various ways from the above-mentioned simulation tools, while the most notable difference is its focus. It does not compete with these simulators in the area of network stack simulation. Instead, Shawn emerged from an algorithmic background. Its primary design goals are:

- Simulate the effect caused by a phenomenon, not the phenomenon itself.
- Scalability and support for extremely large networks.
- Free choice of the implementation model.

Simulate the effects As discussed in Section 4.1, most simulation tools perform a complete simulation of the MAC layer including radio propagation properties such as attenuation, collision, fading and multi-path propagation. A central design guideline of Shawn is to *simulate the effect caused by a phenomenon, and not the phenomenon itself.* Shawn therefore only models the effects of a MAC layer for the application (e.g., packet loss, corruption and delay).

This has several implications on the simulations performed with Shawn. On the one hand, they are more predictable and there is a performance gain since such a model can be implemented very efficiently. On the other hand, this means that Shawn is unable to provide the same detail level that, for example, Ns-2 provides with regard to physical layer or packet level phenomena. However, if the model is chosen well, the effects for the application are virtually the same. Imagine two implementations of a MAC layer: One abstract implementation that yields

an increased packet loss on high local traffic and one that calculates interference for single packets using radio propagation models. Both will produce similar effects on the application layer.

It must be mentioned though that the interpretation of obtained results must take the properties of the individual models into account. If, for instance, a simplified communication model is used to benchmark the results of a localization algorithm, the quality of the obtained solution remains unaffected. However, the actual running time of the algorithm is not representative for real-world environments since no delay or loss occurs.

Scalability One central goal of Shawn is to support orders of magnitudes higher numbers of nodes than the currently existing simulators. By simplifying the structure of several low-level parameters, their time-consuming computation can be replaced by fast substitutes as long as the interest in the large-scale behavior of the system focuses on unaffected properties. A direct benefit of this paradigm is that Shawn can *simulate vast networks*.

To enable a fast simulation of these scenarios, Shawn can be custom-tailored to the problem at hand by selecting appropriate configuration options. This enables developers to optimize the performance of Shawn specifically for each single scenario. For example, a scenario without any mobility can be treated differently than a scenario where sensor nodes are moving.

Free model choice Shawn supports a multi-stage development cycle where developers can *freely choose the implementation model* as depicted in Figure 4.2. Using Shawn, they are not limited to the implementation of distributed protocols. The rationale behind this approach is that – given a first idea for a novel algorithm – the next natural step is not the design of a fully distributed protocol. In fact, it is more likely to perform a structural analysis of the problem at hand. To get a better understanding of the problem in this phase, it may be helpful to look at some exemplary networks to analyze their structure and the underlying graph representation.

The next step may be to implement a centralized version of the algorithm in order to achieve a rapid prototype version. A centralized algorithm has full access to all nodes and has a global, flat view of the network. This provides a simple means to obtain results and get a first impression of the overall performance of the algorithm in question. The results emerging from this process can provide optimization feedback for the algorithm design.

Once a satisfactory state of the centralized version has been achieved, the feasibility of its distributed implementation can be investigated. Since the aim of this step is to prove that the algorithm can be transformed to a distributed implementation, a simplified communication model between individual sensor nodes can be utilized. This allows for an efficient and fast implementation, yet with meaningful results.

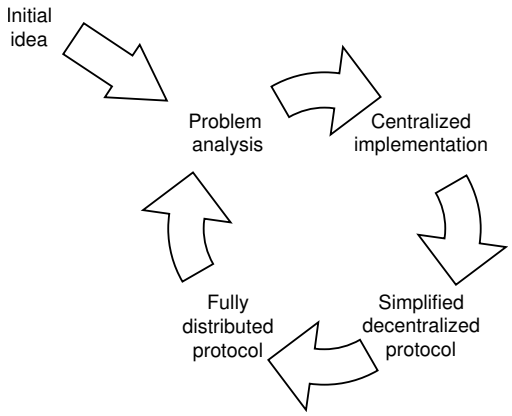


Figure 4.2.: Development cycle encouraged by Shawn

This simplified version can be transformed gradually into a protocol that actually exchanges network messages containing protocol payload. With the protocol and data structures in place, the performance of the distributed implementation can be evaluated. Interesting questions that can be explored are for instance the number of messages, energy consumption, run-time, resilience to message loss and effects of the environment. This development cycle helps the developer to start from an initial idea and gradually leads to a fully distributed protocol. However, each step of the cycle is optional and it is up to the developer to pick only the necessary ones.

4.3. Architecture

Shawn's architecture comprises three major parts (cp. Figure 4.3):

- *Models*
- *Sequencer*
- *Simulation Environment*

Every aspect of Shawn is influenced by one or more *Models*, which are the key to its flexibility and scalability. The *Sequencer* is the central coordinating unit in Shawn as it configures the simulation, executes tasks sequentially and controls the simulation. The *Simulation Environment* is the home for the virtual world in which the simulated sensor nodes reside. In the following, the functionality of these core components is described in detail.

4.3.1. Models

Shawn distinguishes between *models* and their respective implementations. A model is the interface used by Shawn to control the simulation without any

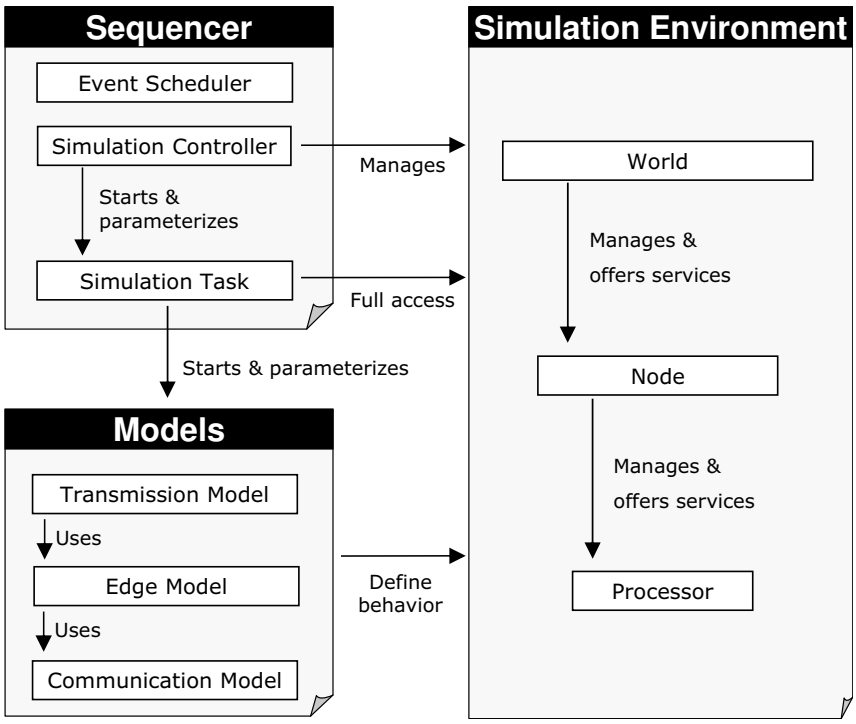


Figure 4.3.: High-level architecture of Shawn and overview of its core components

knowledge on how a specific implementation may look like. Shawn maintains a repository of model implementations that can be used to compose simulation setups by selecting the desired behaviors. The implementation of a model may be simplified and fast, or it could provide close approximations to reality. This enables the user to select the most appropriate implementation of each model to fine-tune Shawn's behavior for a particular simulation task.

As depicted in Figure 4.3, three models form the foundation of Shawn:

- *Communication Model*
- *Edge Model*
- *Transmission Model*

In the following, these models and their already included implementations are explained in detail. Other models of minor importance are briefly introduced at the end of this section.

Communication Model

Whenever a simulated sensor node in Shawn transmits a message, the potential receivers of this message must be identified by the simulator. Please note that this does not determine the properties of individual transmissions but defines

whether two nodes can communicate as a matter of principle. This question is answered by implementations of the *Communication Model*. Figure 4.4 presents the C++ interface of the *Communication Model*. A single method is invoked to determine whether the node b is in reach of the node a.

```
1 class CommunicationModel
2 {
3   ...
4   bool can_communicate_uni (Node& a, Node& b);
5   ...
6 };
```

Figure 4.4.: Application programming interface of the communication model in Shawn (excerpt)

By implementing this interface with user-defined code, arbitrary communication patterns can be realized. Shawn ships with a set of different *Communication Model* implementations that are shown in Figure 4.5.

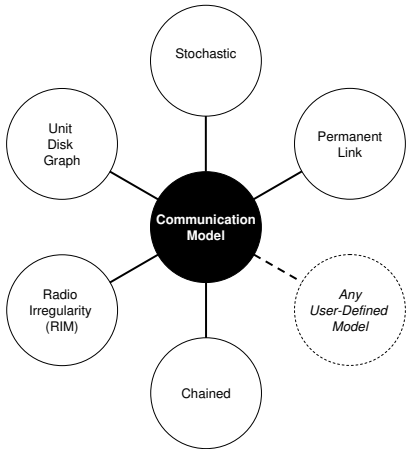
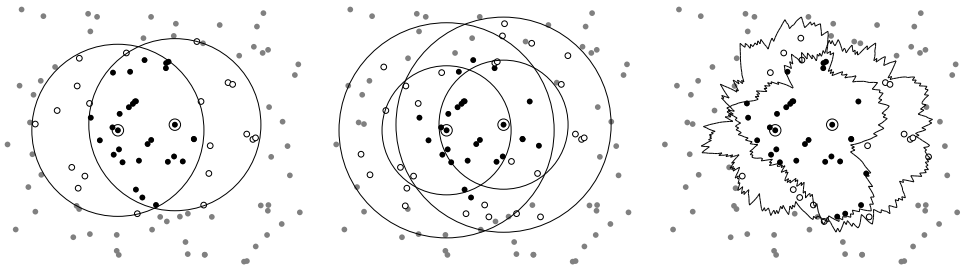


Figure 4.5.: Overview of Shawn's communication models

Three of these five implementations resemble communication patterns that are often used in WSN research. Figure 4.6 shows examples of how these models work. In the figure, the shared neighbors (filled black circles) of two nodes n_1 and n_2 (filled black circle with an extra black ring) are highlighted.

The *Unit Disk Graph* (UDG, cp. Figure 4.6(a)) radio model is based on the observation that the signal strength fades with the square of the distance from the sender. Given a minimum signal strength required for reception, two nodes can communicate bidirectional if the Euclidean distance d between the nodes is less than r_{max}. Regardless of its substantial abstractions from the real world, the model is widely utilized in WSN research because of its simplicity.

The *Quasi-Unit Disk Graph* (Q-UDG) radio model is a variant of the model introduced in [8]. It defines two new distances r_1 and r_2 with $r_1 < r_2$. For



(a) Unit Disk Graph (b) Q-UDG (c) Radio Irregularity Model

Figure 4.6.: Characteristics of different radio models

$0 < d < r_1$ and $d > r_2$, the behavior is equivalent to the UDG Model. For $r_1 \leqslant d \leqslant r_2$, the packet reception probability decreases linearly from 1 to 0. It therefore honors the fact that the probability of a successful reception diminishes with increasing distance. Figure 4.6(b) shows an example with $r_1 = 0.75 * r_{max}$ and $r_2 = 1.25 * r_{max}$.

Based on real-world experiments, the *Radio Irregularity Model* (RIM, [55, 199, 200]) proposes an angle dependant range between a minimum and a maximum communication range (r_{min} and r_{max}). A factor determines the maximum change in transmission range per degree and thus controls the irregularity of the shape (cp. Figure 4.6(c)). In contrast to UDG and Q-UDG, the RIM model also yields unidirectional links.

In addition to the real-world models, the *Permanent Link* model allows the specification of static links to pre-define communication relations such as wired connections to gateway nodes. The *Chained Communication Model* supports combining multiple communication models to a single communication model. For instance, while most of the sensor nodes in a network could use the *Unit Disk Graph* model, some gateway nodes have wired connections that are modeled by a *Permanent Link* model.

Edge Model

The *Edge Model* provides a graph representation of the network. The simulated nodes are the vertices of the graph and an edge between two nodes is added whenever the *Communication Model* returns true. To assemble this graph representation, the *Edge Model* repeatedly queries the *Communication Model*. It is therefore possible to access the direct neighbors of a node, the neighbors of the neighbors, and so on. This is used by Shawn to determine the potential recipients of a message by iterating over the neighbors of the sending node. Simple centralized algorithms that need information on the communication graph can be implemented very efficiently as in contrast to other simulation tools, no messages must be exchanged that serve as probes for neighboring nodes.

Figure 4.7 depicts the C++ interface that must be extended by implementations

of the *Edge Model*. In essence, two methods provide the ability to iterate over the neighbors of a node for a specific communication direction. The communication direction parameter defines how the property "neighbor" is interpreted and can be one of the following: *in*, *out*, *bidirectional* or *any*. If not specified, the communication direction defaults to "bidi".

```
1 class EdgeModel
2 {
3   ...
4   adjacency_iterator begin_adjacent_nodes (Node &,
5     CommunicationDirection d = bidi);
6
7   adjacency_iterator end_adjacent_nodes (Node &);
8   ...
9 };
```

Figure 4.7.: Application programming interface of the edge model in Shawn (excerpt)

The communication direction property is defined as follows: u and v are neighbors if for the communication direction

- "in" holds: $can\_communicate\_uni(v, u)$,
- "out" holds: $can\_communicate\_uni(u, v)$,
- "bidi" holds: $can\_communicate\_uni(u, v) \wedge can\_communicate\_uni(v, u)$,
- "any" holds: $can\_communicate\_uni(u, v) \vee can\_communicate\_uni(v, u)$.

Depending on the application's requirements and its properties, different storage models for these graphs are needed. For instance, mobile scenarios require different storage models than static scenarios. In addition, simulations of relatively small networks may allow storing the complete neighborhood of each node in memory. Conversely, huge networks will impose impractical demands for memory and hence supplementary edge models trade memory for runtime, e.g., by recalculating the neighborhood on each request or by caching a certain amount of neighborhoods. Accordingly, Shawn provides different *EdgeModel* implementations as shown in Figure 4.8.

The *Lazy* edge model is intended for simulations with only a small amount of nodes, hardly any communication or high node mobility. It does not store any information but recalculates the graph on the fly by querying the current *Communication Model*. The *Grid* edge model uses a two-dimensional grid for arranging nodes according to their geometric position. Therefore, the search for neighboring nodes is restricted to nearby ones, thus effectively improving the lookup speed while still fully supporting mobility. The *List* edge model stores the complete graph of the network in memory. This allows for a faster iteration over the neighboring nodes at the cost of a time-consuming initial construction and non-negligible memory demands. Since the list is only built once at the beginning of the simulation, this edge model does not support node mobility but

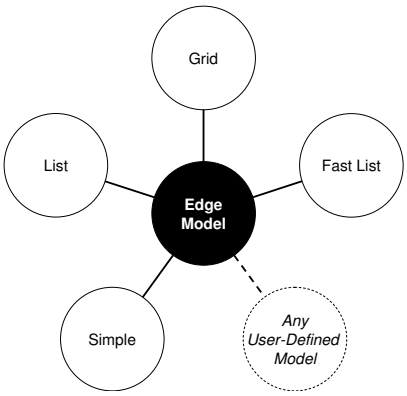


Figure 4.8.: Overview of Shawn's edge models

only static scenarios. The *Fast List* edge model combines the functionality of the *Grid* and *List* edge models into a single model implementation. Internally, it uses a *Grid* edge model for the initial construction of a contained *List* edge model. As a result, it provides the features of the *List* edge model plus a fast construction at the cost of a higher initial memory footprint.

Transmission Model

Whenever a node transmits a message, the behavior of the transmission channel may be completely different than for any other message transmitted earlier (cp. Section 2.1.3). For instance, cross traffic from other nodes may block the wireless channel or interference may degrade the channel's quality. To model these transient characteristics inside Shawn, the *Transmission Model* determines the properties of an individual message transmission. It can arbitrarily delay, drop or alter messages.

This means that a message may not reach its destination even if the *Communication Model* states that two nodes can communicate as a matter of principle and the *Edge Model* lists these two nodes as neighbors. Figure 4.9 shows the C++ interface for transmission model implementations in Shawn. The `send_message()`-method accepts a `MessageInfo` data structure containing the message itself, the time of transmission and the position of the sender.

Again, the choice of an implementation strongly depends on the simulation goal. In case that the runtime of an algorithm is not of interest but only its quality, a simple transmission model without delay, loss or message corruption is sufficient. Models that are more sophisticated could take contention, transmission time and errors into account at the cost of performance.

Figure 4.10 lists the built-in transmission models of Shawn, covering both abstract and close-to-reality implementations. The *Reliable* transmission model delivers all messages immediately, without loss or corruption to all neighboring nodes. *Random drop* is a slight variation in that it discards messages with a

```
1  class TransmissionModel
2  {
3          struct MessageInfo
4          {
5              Node* src_;
6              Vec src_pos_;
7              double time_;
8              MessageHandle msg_;
9          };
10   ...
11   void send_message (MessageInfo&);
12   ...
13 };
```

Figure 4.9.: Application programming interface of the transmission model in Shawn (excerpt)

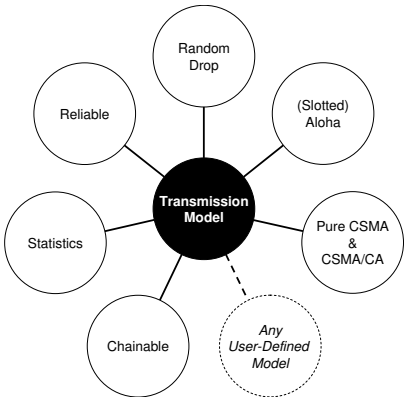


Figure 4.10.: Overview of Shawn's transmission models

given probability but it neither delays nor alters messages. The *Statistics* implementation does not deliver any message but instead records informational data such as the overall message count, different message types, etc. To make use of such a non-functional transmission model, the *Chainable Transmission Model* allows a series of transmission models to process a message sequentially. Like that, a message could first be counted, then delayed and may then be dropped by combining several simple transmission models.

Two additional implementations are closer to the real world than the above-mentioned ones. They simulate the effects of the well-known CSMA/CA and (slotted) Aloha [1, 32, 169] medium access schemes. Please note that not the MAC protocol itself is simulated but only the delay and loss characteristics are modeled for performance reasons.

Miscellaneous models

Besides these core models that shape the fundamental behavior of Shawn, a number of more specialized models provide data for the simulations. Currently, Shawn contains models for random variables, distance estimations and mobility.

Random variables are often needed in simulations to mimic uncertainty and randomness present in the real world. The *Random Variable* model introduces a layer of abstraction between the actual sources of random data and the application. As a result, algorithms can be tested with different underlying random variables without changes to the implementation. Sensor nodes often need distance estimations to other nodes to acquire context information such as their position (cp. Section 2.1). In Shawn, the *Node Distance Estimate* model provides these distance estimations, e.g. to support the evaluation of localization algorithms. Modeling arbitrary mobility is supported by so-called *Node Movements*. Implementations of this model provide the current position of a node when queried, e.g. by the communication model or an application. In order to allow an observation of the movement of nodes, listeners can register with the movements for location updates. Whenever a node is about to leave a previously supplied area given by a bounding box, it notifies the listener and obtains a new bounding box. This mechanism is used e.g. by the *Grid* edge model to adapt its internal status.

4.3.2. Sequencer

The sequencer is the control center of the simulation: it prepares the world in which the simulated nodes live, instantiates and parameterizes the implementations of the models as designated by the configuration input and controls the simulation. It consists of

- *Simulation Tasks*, the
- *Simulation Controller* and the
- *Event Scheduler*.

Simulation Tasks

Simulation Tasks are pieces of code that are invoked from the configuration of the simulation supplied by the user. They are not directly related to the simulated application but they have access to the whole simulation environment and are thus able to perform a wide range of tasks. Example uses are managing simulations, gathering data from individual nodes or running centralized algorithms.

Shawn exclusively uses tasks to expose its internal features to the user. A variety of tasks is included in Shawn that supports the creation and parameterization of new simulation worlds, nodes, routing protocols, random variables, etc. Even

the actual simulation is triggered using a task and the user can specify the amount of time that should be simulated upon the execution of this simulation task.

```
1 class SimulationTask
2 {
3   ...
4   void run (SimulationController&);
5   string name ();
6   string description ();
7   ...
8 };
```

Figure 4.11.: Application programming interface of a Simulation Task in Shawn (excerpt)

Simulation Tasks are configured and invoked by the *Simulation Controller* as discussed later on. They are identified and invoked using a unique name and parameters are passed as simple $(name, value)$-pairs. Figure 4.11 presents the C++ interface that a simulation task implementation must extend: One method that returns the unique identifying name, one that returns a human readable description and one that performs the actual task.

Simulation Controller

The *Simulation Controller* acts as the central repository for all available model implementations and runs the simulation by transforming the configuration input into parameterized invocations of *Simulation Tasks*. In doing so, it mediates between Shawn's simulation kernel and the user. In line with most other components of Shawn, the *Simulation Controller* can be customized by a developer to realize an arbitrary control over the simulation. The default implementation reads the configuration commands from a text file or the standard input stream. Figure 4.12 shows how commands are structured.

Two different line formats can occur: one that defines global variables and one that invokes and parameterizes Simulation Tasks. Line 1 shows how a global $(name, value)$-pair is specified which is valid from the point of its definition until the simulation run completes. Line 3 shows how a task is invoked by specifying its name (as returned by the task's `name()`-method, cp. Figure 4.11) separated with a trailing blank character. Following the tasks name, $(name, value)$-pairs may occur that are valid for the invocation of the task (local values temporarily overwrite global ones if their names are identical).

If a simulation task with this name is found, its `run()`-method is invoked and the current set of $(name, value)$-pairs is passed. Note that tasks are instantiated when Shawn starts and using the same task-name again results in invoking the `run()`-method on the same instance as for the first call. This can be used to collect data continuously and to evaluate it at a later point in time.

```
1 global_name1=global_value1
2
3 task_name name1=value1 name2=value2
```

Figure 4.12.: Plain-text file format used by Shawn's default Simulation Controller implementation

A second implementation allows the use of Java-language scripts to steer the simulation. While providing the same functionality, it allows more complex constructs and evaluations already in the configuration file. A comprehensive plain text configuration file is presented in Section A.1.1 and the semantically equivalent Java-language file is shown in Section A.1.2.

Event Scheduler

Shawn uses a discrete event scheduler to model time. The *Event Scheduler* is Shawn's timekeeping instance. Objects that need the notion of time can register with the Event Scheduler to be notified at an arbitrary point in time. The simulation always skips to the next event time and notifies the registered handlers. This process continues until all nodes signal either that they have powered down or until the maximum configured time has elapsed.

This has some performance advantages compared with traditional approaches that use fixed time intervals (such as a clock-tick every 1ms): First, handlers are notified only at the precise time that they have requested avoiding unnecessary calls to idle or waiting nodes that have no demand for processing. Second, users are not bound to some artificial granularity but an event may occur at the full precision that is offered by floating-point numbers. Figure 4.13 sketches how the *Event Scheduler* allows simulations to utilize this timing mechanism by using one of the several interaction possibilities.

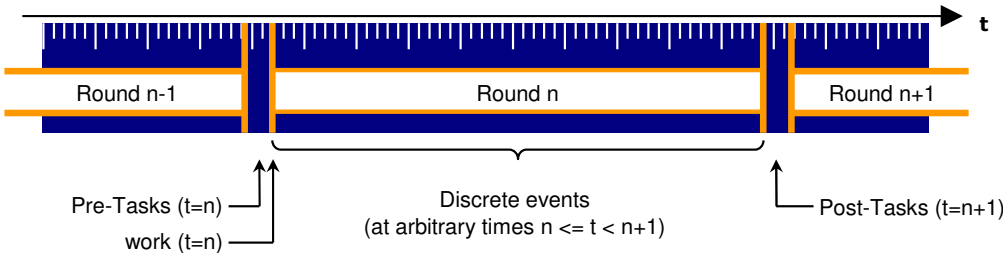


Figure 4.13.: Shawn's discrete event scheduler

Shawn logically arranges simulations into rounds ($r = 0, 1, 2, \ldots$). The user may register *Simulation Tasks* as pre-step and post-step tasks that are executed immediately before and after each of these rounds. This is useful to extract information from the simulation without the need to intermingle simulation code with code that analyzes the performance of the simulated algorithm.

At the beginning of each round, a node's `work()`-method is invoked. Applications can choose to use this method for an eased implementation when precise timing is not required. Apart from these fixed points in time, the nodes or other elements of the simulation may register an event at any point in time. Using these three distinct possibilities offers applications the required flexibility to integrate timing aspects into the simulation without degrading the overall performance.

4.3.3. Simulation Environment

The simulation environment is the home of the virtual world in which the simulation objects reside. As shown in Figure 4.3, the simulated *Nodes* reside in a single *World* instance. The Nodes themselves serve as a container for so-called *Processors*. Developers using Shawn implement their application logic as instances of these Processors. By decoupling the application inside a Processor from the Node, multiple applications can easily be combined in a single simulation run without changing their implementations. For instance, one processor could implement an application specific protocol while another processor gathers statistics data.

Figure 4.14 shows an excerpt of the Processor's API. After a processor has been instantiated, its `boot()`-method is invoked. A Processor can transmit messages by a call to `send()` and whenever a message for the Node is received, it dispatches this message to all its Processors by calling `process_message()`. As mentioned above, the Processor's `work()`-method is invoked whenever the Event Scheduler starts a new simulation round.

```
1 class Processor
2 ...
3   void boot( );
4   bool process_message( MessageHandle& );
5   void work( );
6   Node& owner( );
7   void send( MessageHandle& );
8 ...
9 };
```

Figure 4.14.: Application programming interface of a Processor in Shawn (excerpt)

A Node offers a number of services to the Processors that ease the implementation of algorithms and simplify the overall simulation task. As mentioned above, the *Edge Model* can be used to identify the neighbors of the current node. Unlike other simulators, Shawn does not restrict the user to a message-driven programming model but also allows "shortcuts", i.e., to execute method calls directly on other nodes. This is beneficial when implementing a centralized version of an algorithm or to get things done fast that are only a pre-condition for the current simulation, but not in the focus of the research.

It is often the case that an algorithm requires input that is produced by multiple (potentially very complex) other algorithms. To avoid waiting for the same results of these previous steps repeatedly in every simulation run, Shawn offers the ability to attach type-safe information to Nodes, the World and the Simulation Environment. Two Simulation Tasks (`load_world` and `save_world`) provide the ability to load Tags from and to save Tags to XML documents. Currently, three different Tag types exist: Simple Tags, Group Tags and Map Tags. Simple Tags contain a value of a certain type (e.g., `string`, `int` or `Boolean`), Group Tags contain other Tags and Map Tags contain pairs of values of a certain type. Figure 4.15(a) shows such an XML document that contains these different Tag types and Figure 4.15(b) shows how a Processor would access the Boolean Tag called `base_station`.

```
<Scenario>
 <snapshot id="0">
  <node id="runner12">
   <location x="135.0" y="5.0" z="0.0"/>

   <!-- Simple Boolean Tag -->
   <tag type="bool" name="base_station" value="false"/>

   <!-- Tag group containing other Tags -->
   <tag type="group" name="marathonnet">

    <!-- Map Tag that maps one floating-point value to another -->
    <tag type="map-double-double" name="split_times">
     <entry index="0" value="0"/>
     <entry index="21097.5" value="3801.0"/>
     <entry index="42195" value="7658.0"/>
    </tag>
   </tag>
  </node>
 </snapshot>
</Scenario>
```

(a) Shawn's XML persistence file

```
//Find the Tag
ConstTagHandle th = owner().find_tag("base_station");
//Cast to the expected type (error checking omitted)
const BoolTag* bt = dynamic_cast<const BoolTag*>( th.get() );
//Retrieve the Tag's value
bool is_base_station = bt->value();
```

(b) Extract a Boolean value from a Tag

Figure 4.15.: Exemplary use of Tags

A benefit of this concept is that it allows decoupling state variables from member variables in the user's simulation code. By this means, parts of a potentially complicated protocol can be replaced without code modification because the internal state is stored in tags and not in member variables of a special implementation.

To model sensors and their corresponding sensor values, a generic framework

called *Reading*s and *Sensor*s is provided. Readings deliver position-dependent and time-dependent values that can be arbitrarily typed. Sensors are bound to a specific sensor node and deliver sensor readings. Readings and sensors can be configured by the user and are referenced inside the simulation using unique names. This decoupling allows changing the underlying reading or sensor implementation without changing or recompiling the simulation code. In its current version, Shawn already provides some simple sensor and reading implementations, which obtain their values from XML files or *Tags*.

4.4. Evaluation

This section evaluates the performance and adaptability of Shawn. It first compares Shawn with Ns-2 and TOSSIM to demonstrate that Shawn can handle large networks at high speeds. Then Shawn's adaptability and flexibility is demonstrated at the example of the available *Edge Models*.

Comparison with Ns-2 and TOSSIM

Since the exchange of wireless messages is the key ingredient in wireless sensor networks, a simulator's ability to dispatch messages to their recipients determines the speed of simulations. In the following, measurements are presented that show the amount of memory and CPU time required to simulate a simple application that broadcasts a message every 250ms of simulated time. The communication range of the sensor nodes is set to 50 length units and each simulation runs for 60 simulated time units. The size of the simulated area is 500x500 length units. A number of simulations with increasing node count were performed. Therefore, the network's density increases steadily as more nodes are added to the scenario. This application has been implemented for Ns-2, TOSSIM and Shawn. All simulation tools were used as supplied by the source repositories with maximal compiler optimization enabled. The simulations were run on standard, state of the art i686 PCs.

Figure 4.16 depicts the required CPU time in seconds and Figure 4.17 shows the maximally used amount of RAM for the three simulation tools at different node counts. It should be noted that this kind of comparison is biased in favor of Shawn because the two other simulators perform much more detailed computations to arrive at the same results. It is more to be seen as an indication how application developers can benefit from using Shawn when these detailed results are not in the focus of interest. When interpreting the results, please note that these figures as well as the following ones use a logarithmic scale on both axes.

The first thing to notice is that Shawn outperforms both other simulation tools by orders of magnitude. Ns-2 hits the one-day barrier where Shawn is still finishing in less than one minute with a considerably smaller memory footprint. As mentioned above, this is because Ns-2 performs a very detailed simulation

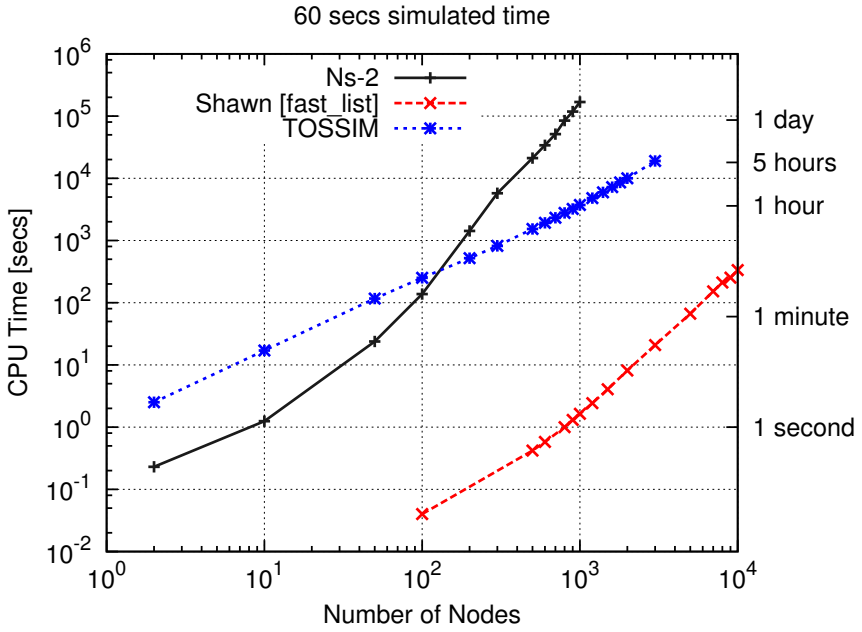


Figure 4.16.: Comparison of the required CPU time of Shawn, Ns-2 and TOSSIM

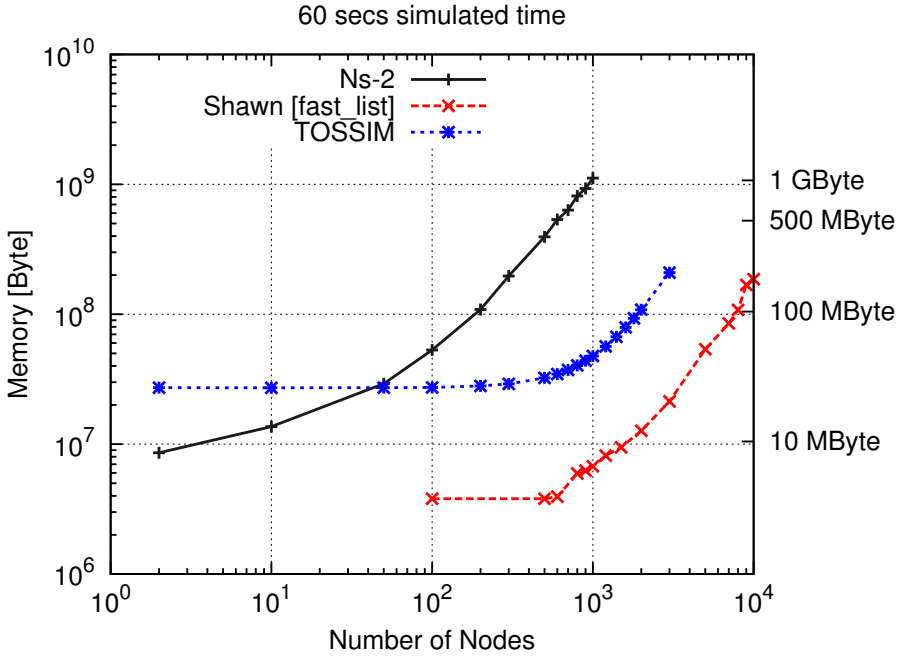


Figure 4.17.: Comparison of the required amount of RAM of Shawn, Ns-2 and TOSSIM

of lower layers such as the physical and the data link layer while Shawn simply dispatches the messages using a simplified model. Nearly the same situation applies to TOSSIM that simulates an underlying TinyOS-supported hardware platform. This clearly shows that Shawn excels in its area of expertise – the simulation of large-scale sensor networks with a focus on abstract, algorithmic considerations and high-level protocol development.

Adaptability of Shawn

As discussed in Section 4.3.1, Shawn's runtime behavior is heavily influenced by choosing a distinct implementation of one of the models (e.g., the Edge Model, Communication Model or the Transmission Model). Depending on the simulated scenario, a different choice may substantially alter the performance and the resource consumption of the simulation. Since selecting a specific implementation simply requires changing a value in the configuration file, users can select the best implementation for each simulation.

To demonstrate the pros and cons of the different edge model implementations, the above-described simulations were repeated using the same underlying scenarios using different edge models. Figure 4.18 shows the required CPU time for the different implementations and node counts. It is evident, that the *List* and the *Fast List* are faster than *Grid* and much faster than *Simple*. However, considering the memory consumption (cp. Figure 4.19), this performance gain comes at a certain price.

Because the size of the simulated world is fixed at 500x500 length-units with a communication range of 50 length-units, the density of the underlying graph – and therefore the amount of memory required for storing the neighbors of each node – increases significantly the more nodes are added to the scenario. A compromise between speed and memory consumption is offered by *Grid*.

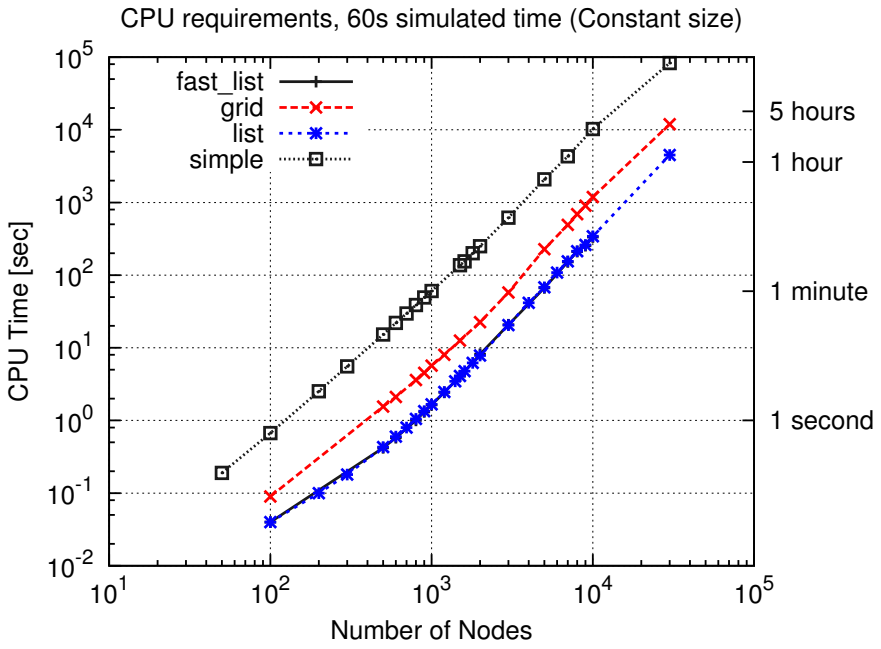


Figure 4.18.: Required CPU time of Shawn's edge models (Constant size)

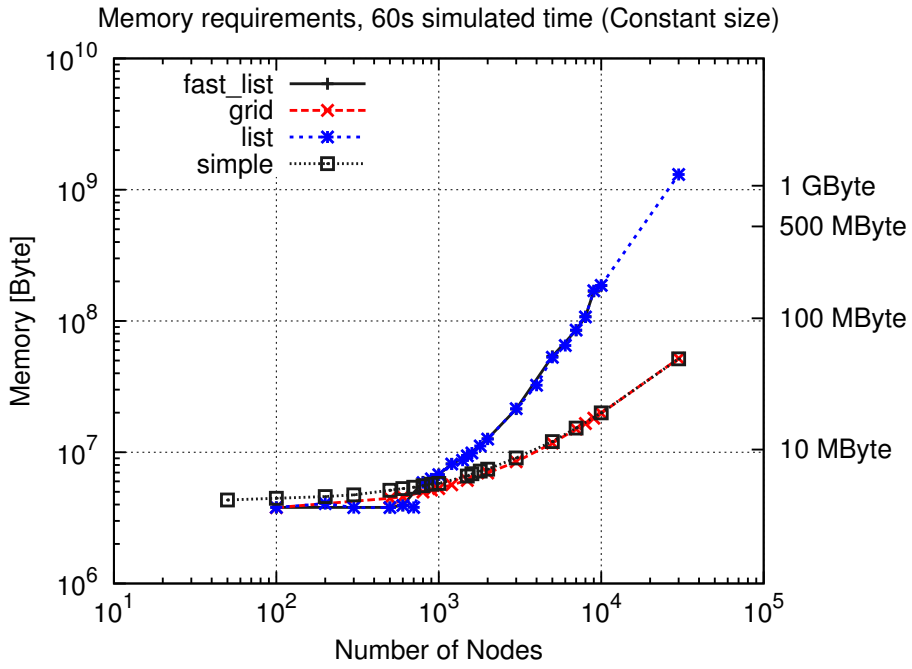


Figure 4.19.: Memory consumption for Shawn's edge models (Constant size)

However, this observation does not hold in general. Consider the situation where the number of nodes increases while the average density of the network remains constant. Figure 4.20 and Figure 4.21 depict the result of such simulations with a constant density (the density corresponds to the one of a scenario with 100 nodes in the simulations presented above).

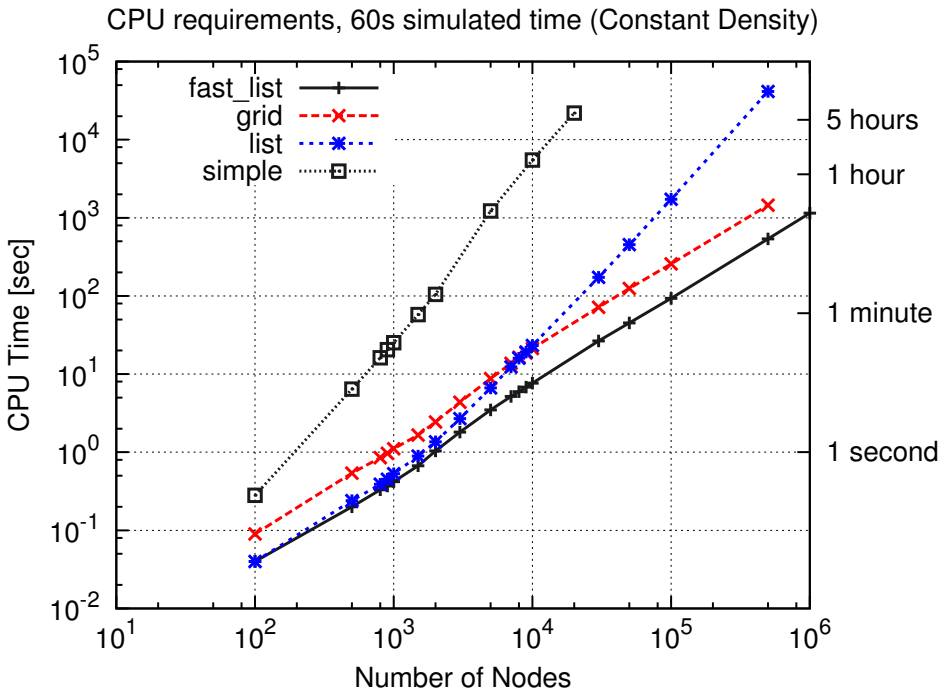


Figure 4.20.: Required CPU time of Shawn's edge models (Constant density)

When looking at simulations using *Fast List*, the runtimes are an order of magnitude lower than in previous case. However, the memory requirements differ only marginally (maximal difference 108.64 MByte, 9.94 MByte in average). This is because, in contrast to the previous scenario, the neighborhood sizes remain constant and the storage space required for keeping them in memory increases only slowly. As in the previous example, *Simple* is the slowest one and *Fast List* the fastest edge model implementation.

In this example, *Grid* outperforms *List* only for node counts larger than 10,000. This is because of the high initial setup cost of the *list* edge model. However, *Fast List* performs better for all node counts which is because it reduces the time required for construction of a *List* edge model by using a *Grid* internally to limit the search space for neighboring nodes.

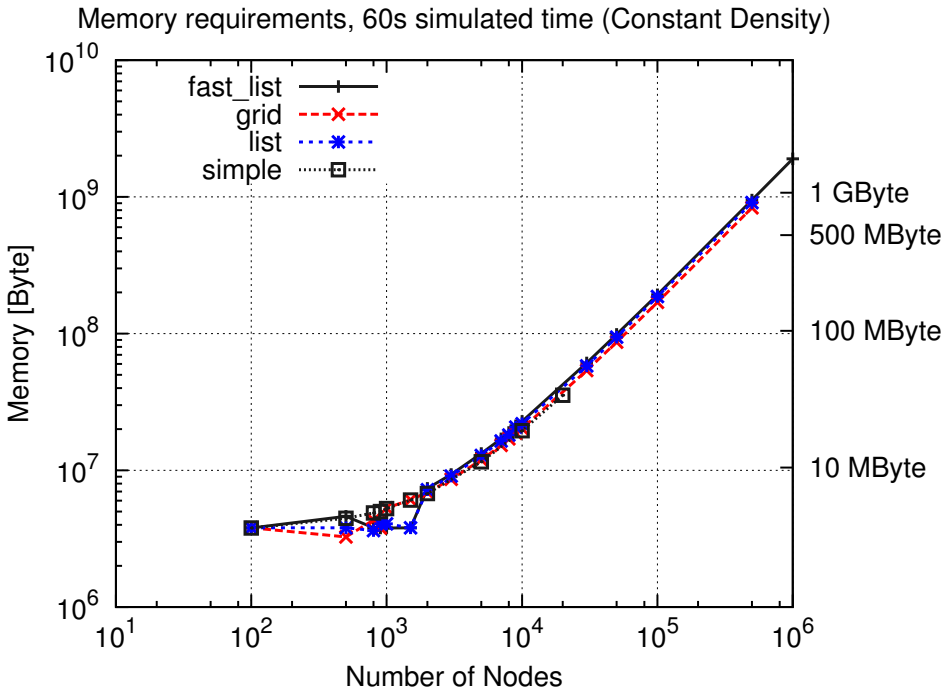


Figure 4.21.: Memory consumption for Shawn's edge models (Constant density)

Conclusion

The above-presented measurements show that Shawn's central design goal (simulate the effect caused by a phenomenon, and not the phenomenon itself) indeed leads to a high scalability and performance compared to traditional approaches to simulation. Furthermore, the results show that Shawn's runtime behavior can be custom-tailored simply by changing configuration parameters. This does not only apply to edge models, but also the other models used in Shawn have a major impact on the performance and resource consumption. Hence, an optimal selection of model implementations must take the properties of the scenario into account.

As a result, developers must carefully select the simulation tool depending on the application area. When detailed simulations of issues such as radio propagation properties or low-layer issues should be considered, Shawn is obviously not the perfect choice. This is where Ns-2 and TOSSIM offer the desired granularity. However, when developing algorithms and high-level protocols for WSNs, this level of detail often limits the expressiveness of simulations and blurs the view on the actual research problem. This is where Shawn provides the required abstractions and performance.

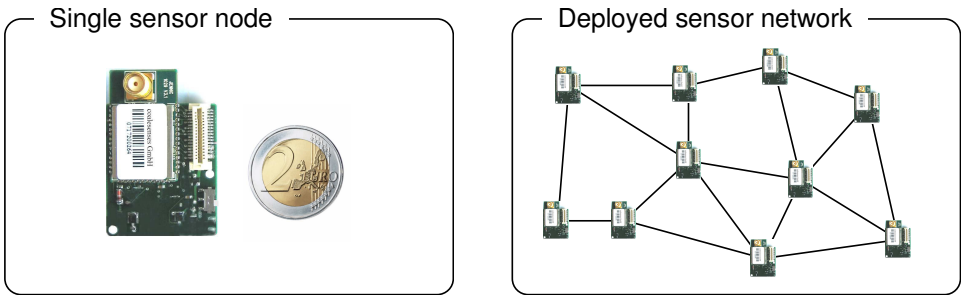
Shawn is currently in active development and use by several universities and companies to simulate wireless (sensor) networks. It was and continues to be an invaluable tool for over 15 research publications and more than 20 bachelor and

master theses. Shawn is also the enabling means behind the development of algorithms and protocols for WSNs in the SWARMS [50], the SwarmNet [47] and the EU-funded FRONTS [155] project. It supports recent versions of Microsoft Windows and most Unix-like operating systems such as Linux and Mac OS X. In addition, Shawn is easily portable to other systems with a decent C++-compiler. It is licensed under the liberal *BSD License*[1] and the full source code is available for download at `http://www.sourceforge.net/projects/shawn`.

[1] http://www.opensource.org/licenses/bsd-license.php

5. SpyGlass: A Generic Visualization Environment

Just like most other embedded devices, wireless sensor nodes typically do not offer any interface that a human could use to interact with these tiny appliances. As can be seen on Figure 5.1(a), there is no display or any other means to present information to the user - it is already luxurious if one or two miniature LEDs are available to convey even a slight amount of information. The user's possibilities for feeding data into a sensor node are also severely limited. Maybe one or two buttons allow the triggering of a diminutive selection of simple actions.



(a) State of a single node (b) State of n-nodes plus the network's state

Figure 5.1.: Situation for developers and end-users when developing and operating WSNs

However, even if they are available at all, in the majority of cases it will not be possible to use any of these in- or output possibilities. Once the network is deployed, it may span a large geographical region. If this region is accessible by humans at all, keeping track of every blinking diode or connecting to individual I/O-pins to determine the network's state is just not feasible (cp. Figure 5.1(b)). Nearly the same situation applies to simulators for WSN applications. As discussed in Chapter 4, they focus on a fast and easy simulation of sensor network applications and typically do not offer any kind of visualization environment. This is because the evaluation of results is strictly application specific and simulators are therefore considering this to be a task of the user.

Therefore, it is difficult to grasp the internal state of nodes or to influence this state from the outside. For that reason, visualization is a key element to develop and manage these networks: On the one hand, a visualization environment acts as a single interface to a network that may be comprised of thousands of tiny sensor nodes. On the other hand, visualization is a perfect means for humans

to understand complex data because of the brain's exceedingly sophisticated image processing capabilities. To support developers and users, many specialized toolkits have been proposed for wireless sensor networks that facilitate the visualization of the network's state as well as feedback mechanisms. However, as discussed later in Section 5.1, none of these tools provide the generality or the flexibility to support heterogeneous WSNs, simulation tools or arbitrary, user-defined visualization demands.

This chapter presents *SpyGlass* [25, 28, 29], a generic visualization environment for wireless sensor networks. Its modular architecture allows users a visualization of the network's state, the outcome of simulations or arbitrary other data. Furthermore, SpyGlass enables the user to feed data back into the network to influence its state. Because of this generic approach, SpyGlass is a flexible and comprehensive toolkit for the visualization and control of WSNs. To support this conjecture, the following paragraphs introduce fundamental properties of visualization environments and Section 5.1 assesses related work with these criteria in mind.

Most of the following properties are strictly technical, but probably the most important issue is a non-technical one: a generic visualization tool must accommodate the specific needs of different target audiences. On the one hand, developers of WSN applications are experts in the domain of protocol design, embedded programming and have an in-depth knowledge on many networking aspects. On the other hand, end-users of WSN products are not interested in technical details of the solution but need a task-oriented way of interacting with the sensor network.

Consider the development and deployment of an application that helps monitoring a forest fire. When implementing such an application, developers face a multitude of challenges such as erroneous sensor readings, incorrect self-localization of nodes or issues with the routing algorithm. Here, visualization allows for a quick localization of errors by visualizing the internal state of single sensor nodes or the topology of the overall network. However, a fire fighter in a control room needs a completely different type of visualization. Here, optimally dispatching the available resources in order to extinguish the fire quickly is the paramount goal. Therefore, a comprehensive visualization solution should support the complete lifecycle of a sensor network starting from the development by researchers up to the final deployment phase at a customer's location.

On the technical side, the support should not be limited to a single operation system, hardware platform, architecture or programming language. As discussed in Section 2.1.4 and Section 2.1.5, the selection of a hardware platform and programming model strongly depends on the specific problem that a sensor network should solve. Hence, it is not practical to employ a different visualization tool for each distinct combination of a hardware platform, operating system, etc. The goal is to support heterogeneous input sources to enable a plethora of application scenarios. Hence, it should be of no importance whether data originates from deployed sensor networks, files on a computer or

even simulations.

In addition, visualizing the network's state is usually only a by-product and therefore not in the centre of interest, since the developer's focus is to arrive at a working application. Therefore, the visualization code that is running inside the sensor network must be as lean as possible and should require only a minimal amount of the node's and the network's resources. For instance, if the throughput of novel routing algorithm is to be visualized, the traffic generated by the visualization code is non-negligible and must be considered in the evaluation.

Another crucial property is its ability to realize application-specific visualizations. This is especially important as the possible visualization tasks are at least as diverse as the application scenarios for WSNs (cp. Section 2.1.1). It is therefore essential that a visualization tool is flexible and can be customized and extended by a developer in order to visualize arbitrary data in a user-defined manner.

The remainder of this chapter is organized as follows. Section 5.1 discusses related work, pinpoints the drawbacks of existing tools and highlights how SpyGlass improves the current state of the art. Subsequently, Section 5.2 presents the architecture of SpyGlass and Section 5.3 demonstrates how custom functionality can be implemented. Section 5.4 concludes this chapter with a summary of the SpyGlass visualization environment.

5.1. Related Work

This section presents visualization tools for wireless sensor networks. In the following the aforementioned properties are used – where applicable – to evaluate these tools. To bear in mind, the key properties for a generic WSN visualization tool are the following:

1. Flexible visualization for different audiences (developers, end-users, ...)
2. Support for heterogeneous input sources (deployments, simulations, ...)
3. Small resource consumption (computation, communication and storage)
4. Extensibility and customizability

The *Surge Network Viewer* [35] and the *Mote-VIEW Monitoring Software* [34] are commercial products from Crossbow to visualize wireless sensor networks. The Surge Network Viewer features topology and network statistics visualization as well as logging of sensor readings and displaying the logged data. The statistics function includes the end-to-end data packet yield and the RF link quality, but is limited to these features. The system is not extensible and as a result, custom-made visualizations are not feasible. The Mote-VIEW Monitoring Software covers essentially the same topics but presents a cleaner user interface and more features. It is capable of logging wireless sensor data to

a database and to analyze and plot sensor readings. It allows querying the sensor network for collected data in a database-like manner, thus hiding the distribution of the data in the WSN. Both tools are tightly coupled to the Mica hardware platform and consequently, they support neither heterogeneous WSNs nor the visualization of simulations.

The *TinyViz* visualization tool is an integral part of *TOSSIM*, the TinyOs [65, 100, 173] simulator. TinyViz has been developed to ease the debugging of applications by visualizing the state of running TOSSIM simulations as well as to interact with these simulations. Out of the box, it visualizes sensor readings, LED states, radio links and message exchanges. TinyViz is easily extensible through a plug-in mechanism and plug-ins can supply arbitrary visualization code. They may also react to events that are generated by the simulator. However, it is also limited to the TinyOS software and the Mica hardware series and can only visualize the state of running simulations.

The aim of the *TosGUI* [114] project was to develop a simple graphical front end for TinyOS simulations similar to TinyViz. It is limited to visualizing the network's topology and lacks support for user defined visualizations. As a result, TosGUI offers only a subset of the features of TinyViz and any further development of the project seems to have stopped.

iNSpect [94] supports the visualization of trace files that are the result of Ns-2 [175] simulations. By parsing mobility and trace files of Ns-2 and a proprietary file format, a fixed set of visualization primitives is displayed on a 3D-canvas. This comprises the visualization of the network's topology, the success and failure of message transmissions and the routes of individual packets. iNSpect offers a number of configuration parameters to adjust the look and feel of the visualizations but does not offer a generic extension interface to add user-defined visualization code and is therefore restricted to a few application scenarios. Similar to TinyViz, iNSspect is primarily intended to visualize the output of a single, specific simulator.

Initially designed to visualize the state of a single WSN application scenario, *MonSense* [132] has been extended gradually whenever the authors realized a demand for new features in one of their projects. In its current version, MonSense supports the visualization of exactly these TinyOS applications and displays technical information such as node position, the communication graph and sensor readings. However, the authors report that a plug-in system is planned to allow an extension of the system's built-in visualizations. At present, MonSense tightly integrates TinyOS specific code and it is limited to the Mica hardware platform.

Ringwald et al. [140, 141] present an approach to monitor and debug a deployed WSN. An additional *deployment support network* sniffs the emitted packets and uses a second wireless interface to forward the collected data packets to a central gateway. The benefit of this approach is that it does not require any changes to the actual application and that it does not use any resources of the deployed network. Although not strictly a visualization tool, it presents an

interesting means to collect data passively from a WSN thus avoiding potential bugs introduced by the visualization code.

By contrast, *Sympathy* [135] collects various metrics inside the sensor network application to detect and debug failures. A number of heuristics may be integrated into a TinyOS application in order to detect critical operating conditions. Problem reports are then forwarded to a sink where developers may use this information to fix a potential error. A similar tool called *Sensor Network Management System* (SNMS, [174]) allows to query the state of TinyOS applications over the network. Equipped with a separate network stack, SNMS runs in parallel to the actual sensor network application. The rationale behind this approach is that SNMS is still functional even when the TinyOS application has crashed. Each TinyOS module can export a number of parameters and the user of SNMS can choose which of these parameters will be available at run-time, thus allowing the user to trim the resource consumption of SNMS to the actual demands.

To match the above-listed requirements, a comprehensive, generic visualization environment for wireless sensor networks must comprise three distinct building blocks: data collection and dissemination inside the WSN, gateways to transit networks as well as the final visualization. To support heterogeneity, each of these building blocks must be easily exchangeable and extensible thus allowing users to adapt the visualization environment to the needs of their specific application.

None of the above-mentioned projects covers all of these requirements: some of the projects are dedicated to support a specific hardware platform or a single simulator, or they merely focus on the extraction and dissemination of events inside the sensor network. Other tools that visualize the state of the network do not offer enough flexibility for user-defined extensions or customizations. This is exactly where SpyGlass improves the current state of the art. It provides an environment that is open for any hardware platform or simulation tool and provides the required flexibility to implement arbitrary user-defined visualizations.

5.2. Overview and Architecture

SpyGlass supports developers in visualizing the state of a sensor network and provides a feedback mechanism to influence this state. The information that is visualized may originate from an arbitrary data source, e.g., a deployed sensor network or a simulator such as Ns-2 or Shawn (cp. Chapter 4). Furthermore, the way in which data is visualized is also entirely up to the user.

SpyGlass is modular, for the most part generic and provides an environment that allows developers to integrate custom functionality. SpyGlass is therefore not bound to any specific hardware platform, simulator, programming language or predefined set of visualization primitives. Figure 5.2 shows the high-level

architecture and the three primary functional entities: One or multiple (real or simulated) sensor networks, a transit network and the visualization component.

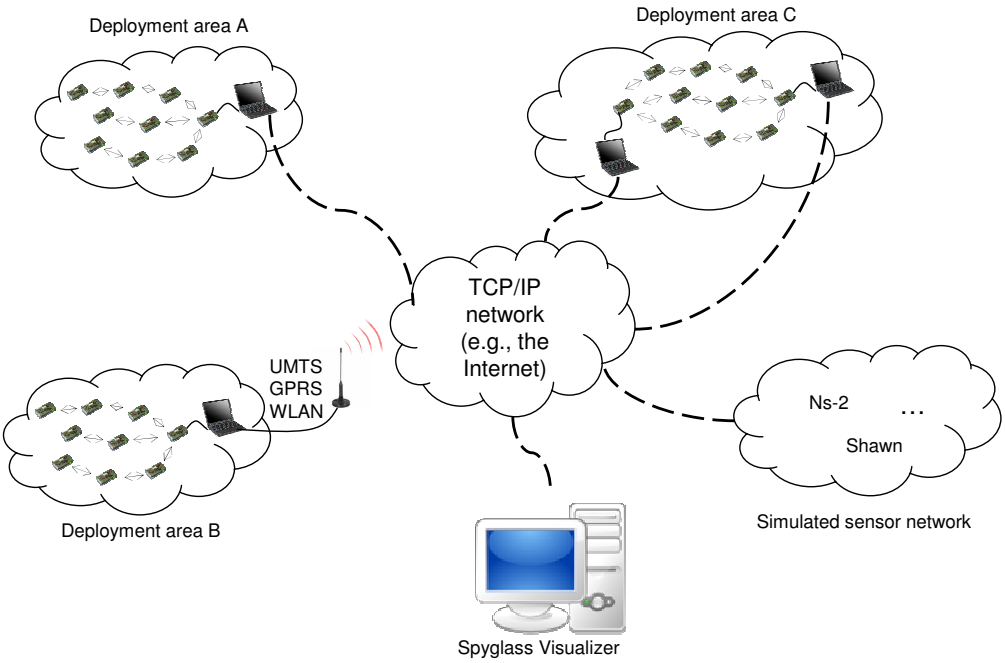


Figure 5.2.: Information from the sensor network is forwarded to a gateway and then transferred to the visualizer

The following subsections present different aspects of the SpyGlass architecture in detail. Section 5.2.1 introduces how visualization and control data flows from and to the sensor network. Then, Section 5.2.2 describes the structure of the core visualization component. Section 5.2.3 explains how so-called plug-ins use this component to display arbitrary visualizations and Section 5.2.4 presents instances of these plug-ins that are already contained in SpyGlass.

5.2.1. Data Flow

As indicated on Figure 5.2, data is routed from any TCP/IP-enabled component towards an instance of the SpyGlass Visualizer and vice versa. This component may be a computer connected to a sensor node, a simulation tool such as Shawn or a user-defined one. In the following, the term *gateway* is used to refer to such a component. One or more SpyGlass visualization instances that are also connected to the same TCP/IP backbone network (such as the Internet) can hence exchange data packets with these gateways. SpyGlass is therefore not bound to any specific WSN hardware platform and can visualize data from arbitrary sources given that the gateway is connected to the same TCP/IP network, e.g., by using a technology such as LAN, WLAN, GPRS or UMTS.

Developers can extend the set of built-in gateways to support arbitrary hard-

ware platforms, simulation tools, etc. Implementing a custom gateway is straightforward: it must accept TCP/IP connections and must conform to a common data format used in the TCP/IP-stream from and to the SpyGlass visualization instances. Figure 5.3 depicts the common data format that is used to split the stream of bytes into individual data packets. Hereby, each data packet is prefixed and suffixed with a special ASCII character sequence. The prefix consists of the DLE, STX (ASCII characters 0x10 and 0x02), the suffix of the DLE, ETX (ASCII characters 0x10 and 0x03) character sequence. Because of its special meaning, each DLE character that occurs between the prefix and the suffix is replaced with DLE, DLE to void its special meaning when it occurs inside the payload. The receiver can then replace every DLE, DLE sequence received between the prefix and the suffix with a single DLE character. By convention, the first byte of the payload indicates the type of the packet.

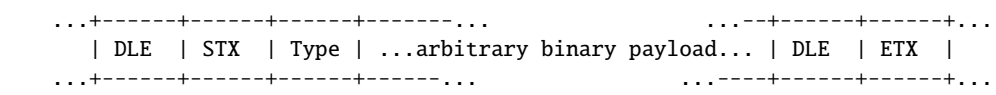


Figure 5.3.: Protocol format used in the TCP/IP stream

Currently, SpyGlass already ships with three different gateway implementations:

- Gateway and WSN implementation supporting real-world sensor networks using the ESB, pacemate and iSense nodes (cp. Section 2.1.4)
- Gateway code for the Shawn and Ns-2 simulation tools (cp. Chapter 4)
- A replay-gateway that reads data packets from a pre-recorded file

The support for real-world sensor networks consists of two distinct software entities: one that runs on the sensor nodes and one that runs on the gateway PCs. Inside the sensor network, each individual sensor node collects data using its sensors, derives new information from calculations or communicates with neighbors. If new information is generated, it is forwarded to one or more gateway nodes using an arbitrary routing mechanism. The current implementation uses straightforward flooding to provide reliability by redundant transmissions. Another option is to use sophisticated routing protocols to minimize the generated in-network traffic. Once the data packets have arrived at a gateway node, it passes the received packets to an embedded PC using a serial connection. The gateway software running on this PC has a ring buffer that stores a configurable number of data packets received from the gateway node. Once a SpyGlass visualization instance connects to this gateway, it delivers the contents of its ring buffer. This allows to bridge the time of transit network failures or to provide data to visualization stations that connect at a later point in time.

The gateway implementation for the two simulation tools Shawn and Ns-2 provides essentially the same service as the gateway component for the embedded PCs; only the source of data is different. Instead of receiving data packets from a gateway node via the serial link, this implementation offers an application programming interface (API) to the simulation code. Developers pass the bi-

nary payload including the type indicator to this API. The same applies to the implementation that reads the data packets from a file. This also represents a different source for the data packets and the following procedure is like with the implementation for the embedded PC.

All data packets flowing through the sensor network, from the gateway node to the attached PC, and through the transit networks to the visualization PC have the same payload format. Independent of the contained information (e.g., sensor readings, calculated data, internal sensor state) they consist of a data type indicator and the actual data. Using this simple format, developers can easily come up with new data types, which will be immediately supported by the sensor network and the gateway software. Up to this point, data has only been forwarded and all processing and display tasks are performed by the visualization software. Using this architecture and data format makes it possible to replace each of the three components individually, since the communication between them follows a well-defined packet format.

5.2.2. The Visualization Component

The graphical user interface of the visualization component (cp. Figure 5.4) consists of three major components: a graphical display canvas (on the upper left), a sidebar for tree-structured textual information on the network as a whole (on the upper right), and a display for line-based output, e.g., for debugging purposes (at the bottom).

Figure 5.5 shows the architecture of the visualization component of SpyGlass. It is comprised of a network information dispatcher, several plug-in containers and the canvas. The network information dispatcher receives data packets from potentially multiple sources such as gateways, simulators or files and distributes them to the subscribed plug-ins. These then convert the received binary data packets into drawing instructions for display on the canvas or into management information used by other plug-ins. The graphical display canvas consists of three layers:

- Background layer
- Relation layer
- Node layer

The *background layer* is used for painting the background of the visualization. This can for instance be useful to display a map of the environment, a plain white background or complex data visualizations derived from sensor readings. The *relation layer* is used for displaying all kinds of relations between nodes. The *node layer* is used for displaying the actual nodes.

The actual visualization is done by user-written plug-ins, one for each visualization demand. When a number of visualization plug-ins independent of each other shall cooperate, an important issue is to make sure that painting opera-

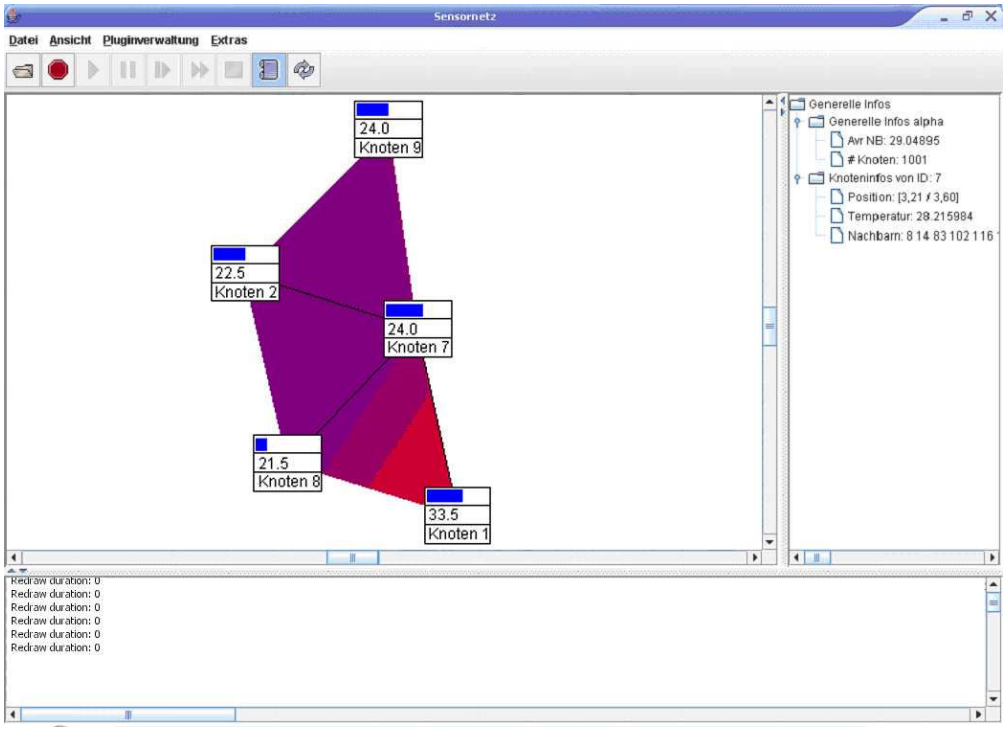


Figure 5.4.: The graphical user interface of the visualization component

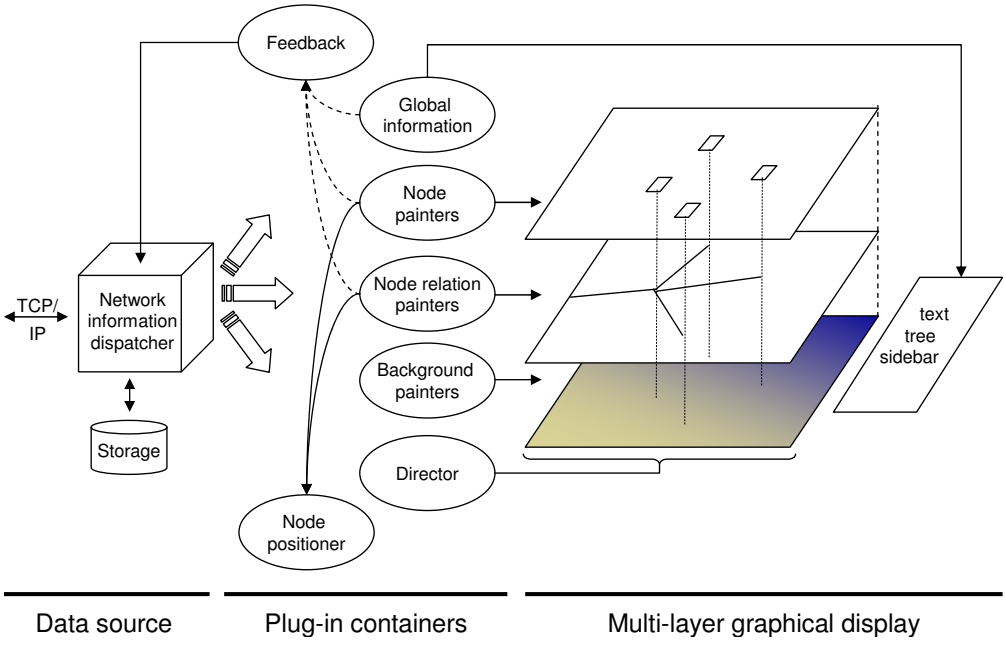


Figure 5.5.: The architecture of the visualization component

tions do not collide. To solve this problem while keeping configuration simple, SpyGlass features different plug-in types, each type corresponding to one of the display layers and being only allowed to paint on their layer. The following plug-in types can be extended by users to integrate custom visualization code:

- Background Painter
- Node Relation Painter
- Node Painter
- Global Information

Background Painter plug-ins draw the background of the visualization canvas. Examples of usage scenarios are drawing a simple white background, sketching a map of the deployment area, displaying satellite images or showing a coordinate system. Another interesting use is to illustrate spatial phenomena, which can be inferred from the received sensor data and positions, such as temperature maps [24].

Node Relation Painter plug-ins display arbitrary relations between sensor nodes on the canvas. This plug-in type can be used to display the sensor network's topology, e.g., by using lines to indicate existing communication links. Node relation painters are not limited to communication relations, but they can also highlight important aspects such as group memberships or routing paths.

Node Painter plug-ins draw the actual nodes and additional information onto the canvas. The depiction may be dependent on the node type (e.g., gateway node, cluster head, etc.), and may comprise a symbol representing the node, as well as additional textual or graphical information arranged around it. Different plug-ins can augment the visualization of a sensor node for example with the node's battery state, current sensor readings or other state information.

Global Information plug-ins display information about the network in a textual way. This information is displayed in a sidebar and is structured as a tree. This sidebar is particularly useful for displaying data that is not directly related to a single node or a specific group of nodes but to the network as a whole. Examples are the overall number of nodes, average neighborhood degree, etc.

Apart from these plug-ins that paint on the canvas, there are three other types of plug-ins which are not directly involved in the visualization process:

- Node Positioner
- Feedback
- Director

As indicated on Figure 5.5, the *Node Positioner* plug-in is used by other plug-ins to determine the location of individual nodes. This is used for instance to paint the nodes and relation endpoints on the canvas. The locations provided by the Node Positioner can be based either on location estimates/measurements received from the sensor network or on strategies based on graph theoretical

calculations that optimize the screen representation. Like this, the actual depiction of nodes is decoupled from the positioning of the node representation, so both Node Positioner plug-ins and Node/Relation painters can be replaced independently. In contrast to the canvas drawing plug-ins, only one plug-in instance of this type can be active at any given time.

Feedback plug-ins can be invoked by the canvas drawing plug-ins. This type of plug-in influences the state of the sensor network by sending data packets back into the sensor network. One usage scenario is to set the verbosity level of the SpyGlass code that runs inside the sensor network application, offering the user a control on the resource consumption and visualization granularity of SpyGlass. Other plug-ins may choose to open a dialog box where users can enter data that is then sent back into the sensor network.

Director plug-ins automate the process of zooming, panning and rotating the current view. Unlike SpyGlass, other tools rely on the user's input to change the current view, e.g., to zoom and pan into an area of interest. However, imagine a display in a firefighter's control room. Here, an operator should not be responsible to survey the whole area manually at any time. Rather, the visualization environment should direct the operator's attention to a specific area of interest where abnormal incidents have been observed by the sensor network. SpyGlass allows these plug-ins to automatically move a virtual camera. Similar to the Node Positioner plug-in, only one director may be active at any time.

5.2.3. Drawing and Plug-In Architecture

As already indicated above, SpyGlass features a flexible drawing and plug-in architecture. Most of its inner components can be exchanged or extended easily. This extensibility has its roots in how plug-ins and drawing instructions are implemented. Figure 5.6 illustrates how data originating from a sensor network or some other data source is processed until it is finally visualized. First, data arrives at the network information dispatcher as binary packets that contain arbitrarily structured information. Then, these data packets are distributed to all plug-ins that have subscribed for this type of packet.

The plug-ins do not directly draw on the canvas, but instead they use a set of drawing primitives available in SpyGlass. Using these primitives results in the assignment of graphical objects (such as lines, rectangles, text, etc.) with the layers. The three canvas drawing plug-in types (Node Painter, Relation and Background plug-ins) have their own layer on the canvas on which their plug-ins exclusively draw. The user can change the order and visibility of the plug-ins within each plug-in container to achieve an optimal presentation of the data. This architecture has several advantages: First, drawing on different layers avoids conflicts between plug-ins that have different priorities (e.g., drawing starts with the background, then the relation between the nodes follow and finally the nodes are visualized). Second, the painting code in the plug-ins is independent from the actual canvas implementation.

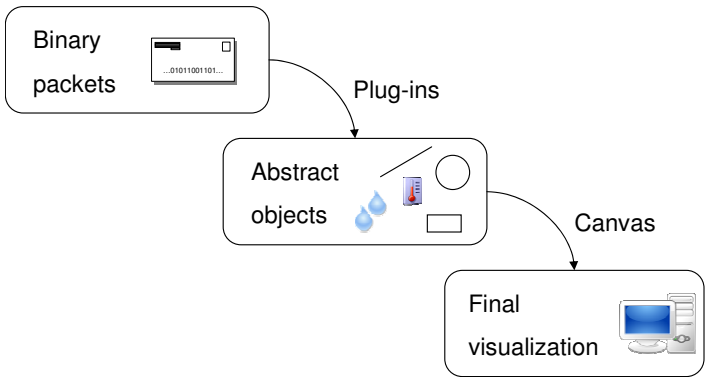


Figure 5.6.: Conversion of data packets until the final visualization

Because the actual mapping of drawing primitives to the final visualization is a property of the canvas, it is easy to add new canvas types, and hence SpyGlass is able to draw on a variety of different canvas types. Other canvas types may be implemented by providing a mapping from the abstract drawing instructions to the concrete target canvas. In its current version, SpyGlass has different built-in canvas types: a standard two-dimensional canvas that is built on top of the Java-2D API [162], a three-dimensional canvas that uses the Irrlicht [57] 3D-engine as its backend and an image writing canvas that stores individual PNG or JPEG images. Like this, it is possible to create new canvasses like a Postscript-canvas to document the sensor network's state in a Postscript document or a canvas that creates a MPEG video for demonstration purposes. Note that it is possible to operate multiple canvases in parallel, so that videos could be created while watching the visualization on the regular Java2D-Canvas on the screen.

SpyGlass cannot only be used for visualizing wireless sensor networks that are currently in operation. It is also able to record all activities starting from an arbitrary point in time. This recorded data can be used for later playback for visualizing it again, watching it at a different speed or changing the type of visualization. This allows not only a playback of the current visualization at a later point in time, but also to get an entirely different visualization by activating a different set of plug-ins for interpreting and visualizing the recorded information. This even allows developing special visualization plug-ins to show interesting details that were not recognizable during the previous visualization run. Apart from different playback speeds, SpyGlass implements features commonly known from video players like fast forward and jumping to certain points in playback.

5.2.4. Built-in Plug-Ins

Currently, SpyGlass already contains a number of plug-ins that range through all seven plug-in categories (Background Painter, Node Relation Painter, Node

Painter, Global Information, Node Positioner, Feedback and Director). These already allow for a wide range of visualization applications and serve as a starting point for new developments and extensions. The following provides a short overview on the built-in plug-ins that ship with the standard distribution of SpyGlass.

The *temperature map plug-in* belongs to the category of the background painters. It allows the visualization component to indicate the spatial temperature distribution by coloring the background between the sensor nodes. Each sensor periodically measures the current temperature; on change, it broadcasts a packet containing its address, the current time and the corresponding temperature. The plug-in residing in the visualization component refers to the currently used node positioning plug-in to assign the temperature value to a position and maps the temperature to a color. Colors between the node positions are interpolated from the values belonging to the surrounding nodes

Currently, a simple *Node Painter* displays a box for each node with the node's address written in it. A *temperature plug-in* indicates the temperature that a node has measured as a numerical value by using the same incoming messages as the temperature map plug-in. The *battery plug-in* uses a bar to indicate how much energy the nodes have left. To do so, the nodes periodically measure their battery voltage and forward packets augmented with their address and a timestamp to the gateway. The *topology plug-in* is a node relation painter. The sensor nodes sporadically send out beacon packets that neighboring nodes use to maintain a list of immediate neighbors. This list is broadcasted periodically and forwarded to the gateway node. In the visualization component, the plug-in processes the packet and draws lines from the sender to all its neighbors. Again, the Node Positioner plug-in is consulted for the nodes' positions.

SpyGlass supplies two different Node Positioner implementations. The first one assumes that the sensor nodes are aware of their positions. Hence, they regularly transmit their coordinates together with their address and a timestamp. The plug-in subscribes to these packets and maps the coordinates to positions on the graphical display canvas. The second Node Positioner plug-in is the *spring embedder plug-in*. In contrast to the aforementioned one, it assumes that the sensor nodes do not have any information about where they are. For this reason, it must calculate positions for each node by other means. To do so, it subscribes to the neighborhood messages and keeps track of the network's topology. Using a spring embedder [144], it then positions nodes that can hear each other close on the canvas, whereas it places nodes without connectivity far away from each other.

The *average neighborhood size plug-in* is a Global Information plug-in. It subscribes to the neighborhood messages used by the topology plug-in, but keeps track only of the number of neighbors. This allows it to display the average network-wide connectivity in the global information sidebar. There is another informational plug-in that counts the overall number of nodes in the network. Subscribing to the neighborhood list packets (or any other periodical packet

type), it maintains a list of the nodes in the network. When a node does not send a packet for a certain time, it is considered not to be part of the network anymore and hence it is removed from the list.

5.3. Implementing Custom Visualization Functionality

Adding new elements to the visualization is straightforward and two cases can be distinguished: one where all necessary information is contained in already available messages and one where additional information is required from the sensor network.

In the first case, the new visualization element uses messages that are already emitted by the sensor network. This only requires the implementation of an additional plug-in. A user must first choose which type of plug-in is needed to fulfill the desired visualization demand. To implement for instance a General Information plug-in that counts the number of network partitions, the messages containing the neighborhood lists could be used. All that is needed is a plug-in that tracks the connectivity information and constructs a graph representation from this data. From this graph representation, the number of network partitions is easily derived and can then be displayed in the sidebar.

In the second case, additional information must be transmitted from the WSN to the visualization component so that a plug-in can visualize this data. Compared to the first case, two elements need modifications: the sensor network application must be extended to transmit the new data packets and a new plug-in must be developed. For example, to implement the visualization of motion detections, a developer needs to define a new data type, construct it as a binary array in the sensor and use the common forwarding service to a gateway node. To visualize the data, a Node Painter plug-in must be implemented which registers itself as a handler for the new data type, parses it and attaches this information to the corresponding node.

5.4. Summary

In this chapter, the SpyGlass visualization environment for wireless sensor networks has been presented. Its generic architecture enables developers of WSN applications to visualize the output of real world or simulated sensor networks in a flexible and straightforward manner. By implementing plug-ins that convert the binary data from the WSN into abstract drawing instructions, SpyGlass decouples the plug-ins from the actual canvas implementation. This allows for an replacement of the output device and format, e.g., a computer screen, a Postscript file, a series of JPEG images or a movie file. Furthermore, SpyGlass distinguishes different plug-in types that perform tasks such as drawing on a specific layer on the canvas, providing node positions or zooming/panning the camera.

With regard to the key properties for a generic WSN visualization environment presented in Section 5.1, SpyGlass complies with these criteria. Its customizable plug-in architecture allows for an realization of audience specific visualizations (e.g., supplying developers with technical details and end-users only with domain specific information). In addition, it is not bound to a specific WSN hardware platform or simulation tool. By contrast, it supports arbitrary input sources and integrating a new platform only requires the ability to send binary data packets to a TCP/IP connection or to store them into a file. Inside the sensor network application, SpyGlass blends well with existing WSN applications because it typically only requires functionality on the nodes that is already present such as a data forwarding facility to a gateway. Finally, it is extensible by means of user-defined plug-ins and custom canvas implementations.

6. Fabric: Data type-Centric Middleware Synthesis

The massively distributed nature of WSNs and their inherent heterogeneity challenges application developers with a number of problems. To solve the application's task it is generally required to establish communication between the different devices involved. In traditional distributed systems, this has led to a widespread use of middleware systems that shield developers from this complexity. These typically base on the client-server paradigm and rely on a stable communication infrastructure. With the advent of sensor networks, these assumptions were no longer appropriate. Strict resource constraints, unreliable communication links and novel communication and programming paradigms render the application of traditional approaches in these networks virtually impossible.

Therefore, a variety of novel programming abstractions and middleware systems were proposed and implemented for the specific requirements of WSNs. In the literature, the term middleware is used for a broad variety of techniques ranging from very simple approaches to far-reaching specifications offering remote method invocations such as CORBA [118]. In the following, we use the definition by Coulouris et al. [32]:

Definition 4 (Middleware) *The term middleware applies to a software layer that provides a programming abstraction as well as masking the heterogeneity of the underlying networks, hardware, operating systems and programming languages.*

In traditional distributed systems, middleware systems tend to be very general in order to support a broad range of applications. This results in rather heavyweight systems that do not match the strict resource constraints of sensor nodes [13, 115]. Since supporting all kinds of applications using a single, static middleware system is not feasible in WSNs, it is essential to adapt the footprint and the offered functionality of a middleware to the requirements of individual applications.

In addition, traditional middleware systems are designed to blend well with the OSI or TCP/IP reference model and typically reside on top of the transport layer. However, novel programming and networking paradigms (cp. Section 2.1) limit the applicability of traditional middleware systems in WSNs. As a result, middleware for WSNs is typically a thin layer of abstraction between the MAC-layer and the application. Against this background, we strongly believe that

the use of custom-tailored middleware solutions for each application can offer the required features while consuming a minimum of resources.

In this chapter, we propose precisely such a middleware synthesis tool called *Fabric* [125, 126, 128]. *Fabric* alleviates the implementation of heterogeneous WSN applications by generating application-specific middleware instances for different target platforms. Application developers supply a high-level data type description in XML Schema augmented with annotations that parameterize the synthesis process. So-called framework developers contribute domain specific expertise by implementing modules that back these annotations with code generating functionality. This separates application and module development and thus leads to a clear distinction between middleware code and application functionality, which is often intermingled in traditional, handcrafted WSN applications.

In addition, we present *microFibre* [130], a novel data type serialization scheme that we have integrated as a module for *Fabric*. On its own merits, *microFibre* provides a methodology for the bit-length optimized serialization of in-memory data structures into payload and vice versa. Its integration into *Fabric* offers code generation functionality for heterogeneous target hardware platforms, which yields a common, bit-length optimized encoding for various target platforms and programming languages.

The remainder of this chapter is structured as follows. Section 6.1 derives challenges that developers face in realizing a typical WSN and Section 6.2 then presents our approach to tackle these challenges. Following, Section 6.3 provides a survey on related work and Section 6.4 explains *Fabric*'s architecture. Subsequently, Section 6.5 describes *microFibre* in detail and Section 6.6 introduces a traditional data type serialization scheme called *macroFibre* that has also been implemented as a module for *Fabric*. Finally, Section 6.7 and Section 6.8 evaluate *microFibre*, *macroFibre* and *Fabric* and present measurements that quantify how WSN applications can benefit from using *Fabric*.

6.1. Motivation

Consider the development of a typical sensor network application where sensor nodes monitor some area for motion events and ambient temperature. As shown in Figure 6.1, the deployed application comprises sensor nodes, gateways and a number of backend systems. Sensor nodes read values from their sensors and forward them towards the gateways. Intermediate nodes perform an aggregation of data received from multiple nodes before they forward received information, e.g., for reducing the amount of network messages or to increase the reliability of data. Gateways provide services that normal sensor nodes cannot perform due to their restricted resources. Examples are the integration with traditional networks or the detection of redundantly received data. Backend systems finally visualize the application data, store it in a database or convert it to another format for the use by third-party systems.

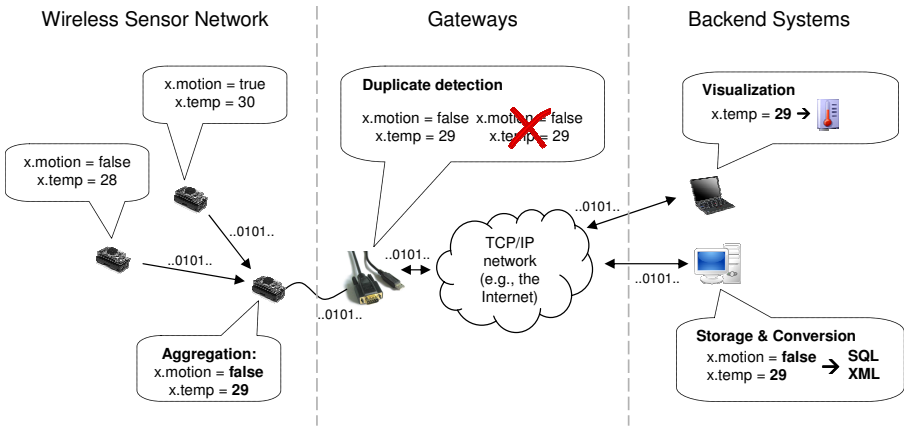


Figure 6.1.: Exemplary WSN application

To implement the applications required for this scenario, an application developer designs the data types storing the application data. In this case, this comprises sensor readings and additional network management information, e.g., to establish routing trees or to synchronize time. At run-time, data is stored in instances of these data types, called *in-memory data structures* or *data structures* in the following (indicated by the speech bubbles in Figure 6.1). To transmit data contained in in-memory data structures to other devices, they are converted to a binary representation serving as the *payload* of wireless network messages (indicated by "..0101.." in the figure).

Already this simple scenario illustrates challenges that developers face when designing such a system. One issue is the implementation of the serialization from in-memory data structures into payload and vice versa. This is required to provide a processing of the application data in conventional programming language constructs (e.g., *structs* in the C programming language). However, data structures are highly application-specific. Hence, the data types and their mapping must be implemented from scratch for every application. Moreover, the devices involved in this scenario are of highly different natures ranging from resource-constrained sensor nodes to resource-rich backend systems. Typically, programming languages, implementation concepts and available features vary between these systems. As a result, the implementation of data types and their mapping from and to payload is not only application-specific but also specific to each class of devices.

This is especially unfavorable because changes are an inherent companion of application development where data structures and the application's logic are constantly subject to change. When manually writing code for the different devices, development teams must keep the implementations consistent manually. Besides the data types and the implementation of (de-)serialization code, applications deal with different types of application data that has different semantics. Depending on the type of data, received network messages must be treated differently by the application. For instance, while the sensor readings

are forwarded towards a gateway, messages for time synchronization might not be forwarded at all.

To conclude, WSN developers deal with a number of repeating tasks for every application. The following list summarizes the above-mentioned aspects that are relevant in the context of this work:

- Serialization from in-memory data structures into payload and vice versa
- Device heterogeneity (programming language, resources, etc.)
- Type-dependent treatment of in-memory data structures
- Consistent integration of changes into the different implementations

6.2. Methodology

Hiding these aspects from the developer by generating application- and device-specific middleware instances for heterogeneous systems is the central design goal of *Fabric*. Application developers exclusively use in-memory data structures and invoke the generated API of the middleware that contains functionalities to transmit and receive instances of these data structures. As depicted in Figure 6.2, the generated middleware implementation converts them to and from the payload of network messages. When performing this conversion, the generated middleware can additionally provide services such as security, reliable transmission or compression.

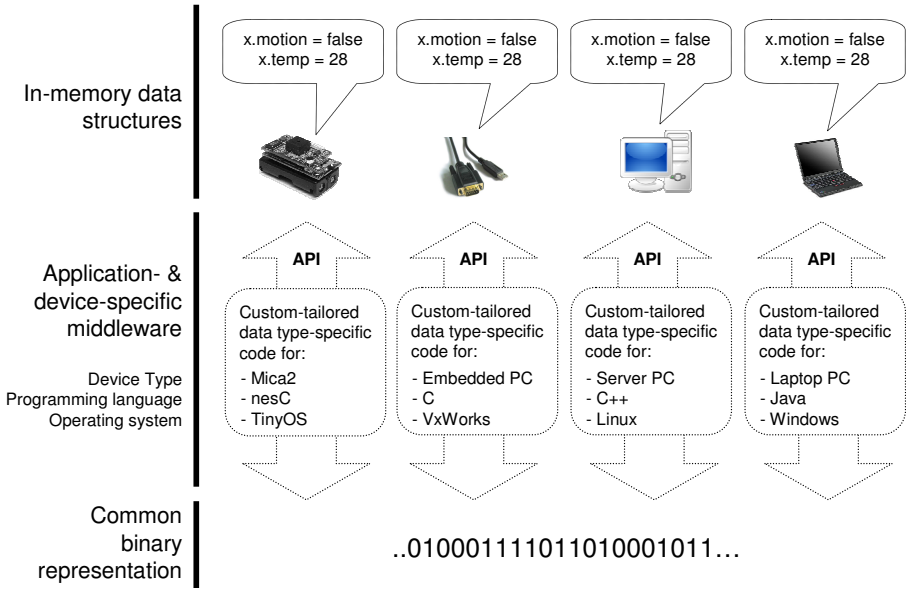


Figure 6.2.: A custom-tailored, application-specific middleware family automates the mapping from in-memory data structures to binary payload and vice versa

Fabric generates the middleware instances based on an abstract data type definition augmented with annotations. Using these annotations the application developer specifies how each data type is treated by the generated middleware. The underlying idea is that data of certain semantics is of a particular, usually complex data type. As a result, each data type has a certain meaning for the application and it must therefore be treated accordingly. To assign this meaning to the individual data types, an application developer annotates data type definitions with treatment *aspects* such as optimized serialization, reliable transport or confidentiality. Hereby, each aspect belongs to a so-called *domain* that embraces related aspects. This concept is referred to as *type annotation* in the following.

The generic Fabric-*generator* passes the annotated data types to *modules* that provide the actual functionality. These generate source code for the annotations they support. The selection of the modules that participate in the code generation process is based on the annotations and a *target specification*. This defines properties of the target hardware platform such as programming language, hardware architecture or peculiarities of the devices such as the lack of dynamic memory allocation.

Invoking *Fabric* with the same annotated data type definitions but different target specifications yields middleware instances for different target devices that share a common knowledge of the individual data types and their treatment. This approach has a number of advantages for the development of WSN applications that are introduced in the following:

Mitigate Effects of Heterogeneity The task of data structure (de-)serialization from/to the payload of network messages is managed by the generated middleware. Consequently, issues arising from the different device architectures, programming languages and alike are hidden from the developer.

Reduced Design Complexity The complexity of realizing heterogeneous WSN applications is reduced since no network-specific code is implemented manually and developers maintain a single definition of the application's data types and their annotations. This is especially desirable when performing changes to the application's data types. Using traditional approaches, development teams keep the different implementations consistent manually, e.g., by sending e-mails describing the changes.

Integrated Networking Optimizations The synthesized middleware can perform optimizations and provide services automatically that represent an optional step in manual implementations. For instance, the generated middleware may provide encryption, compression or reliable transmission. This optional step is frequently omitted and data structures are directly copied from memory into the payload of network messages [140, 141].

Data type-Specific Treatment By annotating individual data types with aspects, the application developer specifies the treatment of individual data types by the generated middleware. Hereby, *data type-specific treatment*

means how an in-memory data structure is converted to payload and how payload of a specific type is handled by the generated middleware. For instance, an in-memory data structure may be stored compressed in the payload or data could be cached by the generated middleware for later retransmission when no acknowledgement was received. Consequently, changes to the annotations automatically manifest themselves in newly generated middleware instances without requiring manual changes to the application's code.

With the above-mentioned functionality and properties of *Fabric* in mind, the following section presents related work and shows how *Fabric* improves the current state of the art in WSN application development.

6.3. Related Work

With the advent of WSNs, a large amount of research concentrated on the development of applications and protocols targeted at specific hardware platforms or application domains [62,109,197]. However, it has been realized that a more general approach is required and that middleware is a promising aspirant to tackle the challenges of WSN application development [61, 143, 186]. This section provides an overview of two major types of middleware systems for WSNs: static middleware solutions and middleware synthesis systems.

Static Middleware Systems The first middleware systems implemented for WSNs provide an abstraction from the underlying hardware platform and are typically custom-tailored to provide an application-specific service. The available solutions can be classified into the following categories:

- service-centric
- database-like
- tuple spaces and publish-subscribe

A number of *service-centric* middleware systems such as Impala or MiLAN have been developed that focus on hardware and programming abstractions. Impala [104], which was created as a part of the ZebraNet project [198], provides an event-based, lightweight layer on top of operating system functionality. Developers implement a set of event and data handlers that react to events generated by the Impala middleware (e.g., timers, new sensor readings or incoming wireless messages) to realize their application functionality. In addition, Impala offers services to the application such as dynamic code update and support for adaptation to changing conditions in the network's state. MiLAN [61] supports sensing applications that specify their Quality of Service (QoS) requirements for different states of the application. MiLAN then monitors the achieved QoS at run-time and tunes the middleware parameters in order to maximize the network's lifetime (e.g., by deactivating unnecessary sensors or hardware com-

ponents). To monitor theses requirements, MiLAN uses existing service discovery and networking protocols that are integrated as plug-ins. The drawback of service-centric middleware systems is that they are typically limited to the WSN. They address neither device heterogeneity, nor the integration with traditional networks. Furthermore, they target a specific application domain such as sensing or data fusion applications, which limits their general applicability.

To overcome the limitations of these service-centric middleware systems, a shift to *data-centric* approaches has been proposed [2, 60, 136, 154]. Systems like TinyDB [107], Cougar [54] and SINA [82, 150] treat the sensor network as a distributed database. Data is requested from outside the WSN by formulating abstract, SQL-like queries. Using special gateway software, these abstract queries are converted into network messages that are then distributed in the WSN. For instance, TinyDB uses controlled flooding to distribute the query and the results in the network. While this database-inspired approach allows for an easy extraction of sensor data from a WSN, it is also limited to this application.

Striving for a more general solution, the *publish-subscribe* paradigm has been suggested to support data-centric application development [38, 196]. Systems such as the well-known tuple-space system Linda were adapted to WSNs [31, 131]. The basic idea of Linda is that devices publish information on a virtual black board or information space. Interested devices subscribe to types of information. The management of subscription, the actual data delivery, etc. is then done by the middleware. To the developer the board appears local and non-distributed. Linda hides the distribution of data from the developer, which is beneficial to alleviate intricate networking aspects. However, Linda also limits the developer's ability to integrate application specific optimizations.

MIRES [154] presents a publish-subscribe API to application developers that is similar to tuple-spaces. Applications publish types of provided data ("topics") which are routed to a sink. A user connected to a sink then subscribes to a number of topics and the MIRES middleware only transmits data belonging to subscribed topics and takes care of the routing towards the sink. It was designed to be deployable on resource-constrained devices and features an implementation on top of TinyOS. MIRES is therefore limited to a single hardware platform and application scenario.

GREEN (Generic & Re-configurable EvEnt Notification service, [153]) explicitly tackles these deficiencies by offering a configurable and run-time adaptable component-based framework. An instance of GREEN is created by connecting the interfaces of a number of predefined components that provide either publish-subscribe services to the application or encapsulate networking aspects. Depending on the target platform and application scenario, a specific instance of GREEN is created by assembling the corresponding components. GREEN is therefore applicable on a broad range of network- and device-types ranging from PDAs to high-power computers. Despite this range of supported device types, GREEN has not yet been ported to resource-constrained WSN devices.

Middleware Synthesis There is a growing interest in techniques synthesizing middleware systems, virtual machines as well as complete applications. In contrast to the aforementioned approaches, these are able to customize the generated code to the targeted hardware environments and application-specific requirements. Graphical or textual specifications enable application developers to construct middleware instances without requiring knowledge of low-level networking aspects [5, 6, 110].

For instance, Lysecky and Vahid propose *eBlocks* [105], a system that generates an implementation from a graphical model, which can be created even by non-technicians. Although easily usable, this tool is limited to a certain class of applications. Bencomo et al. [9] propose the use of model driven software design techniques to generate configurable "middleware families" based on the GREEN framework mentioned above. Developers supply configuration input that comprises application domain, target environment, QoS requirements and UML models. Based on this input, a middleware instance is generated. However, since their approach is based on GREEN, the same limitations apply. ATaG [7] models an application as a set of abstract tasks (information processing entities backed with user-supplied code) and abstract data items from which an ATaG instance is synthesized. Tasks can be instantiated at compile-time or run-time in order to match the actual hardware and software environment. However, the proposed framework remains an abstract proposal that has not yet been implemented on actual WSN hardware.

The *application specific virtual machine* (ASVM) approach [98] generates specialized virtual machines (VMs) for TinyOS operated sensor nodes. Users specify requirements of their application by connecting application handlers to ASVM templates. By exploiting this application-specific knowledge, the generated ASVM instance is a lean VM with an optimized instruction set. This allows for bandwidth-saving dynamic reprogramming of sensor nodes since only the byte code executed by the VM is replaced. However, this approach is limited to the TinyOS operating system and it does not address heterogeneity or networking issues.

The SWARMS project [21,89] uses a concept called *dVSIS* (Distributed Virtual Shared Information Space) offering a programming abstraction. The underlying idea is that the result calculated by the WSN is contained in a virtual document. An application uses local rules to contribute to this virtual document using so-called IN, OUT and LOC filters. At design time, an XML Schema document models the structure and content of the virtual document. A component called STAX (Simple Typed API for XML) generates a middleware instance and an API for different heterogeneous device types. Application developers implement their filters on top of this API, which invokes callbacks for each opening, or closing tag, attribute or character data in the received data. This approach allows programming the devices using a custom-tailored, application-specific middleware that hides networking aspects from the developer. Applications written for the generated middleware resemble the parsing of XML documents and therefore developers typically require state machines

to track the current position in the received data. While this is an interesting programming abstraction, developers are restricted to this programming model.

Conclusion Compared to static middleware solutions, the use of middleware synthesis systems is a promising means to reduce the complexity of implementations and to mitigate challenges of WSN application development such as resource constraints, heterogeneity and optimization of networking issues. A key advantage of these approaches is that lean, custom-tailored code can be generated that contains only the required functionalities thus avoiding unused code in the program memory. By exploiting application knowledge, they can avoid the overhead that would be imposed by traditional, static middleware systems. Furthermore, developers are able to create sensor network applications without the need for special low-level device expertise.

However, to the best of our knowledge, none of the existing frameworks provides the unique features of our proposed middleware synthesis tool *Fabric*. They focus either on a special class of applications (eBlocks, ASVM), specific hardware (TinyDB, Impala), novel programming methodologies (SWARMS, ATaG) or have not been implemented on resource-constraint WSN devices (ATaG and the approach proposed by Bencomo et al.). Furthermore, none of the abovementioned frameworks explicitly addresses networking issues (such as optimized data type serialization and security) or can adapt the synthesized code on a data type level. As stated above, we strongly believe that this kind of support is a key requirement for future WSN middleware frameworks.

6.4. Architecture

This section presents the architecture of *Fabric*, demonstrates how custom-tailored middleware instances are generated and how this functionality is actually implemented using so-called *modules*. Hereby, *Fabric* distinguishes between two roles: an application developer and a framework developer. An application developer's primary interest is an easy to use, flexible system. The framework developer customizes the generic *Fabric* system and implements the functionality available to the application developer. These two roles are clearly separated in *Fabric*'s architecture that is shown in Figure 6.3. It comprises

- input from the application developer (annotated data types, target specification),
- a generic component called *Fabric*-generator and
- input from the framework developer (framework specification, modules).

In the following, these three building blocks are described in detail: Section 6.4.1 presents the application developer's view of *Fabric*. Then, Section 6.4.2 explains the internal mode of operation of *Fabric* and Section 6.4.3 outlines the framework developer's role.

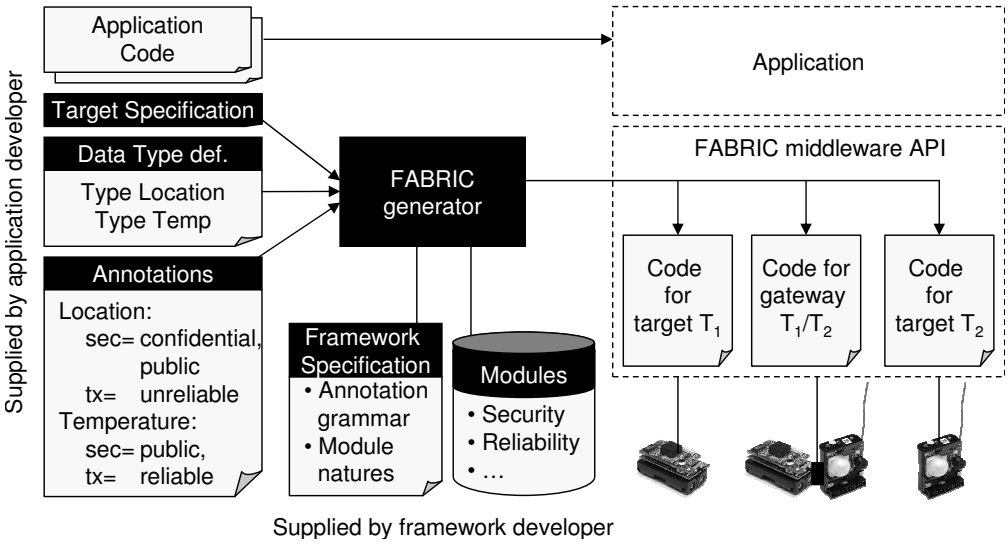


Figure 6.3.: The *Fabric* architecture

6.4.1. Application Developer's View

As shown on the left hand side of Figure 6.3, the application developer provides the

- annotated data types and a
- target specification.

The annotated data types define the data structures of the application data as well as their desired treatment by the generated middleware. The target specification defines properties of the concrete hardware platform such as the programming language and platform-specific peculiarities by using a list of so-called *natures*.

Using this input, *Fabric* synthesizes the middleware instance that offers a number of API functions to the application developer. The application developer then uses the API of the generated middleware to implement his application functionality. The generated middleware is then compiled together with the user's application and linked against an operating system (e.g., TinyOS or a device firmware, cp. Section 2.1.5), a simulation tool (such as Shawn, cp. Chapter 4) or a visualization tool (such as SpyGlass, cp. Chapter 5). In the following, the input sources prepared by the application developer are described in detail. Then, we show how this input is used to synthesize a middleware instance.

Imagine an application that helps optimizing the heating of a building similar to the one described in Section 6.1. Here, the sensor nodes measure the ambient temperature and augment this sensor reading with a position. The first step is to describe the application's data types. Figure 6.4 depicts the data types `Temp` and `Location` that are required for this application. `Temp` only holds a single

floating-point value while `Location` comprises a sequence of two floating-point values named `x` and `y`. Let us further imagine that the application developer likes to select security and reliability annotations for both data types, e.g., encrypting the *Temp* data type and transmitting it reliably.

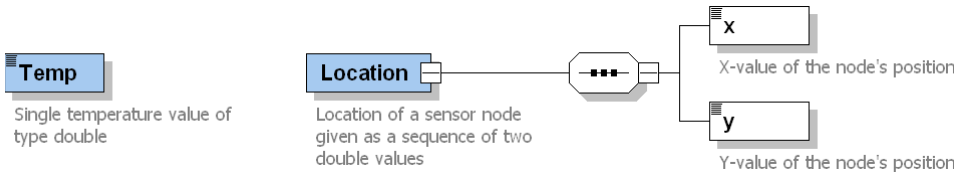


Figure 6.4.: Application data types `Temp` and `Location`

Annotated Data types

This type annotation concept requires a language for the data type definition and the ability to annotate each data type. *Fabric* coherently uses XML and XML Schema technologies (cp. Section 2.2) for the input supplied by the application developer. The data type definitions are expressed as XML Schema documents while the annotations are represented as XML documents.

`Element` declarations of an XML Schema document represent the data types that are available in the generated middleware, which is consistent with the intended use of `element` declarations. These define the elements that may occur as the root element of valid XML instance documents. Accordingly, each `element` declaration manifests itself as a data type in the generated middleware. *Fabric* supports the entire XML Schema standard and hence the data structures may be arbitrarily complex. Figure 6.5 shows an XML Schema document that represents the data types from Figure 6.4. Two element declarations define the structure of both data types.

Both `element` declarations incorporate an XML document containing the annotations for each data type. This is possible because XML Schema supports a machine-readable annotation of nearly all XML Schema tags with user-defined XML documents. As discussed in Section 2.2.2, these machine-readable annotations are contained as children of the `appinfo` tag, which is in turn a child of the `annotation` tag. The tags related to XML Schema and *Fabric* are separated using different namespaces (indicated by the prefixes `xs:` and `fabric:` in this example). The root element of the annotation document (called `fabric`) is contained as a child of the `appinfo` tag. The actual structure of the contained XML document is defined by the framework specification as discussed later on.

In this example, the annotated aspects are `confidential` and `public` from the domain `security` and `(un)reliable` from `tx`. This annotation example is representative for two distinct cases:

1. `Temp` has one annotation per domain (security: public, tx: reliable)

Figure 6.5.: XML-Schema annotation

2. `Location` has more than one annotation per domain (security: confidential and public).

The first case specifies unambiguously how this data type must be treated by the generated middleware and no run-time decisions are necessary. Yet, in the second case, run-time decisions on data treatment exist and the generated middleware contains code for both aspects and options for selecting the active one. The user can therefore choose at run-time whether to activate confidential or unsecured transmission depending on the context.

Target Specification

The annotated data types describe the application's data types and their desired treatment by the generated middleware in a platform- and programming language-independent manner. To convert this abstract specification into a

middleware for a specific device, programming language, etc., the application developer supplies a *target specification* (cp. Figure 6.3).

It expresses properties of the target platform such that *Fabric* can select an optimal set of code generating modules for the code generation process. The target specification is comprised of so-called *natures*. Conceptually, natures are a classification along several orthogonal axes such as programming language, target platform and device type. Details on this process follow in Section 6.4.3.

Middleware Synthesis

To support the process of middleware synthesis, *Fabric* is available as a plug-in for the well-known Eclipse [172] development environment. Instead of dealing with command line parameterizations, this plug-in facilitates the use of *Fabric* by providing the integration into a widely used IDE.

Fabric's Eclipse plug-in supports the application developer in all steps necessary for generating middleware instances for different heterogeneous devices. In addition, the application developer is assisted in preparing all required inputs directly in the graphical IDE including the target specification and the annotated data types. This is achieved by providing features specific to *Fabric* and by using existing development tools for Eclipse.

To compose the target specification, the plug-in offers a configuration wizard. Figure 6.6 depicts a page from the wizard's graphical user interface. It shows the available natures that are given by the framework specification, which allows for a graphical composition of the target specification. This wizard can be completed multiple times to create a number of target specifications for different target devices, platforms and programming languages (e.g., sensor nodes, gateways and backend computers).

Using the wizard, the user also specifies how the generated middleware source code will be integrated into the source tree of his application. Figure 6.7 shows how this integration is configured. It is either possible to choose a target path of an existing Eclipse project or to select an arbitrary other folder in the case that Eclipse is not the primary IDE for a certain project. As a result, the middleware is an integral part of the application and not some third-party library that requires additional linking steps.

To edit the data types and their attached annotations, Eclipse offers graphical editors for XML Schema and XML documents. Since XML Schema documents are valid XML documents, the annotated data types can be edited using any of both editors. The XML Schema editor is used to define the application's data types while the XML editor is used to edit the annotations. Figure 6.8 shows the overview screen of Eclipse's XML Schema editor. On the left, the `element` declarations of `Location` and `Temp` are shown. Using this editor, the application developer can use a graphical editor to design the application's data structures.

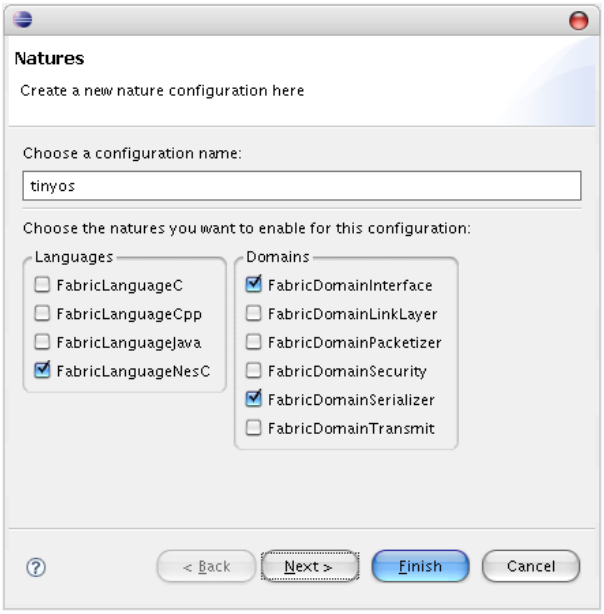


Figure 6.6.: Composing the target description using *Fabric*'s Eclipse plug-in

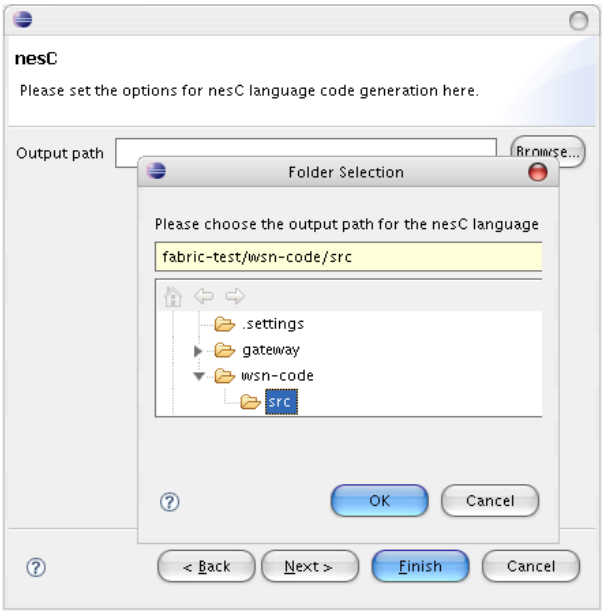


Figure 6.7.: Configuring project and language specific settings using *Fabric*'s Eclipse plug-in

Figure 6.9 shows a detail of the same XML Schema document, this time opened with Eclipse's graphical XML editor. The figure shows the element declaration of the `Temp` data type. Below the element declaration, the annotation of this data type can be seen. Using this graphical XML editor, the application devel-

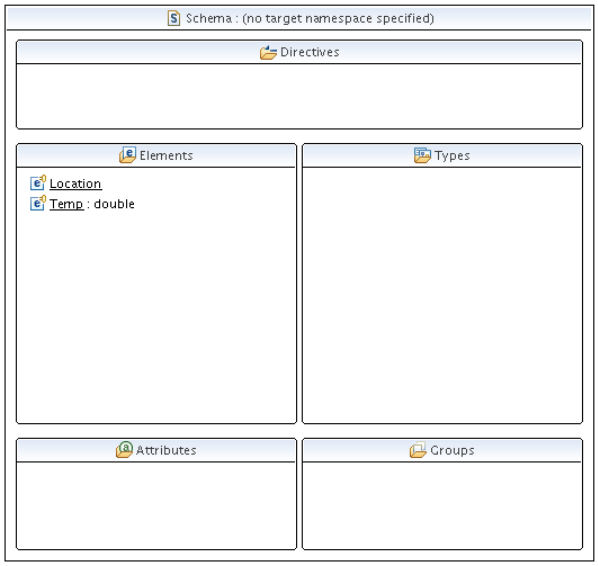


Figure 6.8.: Graphical editing of the XML Schema file in Eclipse

oper can attach and modify the annotations without dealing with syntactical issues of XML.

xs:element	(annotation?, (simpleType | complexType)?, ((unique | key | keyref))*)
name	Temp
type	xs:double
xs:annotation	(appinfo | documentation)*
xs:documentation	Single temperature value of type double
xs:appinfo	(namespace:uri="##any")*
fabric:fabric	
fabric:Domain	
name	security
fabric:Aspect	public
fabric:Domain	
name	tx
fabric:Aspect	reliable

Figure 6.9.: Using Eclipse's XML editor for attaching the data type annotations

From the user's perspective, using *Fabric* boils down to editing the annotated data type definitions in the graphical user interface of Eclipse and saving the changes to disk. Then, *Fabric*'s Eclipse plug-in uses the target specifications generated by the graphical wizard as well as the annotated data type definitions and invokes the *Fabric*-generator. Following, it updates Eclipse's resource view to reflect the newly created or changed files in the GUI (such as C, C++ or nesC source and header files or Java classes). This whole process is transparent to the user and thus promotes the use of source code generation through a seamless integration into an IDE.

6.4.2. Generation Process

The code generation now transforms the annotated data types into middleware source code such that the resulting code complies with the target specification. When synthesizing a middleware instance, annotated aspects from multiple domains influence the handling of a data type in the generated code. Hereby, each aspect is handled by a particular module. This means that for each annotated aspect one module is invoked that generates source code for this aspect.

As a result, the code generated by the modules participating in the synthesis process must be compatible since the output of code generated by modules from one domain is the input for the code generated by modules from the next domain. Consequently, the *Fabric*-generator must select a set of compatible modules for the annotations of each data type and the target specification.

For each annotated aspect of each data type, it passes the aspect, the data type and the target specification to each module. Modules reply with a description containing whether and how they are capable of performing the requested operation. This *handler description* comprises a

- set of *pins* to the previous and next domains and a
- priority (one of `Unable, Default, Low, Normal, High`)

The *pins* specify which data types are accepted as input and which are produced as the output of the generated code. The *priority* determines how well the code generated by a module fits a given aspect and the target specification (e.g., a module that generates standard C code may return a lower priority than a module generating custom-tailored C code for a particular device).

Based on the obtained handler descriptions, the *Fabric*-generator gradually deselects modules until exactly one module remains for each annotated aspect. The selection process results in a *synthesis-graph* for each data type describing which modules are invoked and which of their pins interact. Deselecting modules is performed by a number of checks. First, all modules are removed that do not comply with the target specification (indicated by the priority `Unable`). Then, all consecutive de-selection decisions are based on two pin-interoperability checks (*intra-domain check* and *inter-domain check*) and module priorities. Algorithm 2 gives the formal description of the selection process.

In the following, this process is illustrated at the example of the data type `Location`. Let us assume that Figure 6.10 depicts the available modules. Let us further assume that the framework specification contains four domains: the two from the annotation example as well as two additional domains, `Interface` and `Serialize`.

At the beginning, the handler descriptions are collected from all available modules (lines 2-10 in Algorithm 2). During this process, m9 is deselected because it returned the priority *Unable*. This is because its supported aspect (`reliable`) is not annotated but other aspects from the domain are chosen (line 6).

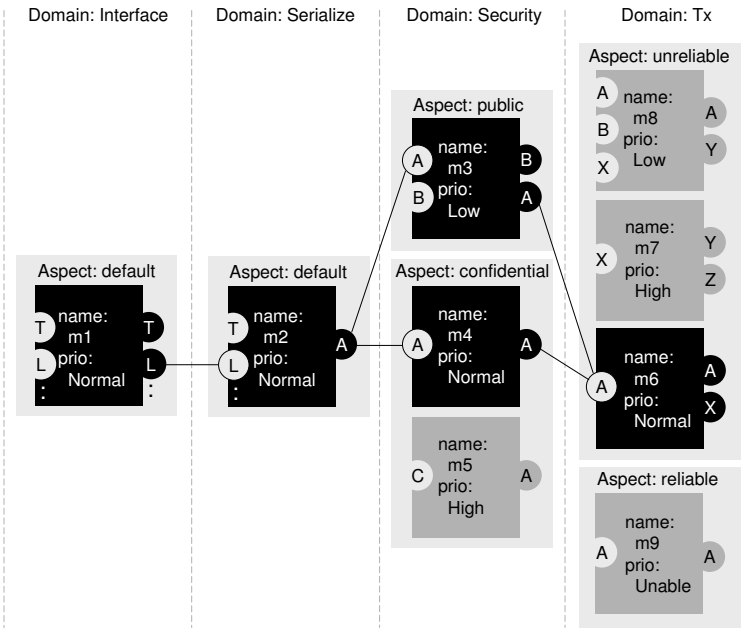


Figure 6.10.: Synthesis graph for the `Location` data type

The first pin-based de-selection step is the *intra-domain check* (lines 11-23): It results in two sets of pins for each domain, one containing pins interfacing to the previous domain (left hand side of a domain), and the other for the next domain (right). Both contain all those pins from modules so that one module for each aspect can be found. Modules that do not have at least one pin in both sets respectively are deselected. This ensures that it is possible to find a module for each aspect of a domain using a single pin. In this example, for the domain `security`, both sets contain only pin A. Since m5 offers only C on the left hand side, it is deselected. For `tx`, the sets contain A, B, X and A, X, Y, Z (note that m9 was deselected earlier).

The second de-selection step is the *inter-domain check* (lines 24-37): Now the next and previous sets of two neighboring domains are intersected. Again, modules that do not have at least one pin in both sets of a domain are deselected. In this example, for the domains `security` and `tx` only pin A remains. As a result, m7 is deselected.

Now only compatible modules remain. If more than one module remains for an aspect, the one with the highest priority is chosen (line 35). Like this, m8 is deselected. If two modules feature the same priority, an arbitrary one is chosen. Now, the selection process has completed and the remaining modules and their chosen pins are indicated in black. Finally, $handler_{d,a}$ holds the handler description for the domain d and aspect a. The synthesis-graphs for each data type contain the entire information on how to synthesize the middleware code. Finally, the *Fabric*-generator executes the modules specified by the synthesis-graphs and merges the individual source code contributions into the resulting

Algorithm 2 Synthesis-graph construction

Require: t {Annotated data-type}
Require: modules {List of available modules}
Require: domains {Ordered list of domains}
Require: natures {List of required natures}
for all $d \in domains$ **do**
  $aspects := Aspects(d, t)$ {All annotated aspects for this data-type and domain}
  for all $a \in aspects$ **do**
    for all $m \in modules$ **do**
      $\langle prio, prevPins, nextPins \rangle := m.canHandle(t, d, a)$
      if prio != Unable **then**
        $handlers_d := handlers_d \cup \{\langle a, m, prio, prevPins, nextPins \rangle\}$
      end if
    end for
  end for
  {Intra-domain pin check}
  for all $a \in aspects$ **do**
    {Create a union of previous/next pins for each aspect of this domain}
    $P_{prev,a} := \bigcup \{prevPins(h) | h \in handlers_d \wedge aspect(h) = a\}$
    $P_{next,a} := \bigcup \{nextPins(h) | h \in handlers_d \wedge aspect(h) = a\}$
  end for
  {Calculate common previous/next pins between all aspects of this domain}
  $domainPins_{prev,d} := \bigcap_{\lambda \in aspects} P_{prev,\lambda}$
  $domainPins_{next,d} := \bigcap_{\lambda \in aspects} P_{next,\lambda}$
  {Remove handler descriptions with no common pins}
  $handlers_d := handlers_d \setminus \{h \in handlers_d | prevPins(h) \cap domainPins_{prev,d} = \varnothing\}$
  $handlers_d := handlers_d \setminus \{h \in handlers_d | nextPins(h) \cap domainPins_{next,d} = \varnothing\}$
end for
{Inter-domain pin check}
for all $d \in domains$ **do**
  if exists(nd := nextDomain(d)) **then**
    {Calculate common previous/next pins between neighboring domains}
    $interDomainPins := domainPins_{next,d} \cap domainPins_{prev,nd}$
    {Remove handler descriptions with no common pins}
    $handlers_d := handlers_d \setminus \{h | nextPins(h) \cap interDomainPins = \varnothing\}$
  end if
  $aspects := aspects(d, t)$ {All aspects for this data-type and domain}
  for all $a \in aspects$ **do**
    {Select the handler description with the highest priority}
    $handler_{d,a} := h \in handlers_d | aspect(h) = a \wedge Priority(h) = MAX$
  end for
end for
return {Inter-domain pin check}

middleware-instance as described in the following section.

6.4.3. Framework Developer's View

While the application developer uses high-level annotations for defining the treatment of individual data types, the framework developer uses his domain expertise to back these annotations with actual functionality. To offer this functionality, the framework developer provides a

- *framework specification* and a
- set of *modules*.

In conjunction with the generic *Fabric*-generator, they form a custom-tailored instance of *Fabric*. This section describes the contents of the framework specification and shows how a framework developer implements modules that provide the actual code generation functionality.

Framework Specification

The framework specification defines the functionality that is provided by a specific instance of *Fabric* and comprises the

- available *natures* (and *domains*) and an
- *annotation grammar* defining the structure of annotations.

Natures are unique string values that represent a classification along orthogonal axes like programming language, target platform and domain. On the one hand, they allow application developers to express the required features of the generated middleware in the target specification. On the other hand, modules use natures to check whether they conform to a target specification. Table 6.1 provides an exemplary list of natures (grouped by related natures) how they could occur in a particular instance of *Fabric*.

Programming language	**Domain**	**Hardware plattform**
Java	Interface	Generic
C++	Serialize	Mica
C	Security	Mica2
nesC	Tx	Telos
...	...	iSense
		...

Table 6.1.: Exemplary list of natures grouped by similarity

The target specification may contain any combination of the available natures with the exception of natures that represent domains. As discussed in Section 6.4.2, the *Fabric*-generator uses the order of the domains for determining neighboring domains. For instance, the domains shown in Table 6.1 resemble the ones from Figure 6.10.

The second ingredient of the framework specification, the annotation grammar, defines structure and contents of valid annotation documents. Since the annotations are expressed as XML documents, the framework specification contains an XML Schema document that defines the rules for valid instance documents. Using a validating XML parser, the *Fabric*-generator checks whether embedded annotations are valid data type annotations.

Module Implementation

Apart from the preparation of the framework description, a framework developer's main task is to implement or to assemble a set of modules. A module's tasks are twofold: on the one hand, the module must interact with the *Fabric*-generator and on the other hand, the actual code generation logic must be implemented. To develop a module, a framework developer creates a Java class that implements *Fabric*'s Module API that is shown in Figure 6.11. The Module API is used by *Fabric* to

- obtain the handler description from each module and to
- trigger the code generation process.

```
1  ...
2
3  HandlerDescription canHandle(TargetSpecification, DataType, Aspect);
4
5  void initialize(Fabric, Schema, TargetSpecification, Workspace);
6
7  void generationBegin(Fabric, HandlerDescription, Workspace);
8
9  void handle(Fabric, HandlerDescription, Workspace);
10
11 void generationEnd(Fabric, HandlerDescription, Workspace);
12
13 void shutdown(Fabric, Schema, Workspace);
14
15 ...
```

Figure 6.11.: Application programming interface (API) of *Fabric* modules

The obtained handler descriptions serve as the input for the computation of the synthesis graphs (cp. Section 6.4.2). As soon as the synthesis graphs are constructed, the set of participating modules is known and the code generation methods are invoked on these modules. The sequence of invocations of the Module-API's methods is executed in six steps until the middleware is generated:

1. For each data type and annotated aspect, `canHandle` is invoked. Based on the supplied target specification, data type and annotated aspect, the method returns a *handler description*. The *Fabric*-generator then computes the synthesis graphs based on the handler descriptions obtained from all modules.
2. Each module that is listed in one of the synthesis graphs is notified once about its participation in the following code generation process by invoking `initialize`. The module is parameterized with the instance of *Fabric*, the target specification and a `Workspace`. The following steps 3-5 are repeated for each data type and they are performed only for modules that are listed in the synthesis graph of this data type.
3. Before the actual code generation, `generationBegin` is invoked.

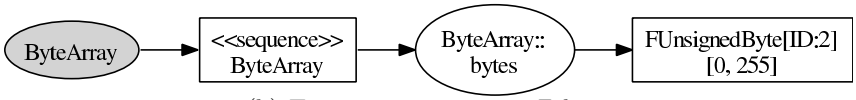
4. A call to `handle` triggers the actual code generation. The handler description returned by the module's `canHandle`-method is passed back to the module now serving as the contract to the previous offer. During the construction of the synthesis graphs, incompatible pins were removed from the handler description and only a single pin to the previous and the next domain remains.
5. After the code generation, `generationEnd` is invoked
6. Finally, a call to `shutdown` terminates the middleware generation.

To perform step 1, a module assembles a handler description as described in Section 6.4.2. The description of pins contained in the handler description is independent from a particular platform or programming-language. Consistent with the annotated type definitions and the annotation grammar, the pins are specified using XML Schema documents.

Imagine a module that generates (de-)serialization code converting a data type to an array of bytes and vice versa. This module would return only one input and one output pin. The input pin is the data type passed as a parameter to `canHandle` (e.g., `Temperature` or `Location`) while the output pin is always a byte array. Figure 6.12(a) depicts an example of how the byte array pin is represented in XML Schema.

```
1  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
2    <xs:element name="ByteArray">
3      <xs:complexType>
4        <xs:sequence>
5          <xs:element name="bytes" type="xs:unsignedByte"
6                      minOccurs="0" maxOccurs="255"/>
7        </xs:sequence>
8      </xs:complexType>
9    </xs:element>
10 </xs:schema>
```

(a) XML Schema representation



(b) Tree representation in *Fabric*

Figure 6.12.: Description of a pin representing an array of bytes

Fabric converts the XML Schema document to a semantically equivalent tree to obtain a coherent representation of the XML Schema document that is independent from its large feature set (such as local or global complex and simple type definitions, inheritance through restriction or extensions, etc.). Figure 6.12(b) shows the tree representation of the XML Schema document from Figure 6.12(a). The elliptical shapes denote elements while the rectangles represent complex or simple types. The root of the tree is the global element

declaration (`ByteArray`), the intermediate complex types define the structure of the data types and the leaves represent simple types and attributes. Using this tree representation, *Fabric* can compare two XML Schema documents for structural equality, which is used to compare pins with each other during the construction of the synthesis graphs (cp. Section 6.4.2).

During steps 3-5, modules actually generate source code. It is at the module's disposal, at which step it actually adds source code contributions to the generated middleware. However, after the final call to *generationEnd*, each module must have completed the generation phase. The resulting middleware source code is then contained in an instance of a *Workspace* class. A specialized Workspace is available for each of the supported programming languages (currently C, nesC, C++ and Java). It allows modules to contribute generated source code and decouples the phase of source code contribution from the actual persistence into files. This is crucial as some programming language constructs require a strict ordering inside the generated files. For instance, the C language requires that variable declarations must occur at the beginning of a method and that functions are declared before they are used.

Consequently, simply writing the contributions of several modules sequentially into files does not yield the desired output. A Workspace allows an arbitrary order of source code contributions and, depending on the context, arranges the contributions in the correct order when writing the files to disk. It also ensures the syntactical correctness of the generated source code. Before committing any change to the workspace, it verifies whether the change would violate the syntactical correctness of the already contained source code. A number of simple checks (e.g., whether a duplicate method or class would be added) are available for all supported languages. In addition, *Fabric* uses the ANTLR [124] framework[1] and its programming language grammars to facilitate a complete syntactical verification. This allows for a quick identification of the module that provided erroneous code. Finally, the workspace is persisted to disk resulting in files on the hard drive and an updated resource view in the Eclipse IDE.

6.5. Bit-length Optimized Data type Serialization (microFibre)

A key feature of WSNs is to solve a problem as a coherent swarm of devices. Since the devices do not share a common memory, communication between the individual devices is the enabling means to evolve a collective behavior. When developing applications for WSNs, energy awareness and harsh resource constraints are constant challenges [64,142]. Given that receiving and transmitting data are the most energy-hungry operations in a WSN [95,152], the overall duration of broadcasting activities should be minimized to conserve the scarce energy budget of the sensor nodes.

[1] ANTLR: ANother Tool for Language Recognition

At the application level, two major options for reducing the amount of energy consumed by the radio interface exist: reducing the number of transmitted messages and minimizing the length of individual transmissions. While reducing the number of messages is highly specific to each protocol, minimizing the length of messages can be applied to every WSN application. To achieve this goal, the representation of application data as payload is optimized to require only a minimum of bits. However, this presents an optional step in the implementation of WSN applications and is frequently omitted [140, 141].

Automating this step by integrating it into *Fabric*'s middleware generation process is the objective of a novel technique called *microFibre* that is proposed here. On its own merits, *microFibre* provides a methodology for the bit-length optimized serialization of in-memory data structures into payload and vice versa. Integrating *microFibre* as a module into *Fabric* is advantageous for two major reasons:

1. Optimized (de-)serialization code is automatically generated for heterogeneous target hardware platforms. This yields a common binary encoding for various target platforms and programming languages.
2. *microFibre* exploits the formal and detailed definition of the data types to optimize the encoding of the payload. Compared to related and frequently used approaches, *microFibre* produces remarkably short binary encodings while the resulting middleware code is lean and custom-tailored to match the resource constraints present in WSNs.

In the following, Section 6.5.1 introduces related techniques, discusses their pros and cons and states how *microFibre* in combination with *Fabric* improves the current state of the art. Section 6.5.2 then presents the architecture of *microFibre* and explains in detail how in-memory data structures are (de-)serialized from/to payload. Section 6.6 introduces a second (de-)serialization module for *Fabric* called *macroFibre*. While *microFibre* provides a bit-length optimized encoding, *macroFibre* represents a traditional approach to data type serialization. Section 6.7 evaluates *microFibre* and *macroFibre* and compares encoding length, footprint and CPU requirements with related work. Section 6.8 concludes this chapter with a summary and an evaluation.

6.5.1. Related Work

To implement the conversion from in-memory data structures to payload, two fundamentally different approaches exist. In the first approach, developers handcraft data structures and manually implement the mapping from and to payload. The second approach is to describe the application's data types in a platform-independent manner and to transform this specification into platform-dependent code for multiple target platforms and languages.

Approaches Based on Handcrafted Code

In the context of WSNs, the application's data types are in the majority of cases implemented as nested data structures directly in the target programming language [30, 141]. An approach frequently used in WSN software is that the payload is an exact, bitwise copy of the in-memory representation on the sensor node. Although widely used due to its simplicity, this approach has various, severe drawbacks: it relies on the assumption that different compilers for different devices represent data structures in exactly the same manner so that data serialized from memory on one device can be de-serialized on another. This assumption often holds inside the sensor network where exactly the same program is running on identical hardware. However, due to the increasing standardization of the radio interface (cp. Section 2.1.3), even inside the WSN, device heterogeneity is a non-negligible issue.

The *network types* [30] technique addresses this problem by proposing a programming language extension to nesC [56] that introduces new keywords to the language. Instead of using the traditional C-language constructs *struct* and *union*, developers use *nx_struct* and *nx_union* and a number of new data types (*nx_int8_t*, *nx_uint8_t*, *nx_int16_t*, etc.). A modified version of the nesC compiler translates these new language features into standard C code before it is actually compiled. This approach specifically enables the cooperation of sensor nodes that are programmed using nesC by mitigating the effects of different memory alignments and endiannesses. Yet, the designed data structures are still confined to the nesC programming language. Hence, this approach is neither suited for heterogeneous WSNs nor for the integration with traditional networks. Custom data types that are created using these new keywords are subject to a number of restrictions. The most important one is that *nx_struct* and *nx_union* may not contain bit-fields. Bit-fields are commonly used to decrease amount of memory required by a data structure. When this in-memory representation is directly copied into the payload of network messages, this also reduces the payload length. As a result, this common optimization cannot be used.

The External Data Representation (XDR, [156]) standard specifies an architecture independent data encoding to ease the data exchange between heterogeneous computers. XDR is typically available as a library that is used by programs to convert data from and to a common encoding. It defines a number of *XDR data types* and their mapping to this common encoding. XDR is typically used by Sun's Remote Procedure Call (RPC, [164]) framework, which is used by the well-known Network File System (NFS, [151]). Despite its usefulness for desktop and server class computers, XDR is not suited for resource-constrained devices and the resulting encoding is not optimized for size.

A textual specification of the payload format is another alternative. Hereby, the meaning of each byte is described in a human-readable form. Common examples for this procedure are Internet protocols, which are defined in so-called *Requests For Comments* (RFCs). Figure 6.13 depicts an excerpt from

RFC 768 [81] that standardizes the *User Datagram Protocol* (UDP). This is a typical example of how protocol headers (that are the payload of other protocols such as IP) are defined. Developers manually implement code that copies the content of in-memory data structures to the correct positions in the payload as defined in the specification. In doing so, they must take care of different memory alignments and endiannesses manually.

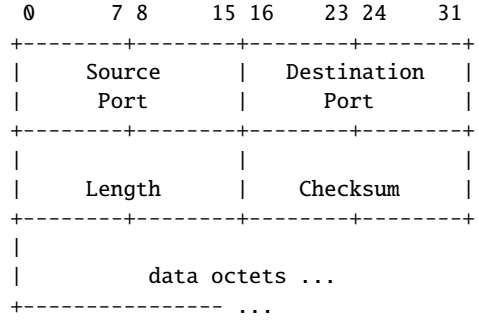


Figure 6.13.: User Datagram Header Format

Code Generation Techniques

The second approach is to describe the application's data types in a platform-independent manner and to transform this specification into platform-dependent code for multiple target platforms and languages automatically. The generated code comprises the language- and platform-specific data types and routines that (de-)serialize them to and from a common encoding. A number of techniques exist that are widely used in various disciplines of computer science.

Walther et al. [179] present an approach that automates the manual implementation of textual protocol header definitions as shown in Figure 6.13. They propose a technique to generate data types and (de-)serialization code automatically from protocol headers that are defined using this ASCII-art like syntax. However, the underlying input format used by this approach only provides a limited expressiveness for the definition of complex, nested data structures compared to languages such as XML Schema or ASN.1.

The Abstract Syntax Notation One (ASN.1, [77]) provides a language for describing data structures in a manner very similar to XML Schema. Figure 6.14 shows a very basic ASN.1 document that is semantically equivalent to the XML Schema document presented earlier in Section 2.2.2 on page 25. Besides the ASN.1 language, encoding and decoding schemes have been standardized, among them the XML Encoding Rules (XER, [80]), the Basic Encoding Rules (BER, [78]) and the Packed Encoding Rules (PER, [79]). The XER encoding produces human- and machine-readable XML documents from in-memory data structures. BER defines a self-contained format that augments the actual data of a document with type and length information such that it can be decoded by a receiver that has no knowledge of the original ASN.1 document. Since

XER and BER documents are quite verbose, PER was designed to yield a very effective and bit-length optimized encoding. Nowadays, ASN.1 has mostly been superseded by XML Schema technologies and as we show in Section 6.8, it is does not inherently target resource-constrained devices.

```
1 Notarget-ns
2 DEFINITIONS AUTOMATIC TAGS ::=
3 BEGIN
4
5 Long ::= INTEGER (-9223372036854775808..9223372036854775807)
6
7 Counter ::= Long
8
9 END
```

Figure 6.14.: Basic ASN.1 data type definition (equivalent representation of the XML Schema document shown in Figure 2.10(a))

XML Schema (cp. Section 2.2.2) as the latest and most generally accepted standard for the definition of data types commonly uses verbose, human-readable XML as payload format, which is unsuitable for the use in WSNs. Even so, approaches enabling XML processing on sensor nodes exist. For instance, Buschmann et al. propose *"<<ASTAX"* [20] that allows for an event-driven programming of WSNs that resembles the Simple API for XML (SAX).

An in-depth overview of compact encodings for XML documents is given by Werner et al. [182, 183]. The authors introduce a novel technique called *Xenia* that yields a very effective encoding of XML instance documents that surpasses previously available XML compressors in terms of encoding efficiency. By constructing a pushdown automaton from an XML Schema document, Xenia encodes an XML instance document as a path through the automaton, which can be encoded very efficiently. The key difference between Xenia and *microFibre* is that *microFibre* does not compress XML documents but in-memory data structures. Xenia optimizes the encoding of the markup of XML instance documents and does not focus on an optimized encoding of character data. *microFibre* performs both optimizations, which is beneficial to reduce the payload length of serialized in-memory data structures.

Conclusion

Using code generation techniques is an appealing means to overcome the inherent heterogeneity of WSNs while optimizing networking issues. However, to the best of our knowledge, none of the existing solutions and techniques provides sustained support for WSN development. Existing schemes neither offer the integration in a framework such as *Fabric*, nor are they optimized for the extreme resource-constraints present in WSNs.

6.5.2. Architecture

To enable *microFibre* to optimize the bit-length of payload, the application developer's input needs to be as precise as possible. As described in Section 6.4.1, XML Schema with its powerful expressions, pre-defined data types, and the compelling extension mechanisms is used by *Fabric* to accomplish this task. In the following, a simple example will illustrate the underlying idea of *microFibre*, the steps performed by the system and the resulting output.

Again, imagine an application developer designing a WSN application that helps optimizing the heating of a building. Sensor nodes that are scattered across several rooms measure the temperature (in either degrees Celsius or Fahrenheit) and detect motion. In order to use *Fabric*, the application developer prepares an XML Schema document that describes the application's data structures. Hereby, he carefully specifies the data types and their required accuracy. Figure 6.15 shows an example of how the application described above could structure its data.

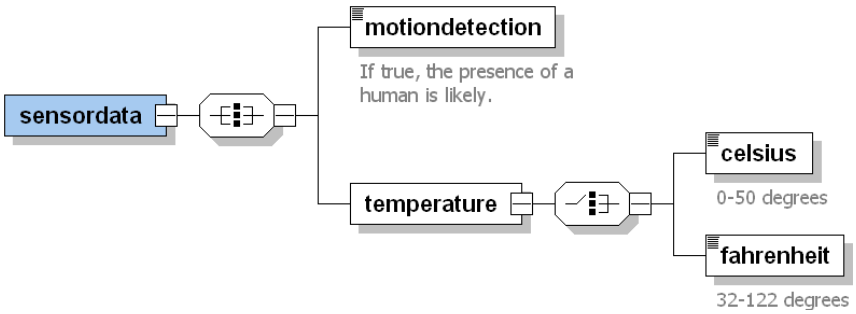


Figure 6.15.: Visual representation of the exemplary data type for the heating control application

A single root element contains an `all`-composition of two other data types (`motiondetection` and `temperature`). While `motiondetection` only contains a single Boolean value, `temperature` is comprised of a `choice` between two integer values (`celsius` ranging from 0-50 and `fahrenheit` ranging from 32-122).

Information Theoretical Considerations

microFibre's goal is to minimize the number of bits that are required to represent an in-memory data structure as payload such that the receiver is still able to decode it back to a semantically equivalent data structure. The fundamental theoretical background and the limits imposed by them were already discussed in Section 2.3. These limits apply given that the symbols emitted by a data source are the result of a stochastic process. However, data in computer programs is typically well-structured and not every element is completely random.

The underlying idea of *microFibre* is to exploit the detailed knowledge about the structure and the possible content of the application's data types. By incorporating this information into the generated middleware, the length of the transmitted messages can be reduced to a bare minimum. This is because both, the senders and the receivers share this knowledge and consequently, it can be omitted in the payload. In order to quantify what is actually meant by bare minimum, this section revives aspects from Section 2.3 and presents where optimization potential exists to achieve bit-length optimized encodings.

In the context of *Fabric*, a detailed structural definition of the application data is available as an XML Schema document (cp. Section 2.2.2). On this basis, *microFibre* integrates information that is not random and may be predetermined into the generated middleware code. This information is left out in the payload because is does not reduce any uncertainty at the receiver (these elements are "certain"). Consequently, only uncertain parts of the in-memory data structure are encoded. Uncertain parts are for instance structural choices allowed by a particular data type (e.g., whether `celsius` or `fahrenheit` is chosen), variable length arrays and strings as well as the actual content of the simple types used or defined by an XML Schema document. These elements can be considered as individual data sources that emit symbols with a certain probability.

Figure 6.16 summarizes this procedure for the example presented above. On the left hand side, a simplified version of the XML Schema document is shown as a tree and the values in brackets denote the possible range of the individual values. Here, some information is static (or "certain") and can therefore be integrated into the middleware at development-time while other elements are only known at run-time. At run-time, the in-memory data structures are serialized into payload such that only the uncertain elements are encoded. The receiver uses the received payload in conjunction with its knowledge on the certain parts and recreates the in-memory data structure.

Since the possible range for both temperature values is known, only the uncertain value ranging from the minimum to the maximum value must be encoded. This technique can be applied to all integer values including structural choices, array and string lengths as well as integer variables. Section 2.3 introduced two fundamental methods to arrive at the shortest possible representation of symbols emitted by a data source. When all symbols x_i are equally likely ($p(x_i) = \frac{1}{n}$), each symbol can be represented using $n = \lceil H_{max}(X) \rceil = \lceil log_2 n \rceil$ bits (cp. Definition 3 on page 32). If not all symbols are equally likely but their probabilities are known a-priori, a Huffman Code achieves an optimal encoding.

(De-)serialization Scheme

In the following, we present how *microFibre* uses both techniques to realize a bit-length optimized encoding while producing lean middleware code. This process is illustrated at the example of the data type shown in Figure 6.15. Figure 6.17 depicts the XML Schema that represents this data type along with its attached

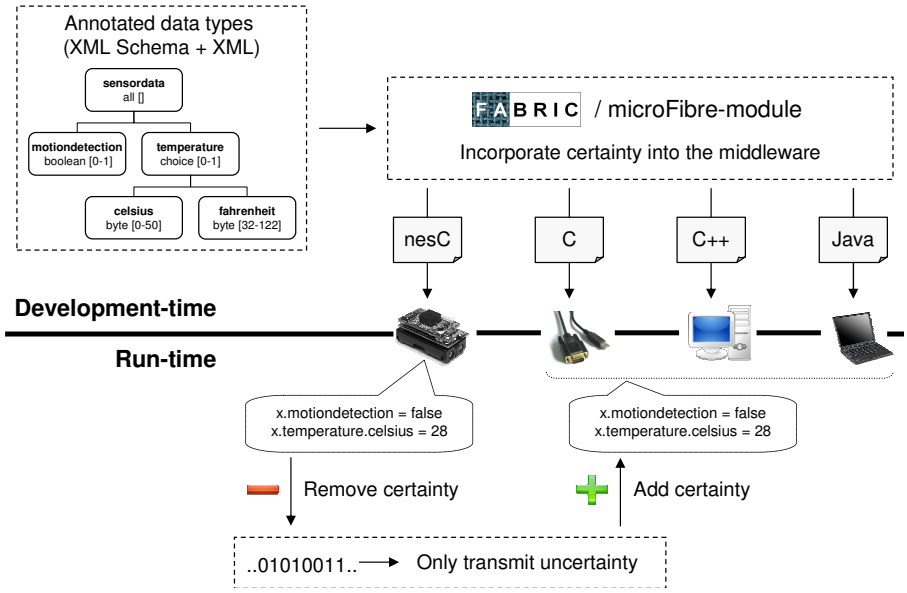


Figure 6.16.: Development- and run-time architecture of *microFibre*

annotations. The XML Schema elements are prefixed with `xs:` whereas the elements containing the type annotation start with the prefix `fabric:`. To recall, the root elements of an XML Schema document represent the data types that are handled by the generated middleware (cp. Section 6.4.1). In this example, the generated middleware will therefore only contain a single type called `sensordata`.

At the core of XML Schema, there are two different kinds of data type definitions: simple and complex types. A variety of *built-in simple types* are standardized that can be categorized into integer and real numbers as well as strings. These built-in types can be further restricted to user-defined simple types (e.g. by limiting their range) using so-called *facets*. When dealing with numerical data types, restrictions may be applied by specifying a minimum and maximum value. Both the lower and upper boundary can be either defined as inclusive (facets `minInclusive` and `maxInclusive`) or exclusive (`minExclusive` and `maxExclusive`) boundaries. Strings, on the other hand, can have a restricted length. Similar to the numerical values a minimum and a maximum or a fixed length (facets `minLength` and `maxLength` or `length`) can be set. *Complex types* may be composed of several simple and complex types. XML Schema supports three different types of compositions: sequence, choice and all. A `sequence` is a data type where the order of the contained elements is fixed. For each element the number of minimum and maximum occurrences may be specified (facets `minOccurs` and `maxOccurs`). A `choice` is similar to a sequence, but contrary to the sequence, only one of these types may be present in any given instance. The `all` composition is somewhat special since the ordering of the contained elements is arbitrary but each element may occur at most once. In addition to the definition of simple and complex types, an XML Schema also defines *top-*

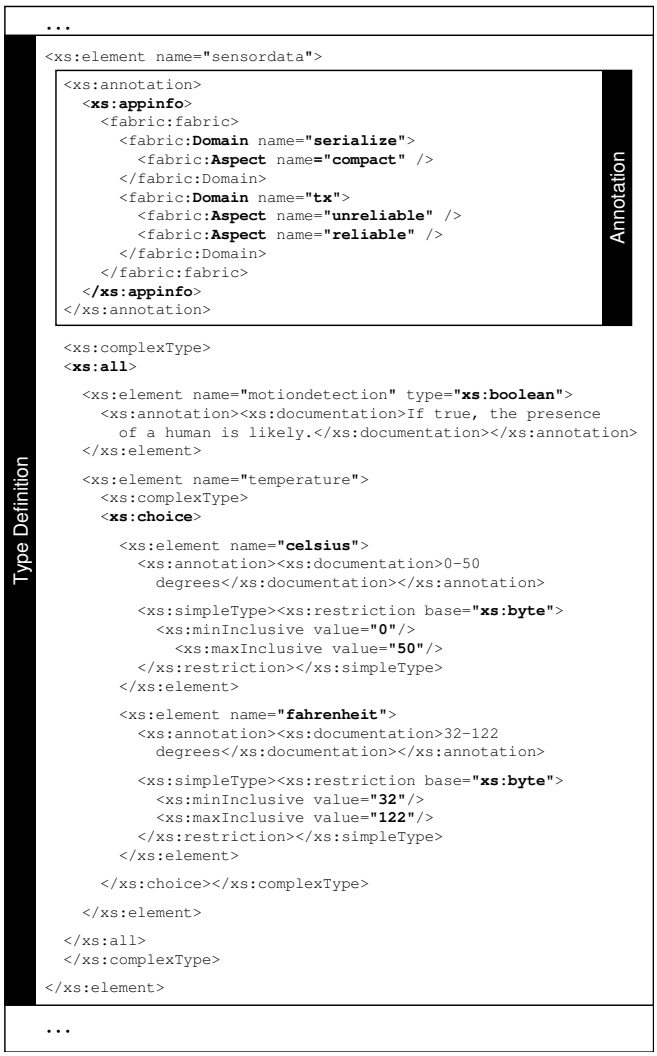


Figure 6.17.: XML Schema document for the exemplary heating control application and its annotation

level elements to be of a certain type (e.g., element *sensordata* on Figure 6.17). For an in-depth discussion of XML Schema and its features, please refer to Section 2.2.2.

First, each global element is converted to a semantically equivalent tree representation (as already discussed in Section 6.4.3 on page 103). The resulting tree provides a representation of the XML Schema document that is suitable for an easy code generation. Figure 6.18 shows the tree that is generated from the XML Schema document depicted in Figure 6.17.

For code generation, a recursive depth-first search through the whole tree is performed. Depending on the visited vertex of the tree, *microFibre* generates code that encodes the value of an element or an attribute (simple type), a structural choice (complex type) or a local element (more precise, how often

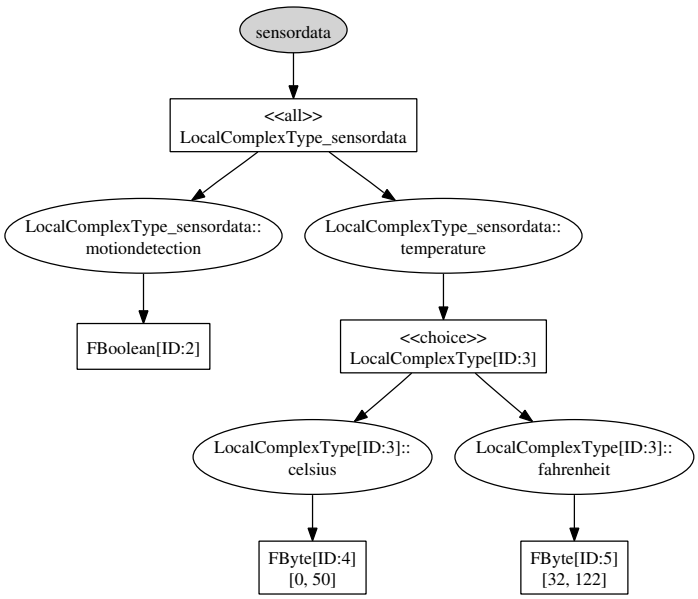


Figure 6.18.: *microFibre*'s internal tree representation of the XML Schema document shown in Figure 6.17

this element occurs).

Simple Types Without any restrictions, all built-in signed and unsigned integers have a fixed size of 8 bits for *byte*, 16 bits for *short*, 32 bits for *int* and 64 bits for *long*. If the range of a type is restricted, its serialization can be compacted. According to Shannon, the number of bits n necessary for transmitting a restricted integer value can be calculated as $n = \lceil \log_2(max - min + 1) \rceil$. When using exclusive boundaries instead, these are transformed to inclusive boundaries by adding/subtracting one. When serializing a restricted value it is normalized by subtracting the lower boundary, which is added again on deserialization. In the example, `celsius` requires $n_{celsius} = \lceil \log_2(50 - 0 + 1) \rceil = 6$ instead of 8 bits saving 25% of the transmitted data. Simple data types such as `date`, `time` or `dateTime` use a string representation of date and time values as standardized by ISO 8601:2004 [75]. For example, "23rd of December 1976, 9:00 pm" is represented as "1976-12-23T21:00" in XML instance documents. To avoid these lengthy strings in the payload, *microFibre* treats these data types as complex types containing multiple simple integer values for the individual elements such as year, month, day or hour.

Real numbers require 32 bits for single (*float*) and 64 bits for double precision (*double*) as defined by IEEE 754 [72]. Reducing the required bits is difficult because of the nature of floating point numbers, where small numbers are encoded with a higher precision after the decimal point than larger numbers. Hence, restricting the range does not decrease the required bits for serialization. Nevertheless, developers can still optimize the serialization of real numbers

by creating user-defined decimal types using XML Schema's `totalDigits` and `fractionDigits` facets. These restrict the number of positions before and after the decimal point. *microFibre* then encodes them as two restricted integer values if this representation requires fewer bits than the standard IEEE 754 representation.

Finally, there are string values, which are encoded as one byte per character. Besides the character data, the string's length must be encoded as well. For applications where the string lengths are bounded to a minimum and maximum value and it is encoded using the same rules as stated above for integers. A special case is a string with a fixed length where the length may be omitted completely. Because WSNs usually run only a single application, additional knowledge about the application's data is available at design time. For instance, the required alphabet and text samples are available prior to deployment. Hence, if this data is included into the annotations, it is used to construct a Huffman tree that is available on all devices and is hence not included in the payload.

Complex Types The encoding of complex types depends on the type of the composition. For `sequence`-compositions, the order of the elements is fixed. Hence, no structural information is required in the payload.

The `all`-composition (`motiondetection` and `temperature` in the example) also requires no representation in the payload since the order of elements can be predetermined at compile time. This is because in contrast to traditional XML Schema based serialization schemes, *microFibre* does not (de-)serialize XML documents, but in-memory data structures. This relieves *microFibre* from the need to encode opening and closing tags or attributes explicitly since the sequence of data may be chosen by *microFibre* where the schema does not dictate a fixed order.

Only the `choice`-composition (between `celsius` or `fahrenheit` in the example) cannot be determined at compile-time and the payload must therefore include the actual selection used at run-time. *microFibre* encodes the selected choice exactly like a range restricted integer with a minimal value $min = 1$ and a maximal value $max = numberOfChoices$.

Elements An element that is part of a complex type may occur more than once, exactly once or not at all in instance documents (or, in this case, in a data structure). The number of possible occurrences is specified using the attributes `minOccurs` and `maxOccurs`. If the number of occurrences is not fixed (i.e. $maxOccurs - minOccurs > 0$), then the actual number of elements needs to be transmitted before the elements themselves. As with choices, a range restricted integer is used. The minimal value is $min = minOccurs$ and the maximal value $max = maxOccurs$.

Code Generation

Integrated as a module for *Fabric*, *microFibre* generates data type definitions as well as (de-)serialization code based on the information contained in the generated tree (cp. Figure 6.18). It is an intrinsic property of WSNs that data is processed on a variety of heterogeneous devices along its way to a destination. The majority of the devices are extremely resource-constrained sensor nodes with only a few ten kBytes of memory. Conserving these scarce resources is therefore essential. Consequently, the generated source code must be lean and should require only very little program memory for the serialization logic as well as a minimal amount of RAM for each in-memory data structure. In the following, generated C code for the above-introduced example is presented, because this is the most ubiquitously used language in the context of WSNs. The current implementation of *microFibre* is additionally capable of nesC, Java and C++ code generation.

Figure 6.19 shows an excerpt of the C header file generated by the framework from the schema document shown in Figure 6.17. In lines 1-11 the data type declaration can be seen, followed by the declarations of the functions which perform the (de-)serialization of the data type (lines 13-17).

When generating the data types in the C programming language, most simple types can be directly mapped to their language dependent counterparts. However, *microFibre* takes care of the differing sizes of these types for different device architectures (e.g., long is 64 bits in XML Schema but depending on the target device only 32 bits in C). Range restricted integers are easily mapped to bit-fields where the number of necessary bits is specified using a colon and the number of bits after the data type definition. Other types such as `boolean` and special types like `dateTime`, `time` and `date` do not have counterparts in C but they as are composed of bit-fields too.

```
1  typedef struct fabric_sensordata {
2    struct  {
3      enum { CELSIUS, FAHRENHEIT } use_to_assign_selected;
4      union {
5        char celsius : 6;
6        char fahrenheit : 7;
7      } choice;
8      unsigned int selected : 1;
9    } temperature;
10   unsigned char motiondetection : 1;
11 } fabric_sensordata;
12
13 unsigned int fabric_serialize_sensordata
14   (fabric_sensordata* dtype, void* buffer, int buflen);
15
16 void fabric_deserialize_sensordata
17   (const void* buffer, int buflen, fabric_sensordata* dtype);
```

Figure 6.19.: Excerpt from the generated source code

Complex types are converted to corresponding language constructs available in C. While `sequence`- and `all`-compositions are represented by a `struct`, a `choice` is represented by a `union` in C. For a `union`, *microFibre* must encode which of the union's elements has actually been set at runtime (`celsius` or `fahrenheit` in the example). For that purpose, an enumeration containing the choices (see line 3 in Figure 6.19) and a selector variable is generated automatically (line 8).

As mentioned before, elements of complex types can have a specified minimum and maximum occurrence value. If one of these occurrences does not equal 1, the child elements are represented as an array of the element's type. If the minimum and maximum occurrences differ, an additional variable is added that encodes the actual number of occurrences at run-time.

Imagine that an application developer has written two different code fragments that fill an in-memory data structure with values. The first one (cp. Figure 6.20(a)) represents a motion detection and a temperature of 79°F. The second one (cp. Figure 6.20(b)) represents no detected motion and a temperature of 26°C. The resulting encoding for both data structures as payload is shown in Figure 6.21.

```
1 fabric_sensordata f;
2 f.motiondetection = 1;
3 f.temperature.choice.fahrenheit = 79;
4 f.temperature.selected = FAHRENHEIT;
```

(a) Motion, 79°F

```
1 fabric_sensordata c;
2 c.motiondetection = 0;
3 c.temperature.choice.celsius = 26;
4 c.temperature.selected = CELSIUS;
```

(b) No motion, 26°C

Figure 6.20.: Exemplary use of the generated data types

The first value that is actually contained in the payload is the contents of `motiondetection`. This Boolean value requires only 1 bit. Then, the choice between `fahrenheit` and `celsius` is encoded. This requires also $n_{choice} = \lceil \log_2(numberOfChoices) \rceil = \lceil \log_2(2) \rceil = 1$ bit. Finally, depending on the selector variable, one of the two integer values is encoded. Note that the value for `celsius` is normalized to the minimum value of 0 while the normalized value of the `fahrenheit` variable is 47 ($= 79 - 32$) because the minimum value specified by the XML Schema document is 32.

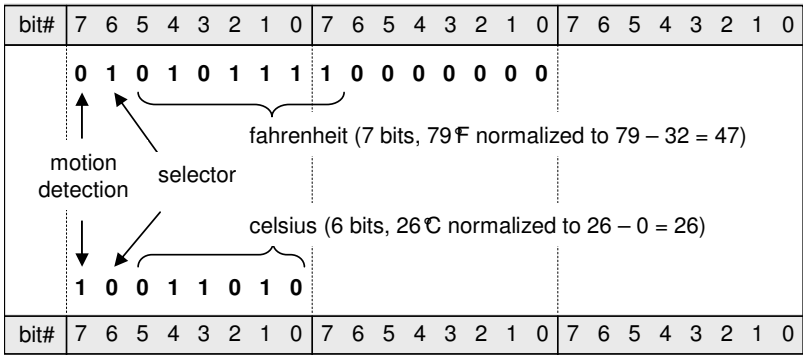


Figure 6.21.: Encoded example message for two distinct cases using *microFibre*

6.6. Traditional Data type Serialization (macroFibre)

When developing applications for sensor networks, the scarce resources of the individual sensor nodes are a constant challenge for application developers. When more and more functionality is added during the development process, the footprint of the application may surpass a node's capabilities and developers must carefully assess where reductions are feasible. The *microFibre* serialization scheme presented in Section 6.5 incorporates additional knowledge on each device to reduce the payload's length. However, for applications where the space constraints outweigh energy considerations, e.g., because communication only occurs infrequently, alternative schemes could be an appropriate choice. The leanest approach to data type serialization is to copy the in-memory representation of a data type byte per byte into the network message payload. However, as discussed in Section 6.5.1, this method has severe limitations in heterogeneous environments.

This section presents *macroFibre*, a simple serialization scheme that complements *microFibre* by focusing on a smaller footprint instead of on a reduced payload length. Application developers using *Fabric* can therefore switch the desired serialization scheme by changing the annotation of the data types. Similar to the *network types* approach discussed in Section 6.5.1, *macroFibre* also addresses device heterogeneity by hiding the effects of different memory alignments and endiannesses from the developer. However, *macroFibre* is not limited to TinyOS and nesC but it is independent from the actual programming language and target platform. In contrast to *microFibre*, *macroFibre* does not optimize the payload's length and aligns the individual elements at byte boundaries while *microFibre* crosses byte boundaries for minimal bit-length encodings. In addition, only simple operations are performed ensuring that heterogeneous devices can exchange messages seamlessly. This means that multi-byte numbers are converted from the in-memory data type representation to a common network byte order. All other data types are directly copied into the network message buffer in a predefined order exactly like *microFibre*.

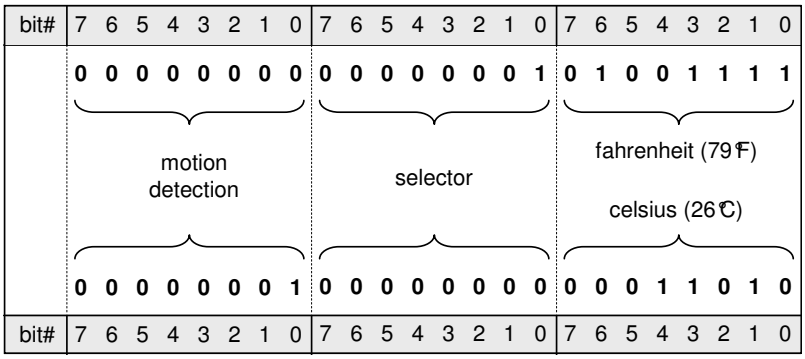


Figure 6.22.: Encoded example message for two distinct cases using *macroFibre*

This process is depicted in Figure 6.22 for the example presented in Section 6.5.2. It shows how *macroFibre* serializes the same data type in a different manner. Here, both `motiondetection` and the selector variable require one byte instead of one bit in the case of *microFibre*. Similarly, `fahrenheit` and `celsius` require one byte while *microFibre* only needs seven/six bits. For this simple data type, *macroFibre* needs three bytes to encode the data type while *microFibre* only requires one byte (8 bits) for `celsius` and two bytes (9 bits padded to two bytes) for `fahrenheit` case.

6.7. Evaluation of microFibre and macroFibre

This section presents measurements that juxtapose encoding quality, footprint and runtime of different approaches. In order to perform this evaluation, it is vital to use data types that represent typical usage scenarios. The evaluation performed here uses three different data types, two from real-world WSN projects and one from a well-known e-commerce application. The WSN projects are used to compare *microFibre* and *macroFibre* with native sensor network implementations. In addition, the use of a data type from an e-commerce application shows that *microFibre*, *macroFibre* and *Fabric* are not limited to embedded WSN devices but can be applied to other application domains as well. The following list summarizes the applications that are used in this evaluation:

- TinyDB (cp. Section 6.3)
- MarathonNet (cp. Chapter 7)
- Amazon.com Web Service [3]

A representative data type from each application is used to benchmark *microFibre* and *macroFibre* against ASN.1's Packed Encoding Rules (PER), Xenia and the corresponding native implementation. This allows comparing handcrafted and automatically generated code with *microFibre* and *macroFibre*. The remainder of this section is structured as follows. Next, the applications and data types that serve as the basis for the evaluation are introduced and it is described

how instances of these data types used for the evaluation were created. Following, the encoding lengths of *microFibre* and *macroFibre* are benchmarked against ASN.1's PER, Xenia and handcrafted code. Finally, the footprint and the memory requirements of *microFibre* and *macroFibre* are compared with generated and handcrafted code.

TinyDB To compare *microFibre* and *macroFibre* with a typical WSN application, the well-known and widely deployed TinyDB (cp. Section 6.3, [107]) sensor network database was chosen. One of its central data types (called `QueryMessage`), which transports queries to sensor nodes is used in the evaluation. It reflects the inherent need of WSN applications to distribute complex data structures as the payload of network messages in the network.

Figure 6.23 presents an excerpt of the original C source code that defines the QueryMessage data type. These C-language data structures were converted to equivalent representations in XML Schema and ASN.1. The XML Schema was created manually and the ASN.1 representation was generated using an XML Schema to ASN.1 converter[2] available online. The ASN.1 document was then compiled to C code using the ASN.1 compiler *asn1c*[3]. The complete C data structure is shown in Section A.2.1, the XML Schema in Section A.2.2 and the ASN.1 representation in Section A.2.3.

```
1  /** Message type for carrying query messages */
2  typedef struct QueryMessage {
3    uint8_t qid; //8 -- note that this byte must be qid
4    ...
5    char type;  //is this a field, expression,
6                //buffer, or event msg -- 18
7    union {
8      Field field;
9      Expr expr; //40
10     BufInfo buf;
11     char eventName[COMMAND_SIZE];
12     short numEpochs;
13     int8_t ttl;  //for delete msg
14   } u; //40
15 } QueryMessage, *QueryMessagePtr;
```

Figure 6.23.: Excerpt from TinyDB's definition of the QueryMessage data type

Speaking in terms of XML Schema, `QueryMessage` comprises a number of simple types as well as nine complex types (`Field`, `Expr` and `BufInfo` in the figure, which in turn contain other complex types). The way this data type is specified, it can be assembled in 78 unique combinations (because of contained choices and arrays of variable length).

To indicate which of the choices has actually been selected in an instance of this

[2] xsd2asn1 project: `http://asn1.elibel.tm.fr/tools/xsd2asn1`
[3] Open Source ASN.1 Compiler asn1c: `http://sf.net/projects/asn1c`

data type, a variable is necessary that contains this choice. One example of such a variable is shown in Figure 6.23: `type` specifies whether the content of the union `u` contains `field`, `expr`, `buf`, `eventName`, `numEpochs` or `ttl`. However, a generic serialization scheme can not be aware of this special meaning of `type`. Therefore, it needs to generate own selector variables.

The amount of data contained in the different variations of this data type varies heavily. For instance, if `ttl` is selected, the union (or choice) `u` only holds one byte. However, if the `expr`-field is chosen, `u` holds 23 bytes. Yet, TinyDB's networking code directly copies the in-memory data structure into the network buffer and hence always requires the same amount of memory independent of the actually contained data. Without any compiler optimization, `QueryMessage` requires 48 bytes. Using a compiler optimization offered by the GCC compiler that aligns bit-fields at bytes boundaries instead of 4-byte boundaries, this gives a payload length of 42 bytes.

MarathonNet The MarathonNet application is prototypical for a different class of applications. While TinyDB uses quite complex data types, MarathonNet is a sense-and-forward application that uses comparatively simple data types to convey its data to a final destination. Before switching to *microFibre*, an early version of the MarathonNet application used a simple, human-readable payload format. Figure 6.24 shows an excerpt of the C-code that converts the in-memory data into the payload of network messages. The `snprintf` function is used to create a textual representation of the individual integer values and to frame them between a start and end sequence. The corresponding annotated XML Schema document for the use with *Fabric* is shown in Figure 7.11 on page 139.

```
1 memset(tx_buffer, '\0', TX_BUFFER_LEN);
2 snprintf((char *) tx_buffer, TX_BUFFER_LEN-1,
3      "%c%c%c%u;%u;%u;%u;%u%c%c",
4      0x10 /* DLE */, 0x02 /* STX */, 0x03 /* Type 3 */,
5      _race_time, _pm_id, _heartrate, _position, _speed,
6      0x10 /* DLE */, 0x03 /* ETX */
7      );
```

Figure 6.24.: Excerpt from an early prototype of the MarathonNet application

This format can be parsed easily on different devices and programming languages at the cost of lengthy payloads. Image a runner with the following properties: id 1, position 21,095m after 7,200s (2h), heart rate 120bpm. The transmitted character sequence (0x10 0x02 0x03 7 2 0 0 ; 1 ; 1 2 0 ; 2 1 0 9 5 ; 0 0x10 0x03) already requires 23 bytes.

Amazon.com A Web Service [189, 190] is a software component whose API is defined using XML Schema documents. To invoke a Web Service, an XML instance document is sent to a software component, which in turn replies with

another XML instance document. A common problem of exchanging XML as payload is that they are very bulky. As discussed in Section 6.5.1, approaches such as Xenia propose alternative, compact representations that compress these XML documents before they are transmitted.

Web Services are therefore related to *Fabric* and *microFibre*: They also use XML Schema documents to describe the exchanged data structures and approaches for reducing the length of transmitted data are an important means to increase the efficiency of the system. The key difference is that *microFibre* compresses in-memory data structures while these approaches compress XML documents. However, the comparison with this research branch demonstrates that *microFibre* can compete with approaches intended for larger computing devices while providing essentially the same functionality in WSNs.

For the following evaluation, the XML Schema document describing the API of the well-known e-commerce platform Amazon.com is used. It offers a Web Service providing an automatic access to the services offered by Amazon.com's web site [3]. The XML Schema document describing this API comprises more than 25 complex types and 100 top-level elements.

Generation of Test Data for the Evaluation When comparing the encoding quality of *microFibre* and *macroFibre* with other approaches, it is not sufficient to use only a few random instances. As mentioned above, the amount of data contained in the different possible combinations varies heavily. For the comparatively simple data types such as the one from TinyDB, the number of unique combinations is bounded to a few hundreds. However, the data type from the Amazon Web Service allows an unlimited number of unique combinations. In order to compare the different approaches with each other, instances of the used data types are required in different formats:

- TinyDB and MarathonNet: Handcrafted data structures
- Xenia and the Amazon.com Web Service: XML documents
- ASN.1 and *microFibre*: Generated data structures

Therefore, the generation of test data required two steps:

1. Generate different instances of these data types
2. Convert these instances to different formats

Generating test cases for the 78 unique combinations of the QueryMessage data type was performed by exploiting a feature of the *microFibre* and *macroFibre* modules. Both optionally generate code that creates test instances automatically, which normally allows for an automatic validation of their implementation. The generated test code fills these test instances with random values for the integer and real data types. Strings are filled with data that matches the character frequencies of the internally generated Huffman-Tree. For this evaluation, the Huffman-Tree has been parameterized to use the character fre-

quencies of the English language [51]. The MarathonNet data type contains no structural choices. Hence, 120 different value sets obtained from a real-world deployment [63] were used to generate test instances. For the Amazon Web Service, three different XML instance documents returned by this web service were used[4].

Converting these test instances was performed mostly automatically. This is possible because in addition to *microFibre* and *macroFibre*, we have implemented two modules that perform the conversion from in-memory data structures to XML and XER and vice versa. Test instances available in data structures generated by *Fabric* are therefore immediately available for the use with ASN.1 and XML compatible tools such as Xenia. The payload length required by MarathonNet was obtained manually by invoking the methods shown in Figure 6.24. TinyDB copies the in-memory data structure into the payload of network messages. As mentioned above, the payload' length is 42 bytes independent of the test case.

Payload Length

Figure 6.25 and Table 6.2 show the number of bytes required by TinyDB, ASN.1 (PER), Xenia, *macroFibre* and *microFibre* to encode the 78 TinyDB test instances. As mentioned above, this comparison is slightly biased in favor of TinyDB since some variables are encoded redundantly by the other schemes. The actual amount of is hard to grasp since the existing variables are not clearly distinguishable from other variables. Hence, the horizontal line at 42 bytes indicates the length of TinyDB's payload but it cannot serve as a direct benchmark for the other approaches.

	Payload length [byte]			Savings achieved by *microFibre* [%]		
	Min	Max	Avg	Min	Max	Avg
microFibre	16	40	30			
ASN.1 PER	16	54	34	0.0	34.0	10.3
Xenia	17	49	35	5.6	24.4	13.3
macroFibre	20	61	40	14.9	42.6	23.2
TinyDB	42	42	42	4.8	61.9	28.8

Table 6.2.: Summary of the payload lengths and the achieved savings of *microFibre* (QueryMessage)

It is clearly visible that *microFibre* and PER yield nearly identical payload lengths for the majority of test instances. This is because mostly integers (with or without any range restriction) are encoded and the schemes achieve an optimal encoding for these values. Xenia produces slightly longer encodings since it does not optimize the encoding of range-restricted integers. In the cases where *microFibre* achieves savings of multiple bytes compared to PER and Xenia, one

[4]By courtesy of Werner et al. [182, 183]

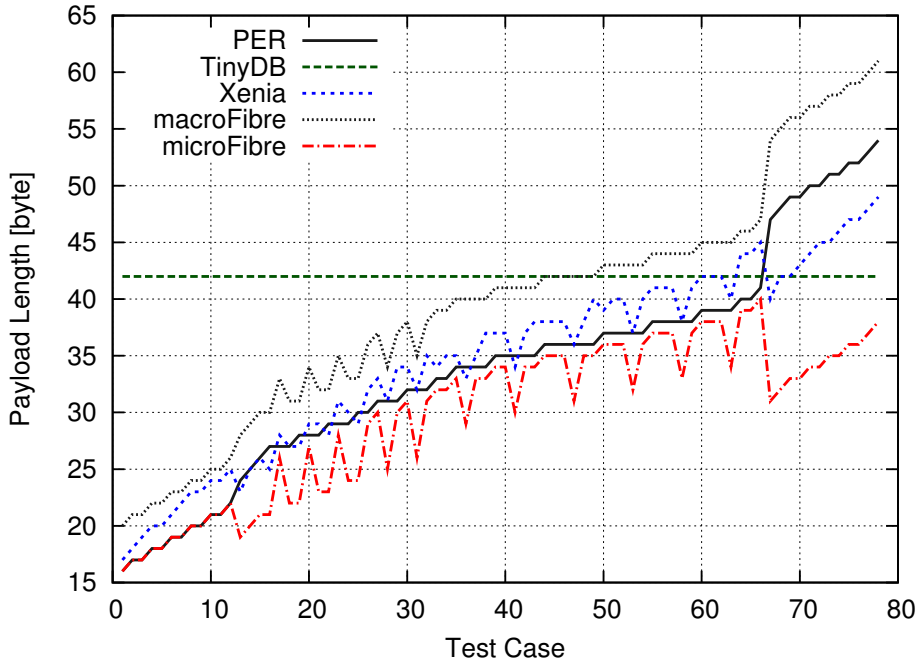


Figure 6.25.: Payload length of encoded QueryMessage test instances

or three string values are contained in the test instances. While PER encodes each character using one byte, Xenia uses a custom Huffman-tree that differs from the one used by *microFibre*. Hence, Xenia encodes string values more efficient than PER and very similar to *microFibre*, only the utilized character frequencies differ. Encodings produced by *macroFibre* are on average 23.2% longer (min. 14.9%, max. 42.6%) than the ones created by *microFibre* since *macroFibre* aligns individual variables at byte boundaries and does not exploit any optimization potential such as minimum or maximal values of variables or Huffman-based string encodings. This shows the optimization potential compared to traditional handcrafted networking code.

The evaluation results for the MarathonNet application are depicted in Figure 6.26 and summarized in Table 6.3. *microFibre*, *macroFibre*, PER, and Xenia always require a constant amount of bytes (7 and 16 for Xenia) to encode the 120 test instances while the implementation of MarathonNet requires on average 23 byte (min. 15, max. 28 byte). In contrast to TinyDB, this data type contains no choices and only (range restricted) integer values are encoded. Hence, only the textual representation used in the early MarathonNet implementation requires a varying amount of bytes.

It can also be seen the range restrictions of the four integer values does not yet lead to length savings of *microFibre* compared to *macroFibre*. Xenia requires 16 byte instead of 7 byte since it directly translates the data type of the XML Schema to a required number of bytes while *microFibre* and *macroFibre* use the actual value range of the data type to determine the number of bits/bytes

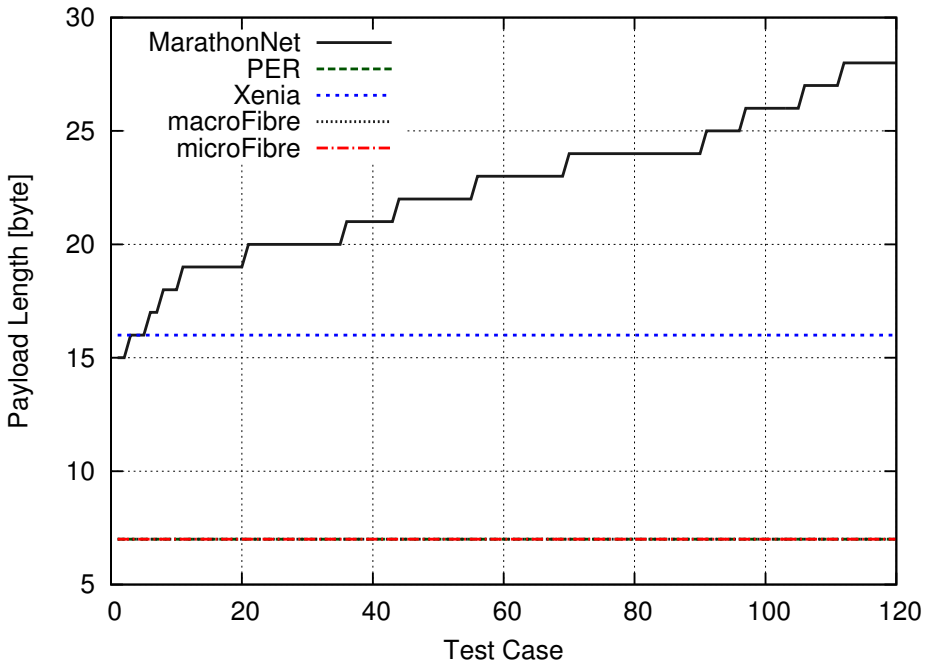


Figure 6.26.: Payload length of encoded MarathonNet test instances

required for a data type.

	Payload length [byte]			Savings achieved by *microFibre* [%]		
	Min	Max	Avg	Min	Max	Avg
microFibre	7	7	7			
ASN.1 PER	7	7	7	0.0	0.0	0.0
macroFibre	7	7	7	0.0	0.0	0.0
Xenia	18	18	18	61.1	61.1	61.1
MarathonNet	15	28	23	53.3	75.0	68.4

Table 6.3.: Summary of the payload lengths and the achieved savings of *microFibre* (MarathonNet)

The results for the three test instances of the Amazon.com Web Service are shown in Figure 6.27. Compared to the TinyDB and the MarathonNet application, the corresponding XML Schema document of the Amazon.com Web Service does not exploit any optimization potential by restricting the value range of integers, string lengths or arrays. It also uses a large amount of string values that dominate the size of the three XML test instances. As a result, the payload lengths are also dominated by the size of the encoded string values. For this evaluation, *microFibre* has been parameterized with two different character frequencies for the Huffman Tree: the frequencies used in the evaluations above and frequencies calculated by evaluating the contents of five random books written in English that were available on the Internet.

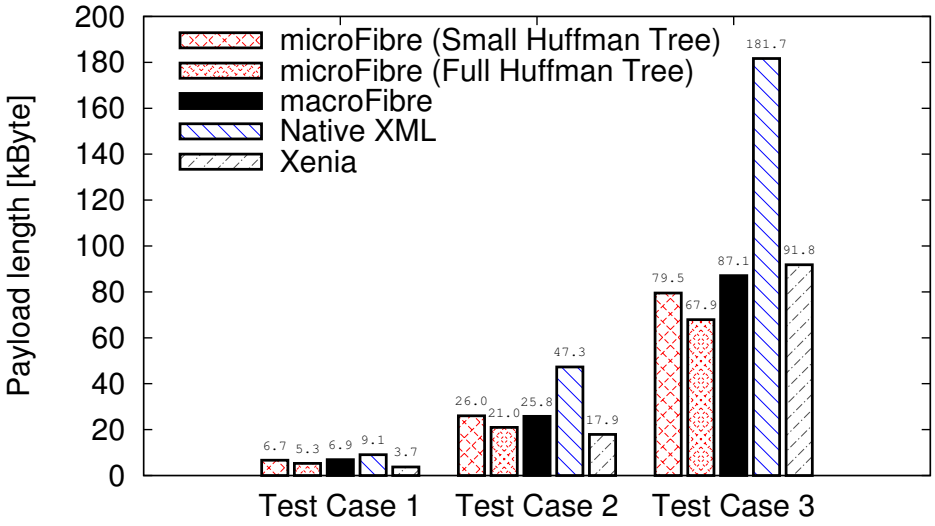


Figure 6.27.: Payload length of encoded Amazon test instances

For this evaluation, Xenia used a technique called *Adaptive Huffman Coding* [145]. Compared to a static Huffman Tree, this technique modifies a Huffman Tree while encoding a document to adapt to the actual character frequencies. It can be seen that Xenia, *microFibre* and *macroFibre* provide similar compression results. While *microFibre* performs slightly better for test instance 3, Xenia encodes the smaller test instances 1 and 2 more efficiently. However, the differences are due to different string encodings as demonstrated by the two different Huffman Trees used by *microFibre*.

Footprint and CPU Requirements

The downside of compression schemes is that minimizing payload length directly translates to additional code required on the devices (as indicated in Figure 6.16). As a result, such schemes are only useful if the footprint of the compiled code matches the resource constraints of sensor nodes. This section evaluates the required program memory, RAM and CPU time. We compare *microFibre* and *macroFibre* with PER and, where feasible, with the native implementations. We have not evaluated the code generated by Xenia since it additionally requires a library to parse XML documents. These libraries are not readily available on sensor nodes and would dominate Xenia's footprint.

Figure 6.28 shows the RAM required by instances of the different data types and the program memory required by compiled (de-)serialization code. The code was compiled using the standard GCC [53] since this is used by the majority of WSN hardware platforms. In the case of TinyDB, a direct comparison is not feasible since the implementation only passes a pointer to TinyOs' networking code. The same applies to the MarathonNet implementation, which relies on

features of C's standard library.

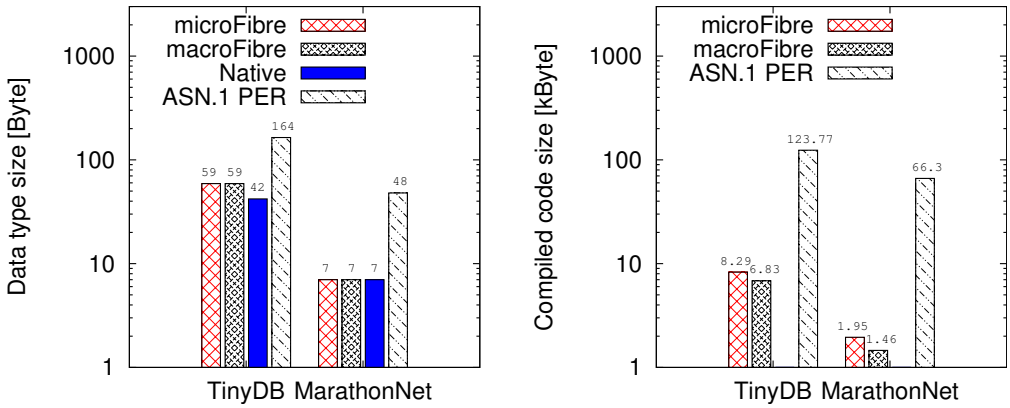


Figure 6.28.: Footprint (RAM and program memory) requirements

microFibre's and *macroFibre*'s generated data types are only slightly larger than the handcrafted TinyDB data structure. As discussed above, this is because of additionally generated variables. The data structures generated by the ASN.1 compiler are approximately four (TinyDB) to seven (MarathonNet) times larger than the TinyDB data structure because it contains various internal state variables. Looking at the size of the compiled code, the code generated by the ASN.1 compiler is clearly not optimized for resource constraint environments. It is already larger then the program memory offered by most WSN hardware platforms and leaves no room for the actual application. The code generated by *microFibre* is 21.4% (TinyDB) and 25.2% (MarathonNet) larger than the code generated by *macroFibre*. As a result, the overhead introduced by *microFibre* compared to traditional, handcrafted solutions is low and *microFibre* is therefore clearly suitable for WSN devices.

Finally, we evaluated the CPU time required by the different approaches. These tests were conducted on standard PC equipment[5] since the code generated by ASN.1 was too large to fit on WSN devices. Figure 6.29 lists the amount of CPU time required by the different approaches to serialize an in-memory data structure to payload and back again. These values shown here are average values of one million iterations.

In the case of TinyDB's QueryMessage data type, which represents a complex data structure with many choices, *microFibre* requires considerably more time to encode the test instance than *macroFibre* and the native implementation. In the case of MarathonNet, *microFibre* and *macroFibre* have similar CPU requirements and both require even less CPU time than the native implementations. The code generated by the ASN.1 compiler requires a multiple of the other approaches in both cases.

[5]Intel Pentium M processor, 1.86GHz, 1GB RAM

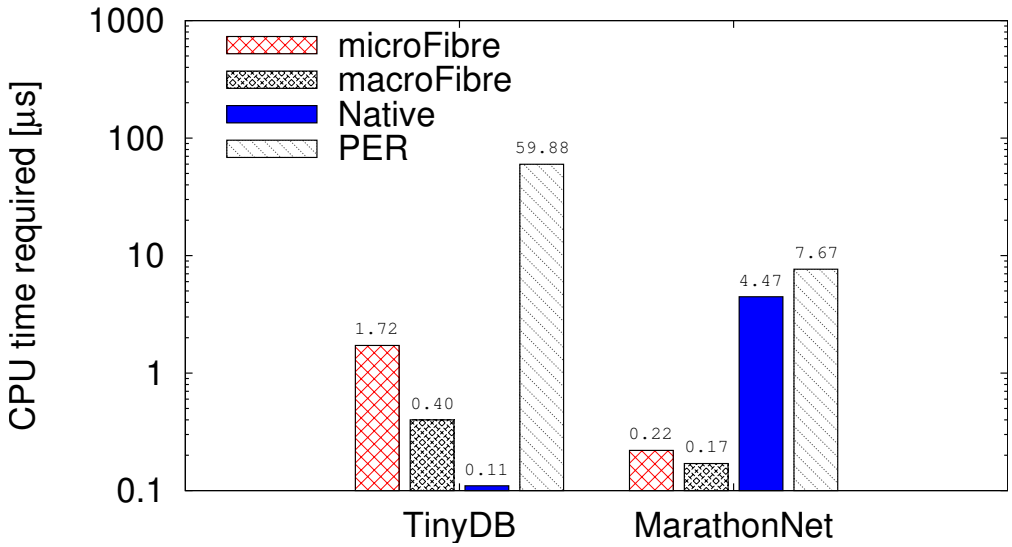


Figure 6.29.: CPU time required for one de-serialization loop on standard PC equipment

6.8. Summary

This chapter introduced our novel middleware synthesis framework called *Fabric*. *Fabric* supports the generation of custom-tailored, application-specific middleware for heterogeneous platforms. It enables a seamless exchange of messages between all target platforms involved in the development of a WSN including sensor nodes, gateways, backend systems, visualization environments and simulation tools. This relieves application developers from dealing with low-level networking aspects and heterogeneity. Instead of handcrafting data types and networking code repeatedly for each application, hardware platform and programming language, *Fabric*'s approach is based on a platform-independent data type specification augmented with *annotations*.

Application developers provide an XML Schema document that is annotated with an XML document per element declaration. These contain the treatment aspects for each data type. Aspects define for each data types how it is treated by the generated middleware. Hereby, the *data type-specific treatment* defines how an in-memory data structure is converted to payload and how payload of a specific type is handled by the generated middleware. For instance, a data type may be compressed and transmitted reliably. A *target description* defines properties of the target platform such as programming language and hardware device. An optional integration into the Eclipse IDE supports the application developer at each step of this process by providing graphical editors for data types, annotations and target specifications.

Framework developers back these annotations with source code generating *modules*. This automates the implementation of recurring tasks while leaving application developers in full control of the features of the middleware. Unlike

traditional middleware systems, code generation tools and middleware solutions for WSNs, *Fabric* explicitly addresses heterogeneity and resource-constraints. In addition, the generated middleware can perform optimizations that are an optional, frequently omitted step in manual implementations. We have demonstrated this at the example of two modules (*microFibre* and *macroFibre*) that generate (de-)serialization code. *microFibre* implements our novel scheme for bit-length optimized payload and *macroFibre* generates code that resembles traditional, handcrafted methods.

The presented measurements show that *microFibre* provides the unique combination of encoding quality, small footprint and acceptable execution overhead, which is vital for resource-constraint sensor nodes. Hereby, the achieved compression ratio competes with state-of-the-art techniques such as ASN.1's PER and Xenia and yields notably shorter payloads compared to handcrafted implementations. For instance, *microFibre* requires on average 23% less bytes to encode the QueryMessage data type than the corresponding manual implementation represented by *macroFibre*. This comes at the price of 20% increased footprint and a runtime increased by the factor of four. However, since the wireless interface consumes considerably more energy than local computations, this is a sustainable tradeoff. It is important to note that the achievable compression ratio highly depends on the input supplied by the application developer. As a result, application developers must carefully design their data types to benefit from *microFibre*.

Integrated optimizations performed by the middleware are only one example of how *Fabric* improves the development process of WSN applications. However, other aspects such as increased reliability of the WSN application, improved development speed or an eased integration of changes are hardly quantifiable.

The *reliability* of WSN applications is supported by *microFibre*, *macroFibre* and the two other XML and XER modules. They optionally generate code that performs a self-validation by creating test cases for the possible combinations of a data type. Each test instance is serialized, de-serialized and then compared with the original test instance to ensure a correct behavior of the generated code. Application developers can therefore rely on the correctness of code generated by these modules, which restricts the search space for bugs in the application and generally makes the application more robust.

Furthermore, *Fabric* has the potential to *speed up the development process* of WSN applications. Imagine an application developer that implements code generated by *microFibre* manually. To estimate the required time for the implementation of this code, the well-known *COnstructive COst Model* (COCOMO) proposed by Boehm [14, 15] can be used. COCOMO estimates the time and the costs required to implement industry-grade software based on the number of lines of code. For the simple example of the QueryMessage data type, this already yields 1.62 person-months for one target platform[6]. Consequently, automatically generating this code can help to increase the development speed.

[6]Generated using David A. Wheeler's SLOCCount, `http://www.dwheeler.com/sloccount/`

7. Case Study: Real-world Application in MarathonNet

To put the development framework presented in Chapter 3 in concrete terms, this chapter demonstrates how it has been applied to realize a real-world WSN project called MarathonNet [63, 103, 129]. In the past few years, Marathon running has evolved from a sport for a small group of fanatics into a sport for the masses where big city marathons register record numbers of more than 40,000 runners and 1,000,000 spectators.

These observations led to the project MarathonNet, which aims not only at researching algorithms, software and hardware but also especially at gaining practical experience with large-scale sensor network deployment and operation. The project's vision is to monitor runners in real-time during competitions. As shown in Figure 7.1, the runners wear special sensor nodes called *pacemates* that measure their heart rate. The pacemates forward this data augmented with time and position information in the WSN to base stations along the track, which pass the received data to a central database.

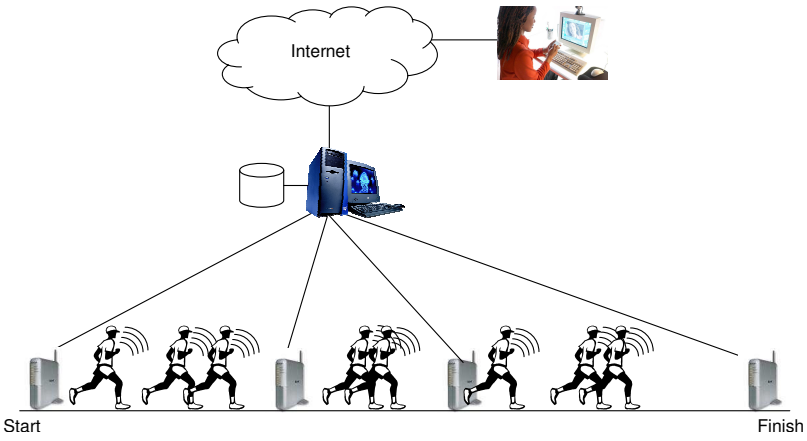


Figure 7.1.: Hardware architecture of the MarathonNet project

This allows spectators to follow the race progress of their friends on the Internet or to watch a real-time visualization of their race progress. Also the notification of friends at the track via SMS when a particular runner is approaching is possible, as well as queries with regard to the location of a runner. Coaches of the athletes can perform a post-facto analysis of the race using the health-condition of their protégés augmented with time and position information to

optimize the strategy for future races. Organizers can direct rescue staff to individual runners when critical values are observed, cheating (leaving track, taking short cuts, etc.) can be detected and road closures can be optimized. Runners can be notified on the current position of competitors or a virtual runner may be used as a reference for his own target speed.

In the following, Section 7.1 gives details on how the development framework presented in Chapter 3 helped in realizing the MarathonNet application. Section 7.2 presents a discussion and evaluation of this development framework and concludes with a summary.

7.1. Application of Shawn, SpyGlass and Fabric

The MarathonNet scenario presents an ideal platform for applied WSN research where the runners form a large, highly mobile sensor network. The architecture of the MarathonNet application (cp. Figure 7.1) consists of three central components:

- Sensor nodes worn by the runners that collect biometrical data and forward it, if necessary via other nodes, to base stations
- Base stations positioned along the track that receive data from passing by sensor nodes
- Central backend storing and visualizing data received from the base-stations

MarathonNet is therefore characteristic of a typical heterogeneous WSN application as introduced in Chapter 1. It requires that multiple, highly different types of devices (sensor nodes, gateways, backend systems) cooperate to solve the application's task. As a result, the applications running on these devices must share a common knowledge on the contents of the payload contained in exchanged network messages. The project is also typical in that it requires simulations prior to real-world deployments and visualizations of data received from the gateways. The remainder of this section presents how Shawn, SpyGlass and Fabric were used to prepare the real-world deployments in Flensburg and Ratzeburg in 2006 and 2007.

7.1.1. Shawn

Due to the special requirements, neither the sensor nodes nor the base stations were readily available off-the-shelf. Designing these hardware components as well as novel protocols and applications required a profound understanding of the underlying structure of the highly dynamic network formed by the runners. This included the modeling of a Marathon race inside the simulation environment and an analysis of the underlying network topology and connectivity.

Based on the obtained results, the pacemates and base stations were designed

and manufactured. After the finalization of the technical specifications, the time span until the delivery of the first prototypes constituted approximately half a year. Consequently, developing the software in parallel to the hardware was mandatory to ensure a continuous progress of the project. The goal was to finish the implementation before the first devices were available for test-deployments. Hence, simulations had to replace the real hardware.

The different goals of the simulations resemble exactly the development cycle encouraged by Shawn (cp. Section 4.2): Starting from an idea for a project, the scenario is analyzed in detail to develop a deeper understanding of the underlying structure. Following, algorithms and protocols are evolved gradually from simple centralized implementations to fully distributed implementations. Because of these features, Shawn was used to model the marathon inside a simulated environment, to derive connectivity information from the WSN formed by the runners and to implement and test the application before the pacemates were available. These three steps are described in the following.

Modeling Marathon Races Modeling a marathon race as a set of mobile sensor nodes required three steps:

- Representation of the marathon track as a polygon shape
- Providing split times for each runner
- Implementation of a custom Node Movement (cp. Section 4.3.1) that uses track data and split times to calculate the current positions of nodes

To represent a Marathon track, a custom XML file format has been defined that holds (x,y)-coordinates of the track. These were obtained from online maps of the city of Hamburg and Berlin by determining (x,y)-pixel positions of the track. The resulting polygon shape is then scaled to a specific overall length. Figure 7.2 shows an excerpt of the XML file representing the Hamburg Marathon track. The individual (x,y)-coordinates of the polygon shape are listed sequentially and the length of track is specified as 42195m.

```
1  <track>
2
3   <header description="Hamburg Marathon Track from 2005"
4     length="42195"
5     track_data_type="coordinates" />
6
7   <track_data>
8    <coordinate x="857" y="271"/>
9    <coordinate x="857" y="196"/>
10   <coordinate x="802" y="93"/>
11   ...
12   <coordinate x="902" y="189"/>
13   <coordinate x="858" y="235"/>
14   <coordinate x="857" y="271"/>
15  </track_data>
16
17 </track>
```

Figure 7.2.: Model of the Hamburg Marathon 2005 track

To calculate a runner's position on the track for a specific point in time, realistic data was required. Most large events provide split times of each runner, which are publicly available on the Internet after the race. Split times provide the time at which a runner passed one of the checkpoints that are typically positioned in intervals of 5km or 10km. By using the split times of popular marathon events such as the Hamburg and the Berlin Marathon 2005, enough realistic data was available to model the movement of individual nodes in the simulation.

Because the split-times are a property of individual runners/nodes, they were attached as Tags (cp. Section 4.3.3) to the nodes in Shawn. Figure 7.3 shows an excerpt of an input file using Shawn's persistence format (cp. Section 4.3.3).

```
<scenario>
    <snapshot id="0">
        <node id="Doe, John">
            <location x="0" y="0" z="0"/>

            <tag type="double" name="start_offset" value="0.0" />
            <tag type="map-double-double" name="split_times" >
                <entry index="0" value="0" />
                <entry index="10000" value="1807.0" />
                <entry index="20000" value="3606.0" />
                <entry index="21097.5" value="3801.0" />
                <entry index="30000" value="5408.0" />
                <entry index="40000" value="7243.0" />
                <entry index="42195" value="7658.0" />
            </tag>
        </node>

        <node id="Selavie, Rose"> ... </node>
        ...
    </snapshot>
</scenario>
```

Figure 7.3.: Runners from the Hamburg Marathon 2005 along with their split times represented in Shawn's standard persistence format

It contains nodes and their properties such as name, initial position and attached Tags. Hereby, the Tag named `start_offset` denotes the difference in seconds between the official start of the race and the point in time when the runner actually crossed the starting line. The Tag named `split_times` holds a map that assigns a 1D-position on the track to a time in seconds. This file is read by Shawn's `load_world`-task that creates the contained nodes at their initial position and attaches their corresponding Tags. Figure 7.4 shows a configuration file that creates the Simulation Environment, loads the runners with their split times as well as the track data and runs the simulation.

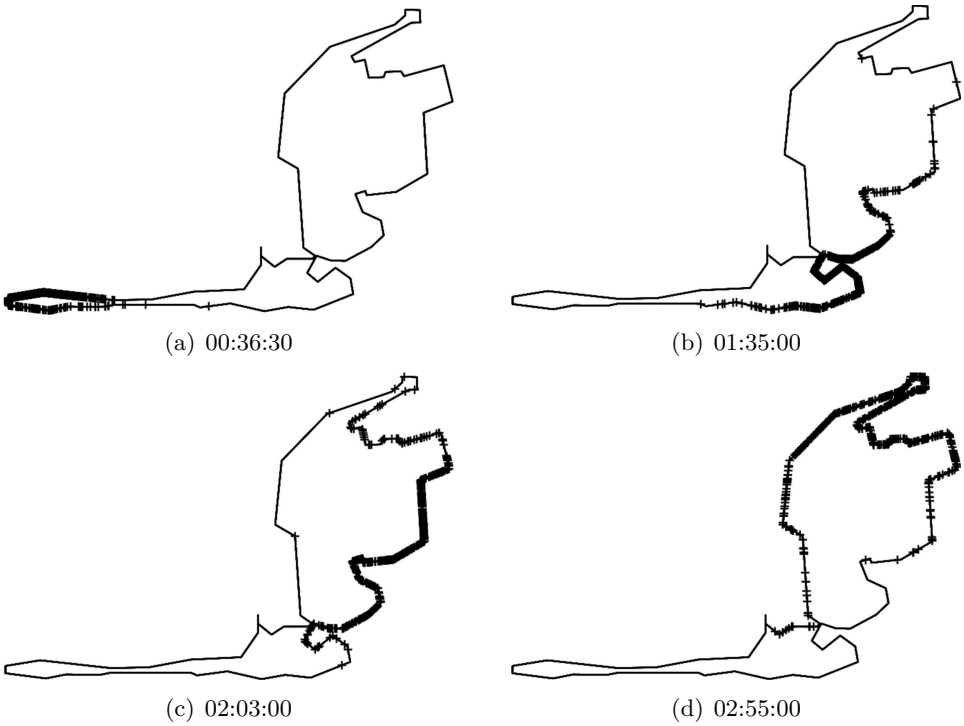
```
#Create the Simulation Environment
prepare_world edge_model=disk_graph comm_model=simple
    range=150 transm_model=reliable

#Load the runners and their split times
load_world file=HamburgSplitTimes2005.xml

#Load track, create NodeMovement and attach it to the runners
mnet_race_task track=HamburgTrack2005.xml

#Run for 6 hours (21600s) of simulated time
simulation max_iteration=21600
```

Figure 7.4.: Configuration for the simulation of the Hamburg Marathon 2005

This data is then available in Shawn and MarathonNet's Node Movement implementation uses the split times of each node to interpolate its current position. To calculate the current position from the list of split times, the two split times immediately before ($split_1$) and after ($split_2$) the current point in time are used. The current position is then calculated as

$$pos = pos(split_1) + \frac{pos(split_2) - pos(split_1)}{time(split_2) - time(split_1)} * (now() - time(split_1))$$

Figure 7.5 shows the track of the Hamburg Marathon 2005 along with the positions of a subset of the participants for different points in time.



(a) 00:36:30 (b) 01:35:00

(c) 02:03:00 (d) 02:55:00

Figure 7.5.: Snapshot of the marathon field for different times (500 runners from the Hamburg Marathon 2005)

Connectivity Analysis The necessity of a real-time data transmission from the pacemates to the backend database requires that a multi-hop path from each pacemate to an arbitrary base station exists for the majority of the time during a race. This connectivity obviously depends on the transmission range of the sensor nodes and the number of base stations. Hence, simulations with different realistic values were conducted to determine the best combination of both parameters.

Since the budget of the project restricted the number of pacemate devices to a few hundred, a random subset of runners from the Hamburg and Berlin

marathon was used. During a simulation, custom Simulation Tasks evaluated the connectivity of the underlying graph representation of the dynamic network formed by these runners. This allowed an extraction of relevant data without requiring any message exchanges or distributed protocols.

Figure 7.6 depicts one result of these simulations. It shows for 500 randomly selected runners the average percentage of time during which a path to an arbitrary base station exists. It is clearly visible that to achieve connectivity for more than 80% of the time and a transmission range of 150m, a minimum of 8 base stations is necessary. Increasing the number of base stations does not yield a significant increase in connectivity; only increasing the transmission range is helpful.

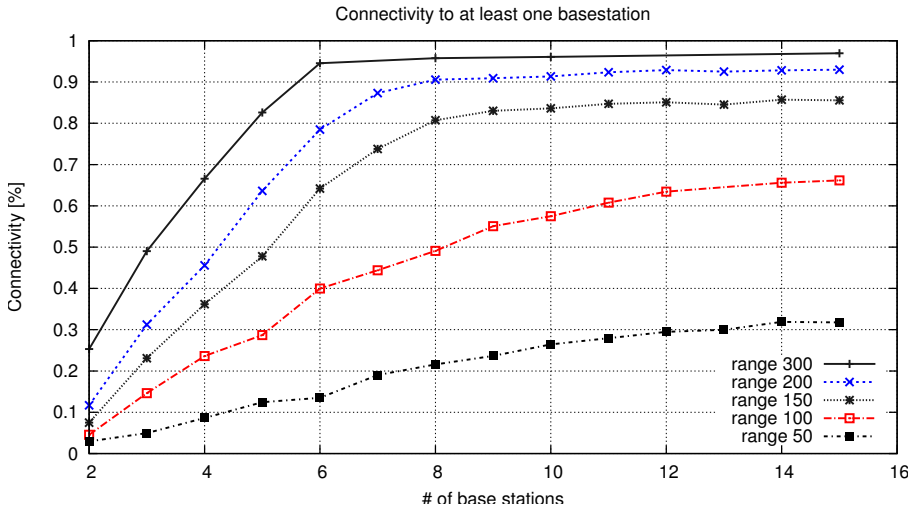


Figure 7.6.: Connectivity to a base station with varying transmission range (500 nodes/runners)

The next step was to analyze how the device density develops in such a mobile scenario. When assuming a fixed upper bound for the communication range of 150m, the size of the one-hop-neighborhood directly translates to a density measure of the network. Figure 7.7 charts the results of this analysis. The boxes show the number of neighbors interval for 50% (25%–75%) of the nodes. A surprising fact is that throughout the entire race, isolated small clusters of runners with zero or one neighbor exist. For these runners, the danger of a decayed overall connectivity prevails. Fortunately, the overall number of these isolated clusters is low, as the size of the boxes indicates.

From the above-mentioned results, a minimum communication range of 150m was chosen. This means that on average, the pacemates do not have a multi-hop connection to a base station for about 20% of the time. Hence, they must bridge this time gap by storing data for later transmission. As a result, the memory requirements were dimensioned accordingly. For further details please refer to a detailed discussion presented in [129].

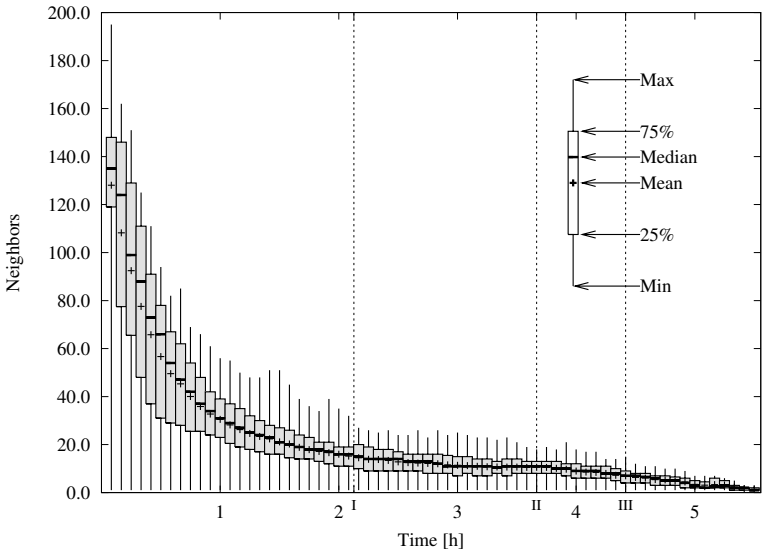


Figure 7.7.: Neighborhood size with a transmission range of 150m (500 nodes/runners)

Pacemate API As mentioned above, developing the software in parallel to the hardware was mandatory to ensure a continuous progress of the project. After the technical specifications were finished, the major software- and hardware-components and their features were known. Figure 7.8 shows the final pacemate device along with these components.

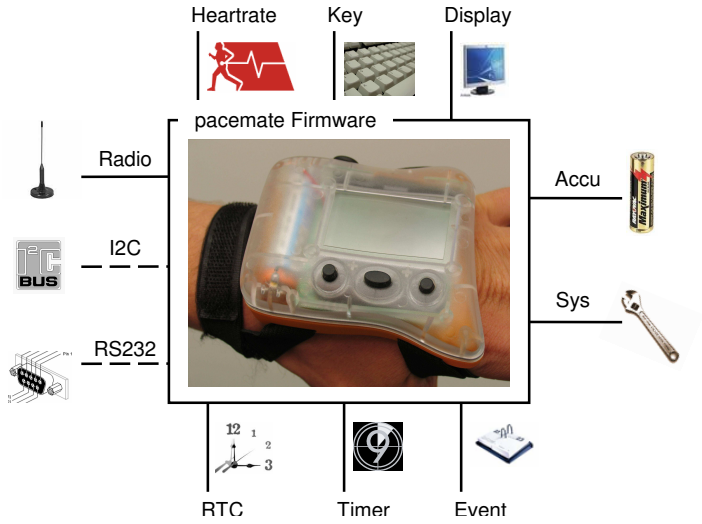


Figure 7.8.: pacemate device and schematic overview of the pacemate-firmware

On this basis, the software API provided by the pacemates was defined. To reduce the required effort when porting the simulation code to the pacemate hardware, the goal was to run the same code inside Shawn and on the pace-

mates. As shown on Figure 7.9, the applications and protocols should not be aware of the underlying implementation of the API.

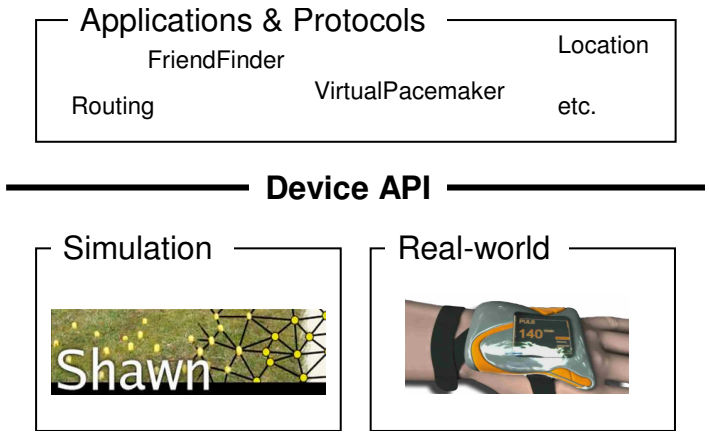


Figure 7.9.: Common pacemate API for simulation and real-world hardware deployment

Consequently, the functionality was implemented in Shawn to provide exactly the same API in the simulator as on the future hardware. Due to the previously realized simulation of the runners' movements, implementing the software API complemented the existing infrastructure to a coherent basis for the simulation of pacemate applications using Shawn.

7.1.2. SpyGlass

A central element of MarathonNet events is to provide additional information to the spectators. Similar to the media coverage of sports events in television, the visualization of available data should display interesting activities such as the position of the first runner and his immediate persecutors, passing maneuvers, etc. In contrast to traditional media coverage, the goal was to select the important scenes automatically without requiring manual intervention. To realize this goal, additional plug-ins for SpyGlass (cp. Chapter 5) were implemented. For the basic visualization of the race, the following data is required:

- Position of each runner augmented with time and position
- Current heart rates of all runners
- Polygon representation of the track

The first two data entities are regularly received at the gateways and are thus easily available. However, the pacemates are not aware of their embedding in the 3D-space but calculate only a 1D-position on the Marathon track. For this reason, a virtual base station converted the track data shown in Figure 7.2 to a data packet that was then transmitted to SpyGlass. A special Node Positioner plug-in converts the received 1D-positions to 3D-positions using the track data

received beforehand.

With this data available, the remaining tasks focus on the actual visualization and the selection of interesting scenes. As described in Section 5.2.3, the plug-ins do not directly draw on the canvas but transform the received payload to abstract drawing objects such as lines, rectangles or – in this case – runners and a track. These abstract objects are then actually drawn by different canvas implementations. Implementing the visualization therefore required four steps:

- Creating abstract drawing objects of a runner and a marathon track including their properties such as current position, rank and heart rate as well as polygon points for the marathon track.
- Implementation of plug-ins: a Background Painter for the marathon track and a Node Painter for the runners that convert the received payload to abstract drawing objects.
- Drawing instructions for each canvas to convert the abstract drawing objects to actual on-screen representations.
- Discovering interesting scenes to zoom and pan the camera. Hereby, a Director plug-in monitors the incoming data and selects a random scene from a set of predefined scenes.

Figure 7.10 depicts such an exemplary scene showing selected runners on a virtual replica of the Hamburg Marathon track.

Figure 7.10.: Automatically selected and rendered scene from the visualization of a marathon event using the SpyGlass visualization framework

7.1.3. Fabric

The architecture of MarathonNet exactly reflects that of a typical WSN. It comprises sensor nodes, gateways and backend systems, which requires the processing of data at several locations. In our case, each of these components was implemented by a different team of developers. As discussed in Section 6.2, this requires a tight synchronization of the teams to avoid differences in the implementations of the networking code. Consequently, *Fabric* was used to generate custom-tailored middleware instances for the different, heterogeneous devices from a single data type definition. As a result, changes to the data type definitions immediately reflect themselves in newly generated middleware instances.

As discussed in Chapter 6, required data structures are represented as annotated data types in XML Schema serving as the input for *Fabric*. Figure 7.11 shows an early version of this document that covers only very basic functionality. This contains a single top-level element called `RunnerData` that is comprised of a sequence of four other local element definitions (`id`, `time`, `location` and `heartrate`). Hereby, the local type definitions precisely describe the range of the individual elements (e.g., `location` ranges from 0 to 42195, the length of a marathon in meters) thus allowing for an optimized synthesis of middleware code (cp. Section 6.5).

Besides the data type definitions, the element `RunnerData` also contains annotations to select and parameterize the modules that synthesize the final middleware. As discussed in Section 6.4.1, the annotations are embedded into the element's definition as an XML document. In this case, only one annotation (`compact`) of one domain (`serialize`) is present. As described in Section 6.4.1, the *top-level elements* of an XML Schema document are mapped to network messages in the generated firmware. Hence, this results in a middleware that uses *microFibre* for data type (de-)serialization of the single data type `RunnerData`.

When invoked with different target specifications (i.e., a list of *natures*) for the sensor nodes, the gateways and the backend, *Fabric* synthesizes a family of compatible middleware instances. Because the annotated data type definition remain the same for the different target specifications, the generated middleware instances share a common knowledge on how to transform the in-memory data structures to network messages and vice versa. This enables all components to understand the payload of network messages independent of their origin. Data originating from the pacemates can therefore be processed seamlessly on the base stations, the backend database or by the SpyGlass visualization framework.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://www.marathonnet.de/v1"
        xmlns:fabric="http://www.coalesenses.com/fabric/v2"
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
        elementFormDefault="qualified" attributeFormDefault="unqualified">
 <xs:element name="RunnerData">
  <xs:annotation>
   <xs:appinfo>
    <fabric:fabric>
     <fabric:Domain name="serialize">
      <fabric:Aspect name="compact" />
     </fabric:Domain>
    </fabric:fabric>
   </xs:appinfo>
  </xs:annotation>
  <xs:complexType>
   <xs:sequence>
    <xs:element name="id">
     <xs:simpleType>
      <xs:restriction base="xs:unsignedInt">
       <xs:minInclusive value="0"/>
       <xs:maxInclusive value="50000"/>
      </xs:restriction>
     </xs:simpleType>
    </xs:element>
    <xs:element name="race_secs">
     <xs:simpleType>
      <xs:restriction base="xs:unsignedInt">
       <xs:minInclusive value="0"/>
       <xs:maxInclusive value="25000"/>
      </xs:restriction>
     </xs:simpleType>
    </xs:element>
    <xs:element name="location">
     <xs:simpleType>
      <xs:restriction base="xs:unsignedInt">
       <xs:minInclusive value="0"/>
       <xs:maxInclusive value="42195"/>
      </xs:restriction>
     </xs:simpleType>
    </xs:element>
    <xs:element name="heartrate">
     <xs:simpleType>
      <xs:restriction base="xs:unsignedInt">
       <xs:minInclusive value="30"/>
       <xs:maxInclusive value="250"/>
      </xs:restriction>
     </xs:simpleType>
    </xs:element>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

Figure 7.11.: Annotated MarathonNet data structures

7.2. Evaluation and Summary

After the conceptual presentation of our development framework for WSN applications in Chapter 3 and the detailed description of its major components Shawn, SpyGlass, *Fabric* and *microFibre*, the section above introduced its application in a real-world setting. This section evaluates this approach to WSN software development based on the criteria that were introduced in Chapter 3. To recall, these are:

- Scale from resource-constraint WSN devices to more powerful gateways and high-performance backend systems
- Integrate simulation tools and visualization environments
- Encourage application evolution over its lifetime
- Support the integration with traditional networks

- Offering support for all classes of WSN applications

The following paragraphs discuss for each criterion, how the approach presented in this work fulfills the criterion and illustrates how this is realized and how users benefit from the proposed framework.

Platform and Language Independence *Fabric* intrinsically supports middleware code synthesis for heterogeneous platforms by its generic design. The set of possible target hardware depends on the available modules and the supported programming languages. As mentioned in Chapter 6, *Fabric* currently supports the most common programming languages in WSNs such as C/C++, nesC and Java. The design of the Workspace that holds the generated source code is generic and supporting additional programming languages is straightforward.

The same situation holds for the supported hardware platforms where modules for a number of targets are already implemented. This includes the majority of gateways and backend systems because these devices are typically able to run compile code using Java or standard C/C++. Furthermore, we have successfully tested and run the generated middleware code on the pacemate, iSense and Scatternode sensor nodes. Since the programming techniques between the most hardware platforms differ only marginally, the currently generated code may already be suitable for other platforms and integrating the changes to them is straightforward.

Developers can therefore automatically generate middleware code for a heterogeneous WSN deployment that comprises sensor nodes, gateways and backend systems from a single annotated data type definition by invoking *Fabric* multiple times with different target specifications.

Integration with Simulation and Visualization Usually, application developers first use one of the simulation frameworks presented in Chapter 4 for validating the correct behavior of the application and for benchmarking its performance. Next, the implementation for the simulation framework is ported to the targeted hardware platform. From *Fabric*'s point of view, a simulation tool is simply another target platform that framework developers can support by implementing corresponding modules.

The same situation applies to visualization environments. Simulation environments typically visualize data contained in the payload of network messages that are received at gateways. As a result, a common task is the de-serialization of the received payload to in-memory data structures, which are then used to visualize the contained data. Using the SpyGlass visualization environment, adding a new visualization boils down to the mapping of in-memory data structures to abstract drawing instructions. These are then drawn on selected output devices such as a computer screen, a Postscript file or a movie file.

Application Evolution Our development framework supports all steps from an initial idea for a WSN application to the final implementation and visualization. Prior to deployments on real hardware, Shawn is used to evolve an idea gradually to a fully distributed protocol. Following, *Fabric* alleviates the step of implementing the application on the heterogeneous WSN hardware. *Fabric* alleviates this task because developers only change the data types and their annotations using a single document. The changes are automatically reflected in newly generated middleware code for each target hardware platform. On the one hand, this reduces the need to synchronize distinct implementations. On the other hand, this approach supports the willingness of developers to add new features or to fix bugs because the risk of integrating new errors at the networking level is considerably reduced.

Interconnection with Traditional Networks Since *Fabric* generates middleware code for sensor nodes as well as for gateways and backend systems, each component can (de-)serialize the application's data types from/to payload. Gateways that are connected to the sensor network and a traditional network using wireless LAN, UMTS or GPRS can therefore either forward the payload as received from the sensor nodes or perform additional services based on the contents of the payload. In the first case, all processing is performed by the backend system. In the second case, some processing is performed on the gateways. However, the choice on where to perform this processing does not increase the required implementation effort, since *Fabric* provides the necessary abstractions on every device involved in a WSN deployment.

Support for all Kinds of WSN Applications A number of code synthesis tools target a specific application domain (cp. Section 6.3) and therefore inherently support only a subset of the possible applications for WSNs. By contrast, the approach presented here offers a generic approach. The generated middleware code supports the user in the process of application development and does not generate the complete application from some kind of model.

Closing Remarks We have introduced a novel development framework that integrates simulation, visualization and application development by generating custom middleware instances for heterogeneous devices. In addition, we presented two new approaches, one in the field of simulation (Shawn) and one in the area of visualization (SpyGlass).

We strongly believe that model based development is beneficial and accelerates WSN development. Hence, the approach presented here uses a hybrid strategy where the user specifies the application's data types and their desired treatment by the generated code by using a single model. However, the user still implements the application logic manually. We believe that this is necessary for two reasons: The strict resource constraints of sensor nodes demand for a manual optimization by the user and the generation of the complete application from

models restricts the set of possible applications.

Shawn allows for a fast and eased optimization of the application prior to real world deployments by using simulations. The ability to inspect the underlying communication graph and other properties of the sensor network enables the development team to gain an in-depth knowledge of the overall scenario and provides hints on how to optimize the individual components of the application. This reduces the amount of tests with the deployed sensor network application. SpyGlass provides a platform for the visualization of the deployed and simulated sensor networks. Its flexible plug-in and drawing architecture allows developers and operators of the WSNs an easy adaptation of the visualization to their demands. While developers typically enable plug-ins that display a number of technical parameters, operators can choose plug-ins that visualize only plug-ins to support their actual task. Finally, *Fabric* is the key element for a systematic evolution of the application, for mastering the inherent device heterogeneity of the scenario and for relieving developers from recurring and error-prone tasks during the development of the application.

8. Conclusion and Future Work

Wireless sensor networks are an active area of research and currently, they are at the verge of commercial success. However, application development for WSNs remains complex as it unites the challenges of distributed applications and embedded programming. In addition, severe resource-constraints, heterogeneity, unpredictable environmental influences and the size of the networks further complicate this situation. Apart from these challenges, another issue complicates the development process. Since project specifications are subject to modification and because applications evolve over time, changes are an inherent companion of the development process. However, a common rule in project management is that changes are expensive, time consuming and error-prone the later they are introduced in the project.

Alleviating this situation requires frameworks that shield developers from these issues. However, until today, no widespread availability and use of such tools for WSNs can be observed. On the contrary, handcrafting applications from scratch is still by far the predominant approach to WSN development. This lack of powerful and easy-to-use development frameworks chokes research progress as well as a widespread industrial adoption. To improve this situation, we presented a novel development framework for WSNs that eases the implementation of WSN applications. The motivation for this work is based on the observation that WSN developers typically require simulations and visualizations in addition to the implementation of the application on sensor nodes, gateways and backend systems.

Consequently, our approach integrates simulation, visualization and heterogeneous application development into a coherent framework. This framework is comprised of the four components

- *Shawn* (a high-level and high-performance simulation tool),
- *SpyGlass* (a generic visualization environment),
- *Fabric* (a middleware synthesis framework) and
- *microFibre* (a scheme for bit-length optimized payload of network messages).

Shawn is a novel simulation tool for the design and optimization of applications prior to their real-world deployment that emerged from an algorithmic background where the design of high-level protocols and algorithms for large WSNs are the primary research goals. Starting from an initial idea for an application, Shawn encourages a multi-stage approach to simulation, which supports developers in evolving this idea to a fully distributed protocol. The central idea of

Shawn is to replace low-level effects with abstract and exchangeable models so that simulations can be used for huge networks in reasonable time while keeping the focus on the actual research problem.

We provide measurements showing that Shawn excels in its area of expertise where it outperforms existing simulation tools by orders of magnitude in runtime and resource consumption. In our evaluations, Shawn required a few seconds to simulate a network of 1,000 nodes where Ns-2 already required more than one day. Hence, developers must carefully select the simulation tool depending on the application area. When detailed simulations of issues such as radio propagation properties or low-layer issues should be considered, Shawn is obviously not the perfect choice. However, when developing algorithms and high-level protocols for WSNs, this level of detail often limits the expressiveness of simulations and blurs the view on the actual research problem. This is where Shawn provides the required abstractions and performance.

SpyGlass provides a visualization environment for wireless sensor networks. The payload of received network messages is converted to abstract drawing instructions by so-called *plug-ins*. These are visualized by different *canvas* implementations. This allows the presentation of data using different output formats including 2D- and 3D-representations on computer screens, images or movies. This modular architecture allows users an arbitrary visualization of the network's state and the outcome of simulations. Furthermore, SpyGlass enables the user to feed data back into the network to influence its state. Because of this generic approach, SpyGlass is a flexible and comprehensive toolkit for the visualization and control of WSNs.

Fabric links simulation, visualization and heterogeneous WSN devices by generating application- and data type-specific middleware for these heterogeneous target platforms. Instead of implementing networking code manually for different target platforms, developers provide a data type definition augmented with annotations. Based on this input, multiple *modules* conjointly generate optimized middleware instances. The generated middleware enables a seamless exchange of messages independent from their origin. This allows all WSN components to operate on received data and to emit data using accustomed programming language constructs while networking issues are handled by the generated middleware. The outstanding feature of the proposed framework is that it creates lean, optimized middleware instances for resource-constraint devices and that the treatment of data types is specified individually for each type.

We have demonstrated this by implementing *microFibre*, our novel serialization scheme, as a module for *Fabric*. Measurements show that *microFibre* provides a unique combination of encoding quality, small footprint and adequate execution overhead, which is vital for resource-constraint sensor nodes. Hereby, the achieved compression ratio competes with state-of-the-art techniques such as ASN.1's PER and Xenia. Compared to a manual implementation of the TinyDB data base, *microFibre* yields on average 23% (min. 14%, max. 43%)

shorter payload lengths. This comes at the price of 20% increased footprint and a runtime increased by the factor of four. However, since the wireless interface consumes considerably more energy than local computations, this is a sustainable tradeoff. In the case of the MarathonNet application, the length of the payload encoded by *microFibre* is on average 68% (min. 53%, max. 75%) shorter than the native implementation while even requiring 95% less CPU time. This clearly shows the advantage of automatically generated networking code: The synthesized middleware can perform optimizations and provide services automatically that represent an optional step in manual implementations. In this case, considerable bandwidth savings are achieved without requiring manual optimizations.

Apart from integrated optimizations, *Fabric* has the potential to increase the *reliability* of WSN applications since the generated code can optionally perform self-validations. Furthermore, *Fabric* can *speed up the development process* of WSN applications. If the generated code was implemented manually, a considerable amount of time is required to produce industry-grade code for the different target platforms. Already for the single data type of TinyDB, the COCOMO model gives an estimate of 1.62 person-months for one target platform while the design of this data type in XML Schema and the generation of a middleware using *Fabric* required less than half an hour.

The presented development framework is currently in active development and is used in both academia and industry. To name only a few, it is used for the development of applications, algorithms and protocols in the SWARMS [50], SwarmNet [47], AutoNomos [46], MarathonNet [22] and the EU-funded FRONTS [155] project as well as by the coalesenses GmbH. Furthermore, it served as the basis for more than 25 research publications, over 20 bachelor and master theses and several lectures.

Future work will concentrate on an extension of this development framework and an improvement of its individual components. For Shawn, a crucial point will be to provide more model implementations. Our current plans are to supply more mobility models and additional communication and transmission models. We strongly encourage the Open Source community to participate in this process and to enhance Shawn by contributing to its growth. We have therefore released Shawn into the public domain and it is now hosted at SourceForge.net, currently world's largest Open Source software development web site. We also plan to publish the source code of SpyGlass in the near future on SourceForge.net. In addition, we are currently working on a variety of enhancements for *Fabric*. This includes changes to *Fabric*'s core, new features for the Eclipse plug-in and additional modules. While *Fabric* currently focuses on relieving the application developer from dealing with networking aspects, we are convinced that development support should be extended to other areas like storage, aggregation, sensor and actor control, etc. We believe that *Fabric*'s concept also proves to be functional for such extensions. Therefore, it might be helpful to relax the requirements that domains must be sorted and have to interact with the neighboring ones. Concerning Fabric's Eclipse plug-in, we will include a

custom graphical editor for the annotations and we will provide an update site such that users can use Eclipse's update mechanism to install newer versions of *Fabric* automatically. Current work on additional modules includes security, packetization and forward error correction modules that will be finished shortly. Future work will focus on an even tighter integration with traditional networks where we plan to investigate how Web Services can be implemented directly on resource-constraint sensor nodes using *Fabric*.

A. Source Code Listings

A.1. Shawn Configuration Files

A.1.1. Plain Text

```
# Construct an empty world using the "list" edge model,
# the communication model "disk_graph" (nodes can
# communicate iff they are within range "range", which is set to 10)

prepare_world edge_model=list comm_model=disk_graph range=10

# Add 800 nodes in a 25x25-sized box.
# Each node gets one processor, namely "helloworld"

rect_world width=25 height=25 count=800 processors=helloworld

# Run the simulation until all nodes are inactive or 10 time units
# have elapsed.

simulation max_iterations=10
```

A.1.2. Java-language Scripting

```
HashMap worldConfig = new HashMap();
worldConfig.put("edge_model", "list");
worldConfig.put("comm_model", "disk_graph");
worldConfig.put("range", "10");

int w = 25, h = 25;
String processors = "helloworld";

shawn.runCommand("prepare_world", worldConfig);

shawn.runCommand("rect_world" "width=" + w +
    " height=" + h +
    " count="+ count +
    " processors=" + processors);

shawn.runCommand("simulation", "max_iterations=10");
```

A.2. TinyDB Data type

A.2.1. Excerpts from the TinyDB Code

```
/*
 * "Copyright (c) 2000-2003 The Regents of the University  of California.
 * All rights reserved.
 *
 * Permission to use, copy, modify, and distribute this software and its
 * documentation for any purpose, without fee, and without written
 * agreement is hereby granted, provided that the above copyright notice,
 * the following two paragraphs and the author appear in all copies of
 * this software.
 *
 * IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR
 * DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING
 * OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF THE
 * UNIVERSITY OF CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES,
 * INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY
 * AND FITNESS FOR A PARTICULAR PURPOSE.  THE SOFTWARE PROVIDED HEREUNDER IS
```

```
20  * ON AN "AS IS" BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATION TO
21  * PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS."
22  *
23  * Copyright (c) 2002-2003 Intel Corporation
24  * All rights reserved.
25  *
26  * This file is distributed under the terms in the attached INTEL-LICENSE
27  * file. If you do not find these files, copies can be found by writing to
28  * Intel Research Berkeley, 2150 Shattuck Avenue, Suite 1300, Berkeley, CA,
29  * 94704.  Attention:  Intel License Inquiry.
30  */
31
32 enum {QUERY_FIELD_SIZE = 8,
33       COMMAND_SIZE = 8,
34       STRING_SIZE = 8};
35
36 /* Fields are just an 8 character name,
37    plus a transformation operator? */
38 typedef struct {
39   char name[QUERY_FIELD_SIZE]; //8
40   uint8_t op; //9
41   uint8_t type;//10
42   //alias info here?
43 } Field, *FieldPtr;
44
45
46 /* We can (optionally) invoke a command in response to a query.
47    The command may include a single, short parameter.
48 */
49 typedef struct {
50   char name[COMMAND_SIZE]; //8
51   bool hasParam;  //9
52   short param; //11
53 } CmdBufInfo;
54
55 typedef struct {
56   bool hasOutput:1;
57   bool hasInput:1;
58   bool create:1;
59   uint16_t numRows:13; //2
60   char outBufName[STRING_SIZE];  //10
61   char inBufName[STRING_SIZE]; //18
62   BufferPolicy policy; //19
63 } RamBufInfo;
64
65 typedef union {
66   CmdBufInfo cmd; //11
67   RamBufInfo ram; //19
68 } BufInfo;
69
70 typedef struct {
71   uint16_t group;
72   //  uint8_t len;
73   char *data;
74 } AggResultRef;
75
76 typedef char Op;
77
78 enum {
79   EQ = 0,
80   NEQ = 1,
81   GT = 2,
82   GE = 3,
83   LT = 4,
84   LE = 5
85 };
86
87 typedef char Agg;
88
89 //field operators, for use in exprs
90 enum {
91   FOP_NOOP = 0,
92   FOP_TIMES = 1,
93   FOP_DIVIDE = 2,
94   FOP_ADD = 3,
95   FOP_SUBTRACT = 4,
96   FOP_MOD = 5,
97   FOP_RSHIFT = 6
98 };
99
100 //expressions are either aggregates or selections
101 //for now we support the simplest imagineable types (e.g.
102 //no nested expressions, joins, or modifiers on fields)
103 typedef struct {
104   short field;  //2
105   Op op; //3
106   short value; //5
107 } OpValExpr;
108
109 typedef struct {
```

```
110     short field;  //2
111     short groupingField;  //field to group on //4
112     short groupFieldOp; //6
113     short groupFieldConst; //8
114
115     Agg op; //9
116   } AggregateExpression;
117
118   typedef struct {
119     AggregateExpression agg; //9
120     //temporal agg can have at most 4 arguments
121     uint8_t args[4];//13
122   } TemporalAggExpr;
123
124   typedef struct {
125     Op op; //1
126     short field; //3
127     char s[STRING_SIZE]; //11
128   } StringExpr; //11
129
130   enum {
131     kNO_GROUPING_FIELD = 0xFFFF
132   };
133
134   //operator state represents the per operator
135   //query state stored in the tuple router and
136   //sent to the operators on invocation
137   typedef char** OperatorStateHandle;
138
139   enum {
140     kSEL = 0,
141     kAGG = 1,
142     kTEMP_AGG = 2,
143   };
144
145   typedef struct {
146     char opType:6;
147     bool isStringExp:1; //is this a string expression or not?
148     bool success:1; //boolean indicating if this
149                     //query was successfully applied //1
150     char idx; //index of this expression in the query //2
151
152     union {
153       OpValExpr opval;
154       AggregateExpression agg;
155       TemporalAggExpr tagg;
156       StringExpr sexp; //for comparisons with strings
157     } ex; //15
158     short fieldOp; //17 -- from FOP... defines above
159     short fieldConst; //19
160
161     OperatorStateHandle opState; //21
162   } Expr, *ExprPtr;
163
164
165   enum {
166     kFIRST_RESULT = 0xFF,
167   };
168
169   //enums for the QueryResult qrType (what kind of query result)
170   enum {
171     kUNDEFINED = 0,
172     kIS_AGG = 1,
173     kNOT_AGG = 2,
174     kAGG_SINGLE_FIELD = 3
175   };
176
177   /** Message type for carrying query messages */
178   typedef struct QueryMessage {
179     // XXX recompute header size //7
180     uint8_t qid; //query id //8 -- note that this byte must be qid
181     uint16_t fwdNode; //10 -- node that forwarded the query message
182     char msgType;  //type of message (e.g. add, modify, delete q) //11
183     char numFields; //12
184     char numExprs; //13
185     char fromBuffer; //14
186     uint8_t fromCatalogBuffer:1; //15
187     uint8_t hasEvent:1; //15
188     uint8_t hasForClause:1; //15
189     uint8_t bufferType:5; //15 -- output buffer type
190     short epochDuration; //in millisecs -- 17
191     char type;  //is this a field, expression, buffer, or event msg -- 18
192     char idx; //19
193     uint8_t timeSyncData[5];
194     int16_t clockCount;
195     union {
196       Field field;
197       Expr expr; //40
198       BufInfo buf;
199       char eventName[COMMAND_SIZE];
```

```
200     short numEpochs;
201     int8_t ttl;  //for delete msg
202   } u; //40
203
204 } QueryMessage, *QueryMessagePtr;
```

A.2.2. XML Schema Representation

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <xs:schema targetNamespace="http://www.tinydb.net/fabrictest"
4             xmlns:tinydb="http://www.tinydb.net/fabrictest"
5       xmlns:xs="http://www.w3.org/2001/XMLSchema"
6       xmlns:fabric="http://www.coalesenses.com/fabric/v2"
7       elementFormDefault="qualified"
8       attributeFormDefault="unqualified">
9
10  <xs:element name="query" type="tinydb:QueryMessage" />
11
12  <xs:complexType name="OpValExpr">
13   <xs:all>
14    <xs:element name="field" type="xs:short"/>
15    <xs:element name="op" type="xs:unsignedByte"/>
16    <xs:element name="value" type="xs:short"/>
17   </xs:all>
18  </xs:complexType>
19  <xs:complexType name="AggregateExpression">
20   <xs:all>
21    <xs:element name="field" type="xs:short"/>
22    <xs:element name="groupingField" type="xs:short"/>
23    <xs:element name="groupFieldOp" type="xs:short"/>
24    <xs:element name="groupFieldConst" type="xs:short"/>
25    <xs:element name="op" type="xs:unsignedByte"/>
26   </xs:all>
27  </xs:complexType>
28  <xs:complexType name="TemporalAggExpr">
29   <xs:sequence>
30    <xs:element name="agg" type="tinydb:AggregateExpression"/>
31    <xs:element name="args" type="xs:unsignedByte" maxOccurs="4"/>
32   </xs:sequence>
33  </xs:complexType>
34  <xs:complexType name="StringExpr">
35   <xs:all>
36    <xs:element name="op" type="xs:unsignedByte"/>
37    <xs:element name="field" type="xs:short"/>
38    <xs:element name="s">
39     <xs:simpleType>
40      <xs:restriction base="xs:string">
41       <xs:minLength value="0"/>
42       <xs:maxLength value="8"/>
43      </xs:restriction>
44     </xs:simpleType>
45    </xs:element>
46   </xs:all>
47  </xs:complexType>
48  <xs:complexType name="Field">
49   <xs:all>
50    <xs:element name="name">
51     <xs:simpleType>
52      <xs:restriction base="xs:string">
53       <xs:minLength value="0"/>
54       <xs:maxLength value="8"/>
55      </xs:restriction>
56     </xs:simpleType>
57    </xs:element>
58    <xs:element name="op" type="xs:unsignedByte"/>
59    <xs:element name="type" type="xs:unsignedByte"/>
60   </xs:all>
61  </xs:complexType>
62  <xs:complexType name="Expr">
63   <xs:sequence>
64    <xs:element name="opType">
65     <xs:simpleType>
66      <xs:restriction base="xs:unsignedByte">
67       <xs:minInclusive value="0"/>
68       <xs:maxInclusive value="63"/>
69      </xs:restriction>
70     </xs:simpleType>
71    </xs:element>
72    <xs:element name="isStringExp" type="xs:boolean"/>
73    <xs:element name="success" type="xs:boolean"/>
74    <xs:element name="idx" type="xs:unsignedByte"/>
75    <xs:element name="content">
76     <xs:complexType>
77      <xs:choice>
78       <xs:element name="opval" type="tinydb:OpValExpr"/>
79       <xs:element name="agg" type="tinydb:AggregateExpression"/>
```

```
80          <xs:element name="tagg" type="tinydb:TemporalAggExpr"/>
81          <xs:element name="sexp" type="tinydb:StringExpr"/>
82         </xs:choice>
83        </xs:complexType>
84       </xs:element>
85       <xs:element name="fieldOp" type="xs:short"/>
86       <xs:element name="fieldConst" type="xs:short"/>
87      </xs:sequence>
88     </xs:complexType>
89     <xs:complexType name="CmdBufInfo">
90      <xs:all>
91       <xs:element name="name">
92        <xs:simpleType>
93         <xs:restriction base="xs:string">
94          <xs:minLength value="0"/>
95          <xs:maxLength value="8"/>
96         </xs:restriction>
97        </xs:simpleType>
98       </xs:element>
99       <xs:element name="hasParam" type="xs:boolean"/>
100      <xs:element name="param" type="xs:short" minOccurs="0"/>
101     </xs:all>
102    </xs:complexType>
103    <xs:complexType name="RamBufInfo">
104     <xs:all>
105      <xs:element name="hasOutput" type="xs:boolean"/>
106      <xs:element name="hasInput" type="xs:boolean"/>
107      <xs:element name="create" type="xs:boolean"/>
108      <xs:element name="numRows">
109       <xs:simpleType>
110        <xs:restriction base="xs:unsignedShort">
111         <xs:minInclusive value="0"/>
112         <xs:maxInclusive value="8191"/>
113        </xs:restriction>
114       </xs:simpleType>
115      </xs:element>
116      <xs:element name="outBufName">
117       <xs:simpleType>
118        <xs:restriction base="xs:string">
119         <xs:minLength value="0"/>
120         <xs:maxLength value="8"/>
121        </xs:restriction>
122       </xs:simpleType>
123      </xs:element>
124      <xs:element name="inBufName">
125       <xs:simpleType>
126        <xs:restriction base="xs:string">
127         <xs:minLength value="0"/>
128         <xs:maxLength value="8"/>
129        </xs:restriction>
130       </xs:simpleType>
131      </xs:element>
132      <xs:element name="policy" type="xs:unsignedByte"/>
133     </xs:all>
134    </xs:complexType>
135    <xs:complexType name="BufInfo">
136     <xs:all>
137      <xs:element name="cmd" type="tinydb:CmdBufInfo"/>
138      <xs:element name="ram" type="tinydb:RamBufInfo"/>
139     </xs:all>
140    </xs:complexType>
141    <xs:complexType name="QueryMessage">
142     <xs:sequence>
143      <xs:element name="qid" type="xs:unsignedByte"/>
144      <xs:element name="fwdNode" type="xs:unsignedShort"/>
145      <xs:element name="msgType" type="xs:unsignedByte"/>
146      <xs:element name="numFields" type="xs:unsignedByte"/>
147      <xs:element name="numExprs" type="xs:unsignedByte"/>
148      <xs:element name="fromBuffer" type="xs:unsignedByte"/>
149      <xs:element name="fromCatalogBuffer" type="xs:boolean"/>
150      <xs:element name="hasEvent" type="xs:boolean"/>
151      <xs:element name="hasForClause" type="xs:boolean"/>
152      <xs:element name="bufferType">
153       <xs:simpleType>
154        <xs:restriction base="xs:unsignedByte">
155         <xs:minInclusive value="0"/>
156         <xs:maxInclusive value="31"/>
157        </xs:restriction>
158       </xs:simpleType>
159      </xs:element>
160      <xs:element name="epochDuration" type="xs:short"/>
161      <xs:element name="type" type="xs:unsignedByte"/>
162      <xs:element name="idx" type="xs:unsignedByte"/>
163      <xs:element name="timeSyncData" type="xs:unsignedByte"
164                   minOccurs="0" maxOccurs="5"/>
165      <xs:element name="clockCount" type="xs:short"/>
166      <xs:element name="content">
167       <xs:complexType>
168        <xs:choice>
169         <xs:element name="field" type="tinydb:Field"/>
```

```
170         <xs:element name="expr" type="tinydb:Expr"/>
171         <xs:element name="buf" type="tinydb:BufInfo"/>
172         <xs:element name="eventName">
173          <xs:simpleType>
174           <xs:restriction base="xs:string">
175            <xs:minLength value="0"/>
176            <xs:maxLength value="8"/>
177           </xs:restriction>
178          </xs:simpleType>
179         </xs:element>
180         <xs:element name="numEpochs" type="xs:short"/>
181         <xs:element name="ttl" type="xs:byte"/>
182        </xs:choice>
183       </xs:complexType>
184     </xs:element>
185    </xs:sequence>
186  </xs:complexType>
187 </xs:schema>
```

A.2.3. ASN.1. Representation

```
1  Fabrictest
2  DEFINITIONS AUTOMATIC TAGS ::=
3  BEGIN
4
5  XMLCompatibleString ::= UTF8String (FROM (
6      {0, 0, 0, 9} | {0, 0, 0, 10} | {0, 0, 0, 13} |
7      {0, 0, 0, 32}   .. {0, 0, 215, 255} |
8      {0, 0, 224, 0} .. {0, 0, 255, 253} |
9      {0, 1, 0, 0}    .. {0, 16, 255, 255}))
10
11 Short ::= INTEGER (-32768..32767)
12
13 String ::= XMLCompatibleString
14
15 UnsignedShort ::= INTEGER (0..65535)
16
17 Query ::= QueryMessage
18
19 OpValExpr ::= SEQUENCE {
20     order SEQUENCE OF ENUMERATED {
21  field,
22  op,
23  value
24     },
25     field Short,
26     op    INTEGER (0..255),
27     value Short
28 } (CONSTRAINED BY
29     {/* Shall conform to ITU-T Rec. X.693 | ISO/IEC 8825-4, clause 35 */})
30
31 AggregateExpression ::= SEQUENCE {
32     order           SEQUENCE OF ENUMERATED {
33  field,
34  groupingField,
35  groupFieldOp,
36  groupFieldConst,
37  op
38     },
39     field           Short,
40     groupingField   Short,
41     groupFieldOp    Short,
42     groupFieldConst Short,
43     op              INTEGER (0..255)
44 } (CONSTRAINED BY
45     {/* Shall conform to ITU-T Rec. X.693 | ISO/IEC 8825-4, clause 35 */})
46
47 TemporalAggExpr ::= SEQUENCE {
48     agg        AggregateExpression,
49     args-list SEQUENCE (SIZE(1..4)) OF args INTEGER (0..255)
50 }
51
52 StringExpr ::= SEQUENCE {
53     order SEQUENCE OF ENUMERATED {
54  op,
55  field,
56  s
57     },
58     op    INTEGER (0..255),
59     field Short,
60     s     String (SIZE(0..8))
61 } (CONSTRAINED BY
62     {/* Shall conform to ITU-T Rec. X.693 | ISO/IEC 8825-4, clause 35 */})
63
64 Field ::= SEQUENCE {
65     order SEQUENCE OF ENUMERATED {
66  name,
```

```
 op,
 type
    },
    name  String (SIZE(0..8)),
    op    INTEGER (0..255),
    type  INTEGER (0..255)
} (CONSTRAINED BY
    {/* Shall conform to ITU-T Rec. X.693 | ISO/IEC 8825-4, clause 35 */})

Expr ::= SEQUENCE {
    opType       INTEGER (0..63),
    isStringExp BOOLEAN,
    success      BOOLEAN,
    idx          INTEGER (0..255),
    content      SEQUENCE {
 choice CHOICE {
     opval OpValExpr,
     agg   AggregateExpression,
     tagg  TemporalAggExpr,
     sexp  StringExpr
 }
    },
    fieldOp      Short,
    fieldConst   Short
}

CmdBufInfo ::= SEQUENCE {
    order    SEQUENCE OF ENUMERATED {
 name,
 hasParam,
 param
    },
    name      String (SIZE(0..8)),
    hasParam BOOLEAN,
    param     Short OPTIONAL
} (CONSTRAINED BY
    {/* Shall conform to ITU-T Rec. X.693 | ISO/IEC 8825-4, clause 35 */})

RamBufInfo ::= SEQUENCE {
    order       SEQUENCE OF ENUMERATED {
 hasOutput,
 hasInput,
 create,
 numRows,
 outBufName,
 inBufName,
 policy
    },
    hasOutput  BOOLEAN,
    hasInput   BOOLEAN,
    create     BOOLEAN,
    numRows    UnsignedShort (0..8191),
    outBufName String (SIZE(0..8)),
    inBufName  String (SIZE(0..8)),
    policy     INTEGER (0..255)
} (CONSTRAINED BY
    {/* Shall conform to ITU-T Rec. X.693 | ISO/IEC 8825-4, clause 35 */})

BufInfo ::= SEQUENCE {
    order SEQUENCE OF ENUMERATED {
 cmd,
 ram
    },
    cmd   CmdBufInfo,
    ram   RamBufInfo
} (CONSTRAINED BY
    {/* Shall conform to ITU-T Rec. X.693 | ISO/IEC 8825-4, clause 35 */})

QueryMessage ::= SEQUENCE {
    qid                 INTEGER (0..255),
    fwdNode             UnsignedShort,
    msgType             INTEGER (0..255),
    numFields           INTEGER (0..255),
    numExprs            INTEGER (0..255),
    fromBuffer          INTEGER (0..255),
    fromCatalogBuffer BOOLEAN,
    hasEvent            BOOLEAN,
    hasForClause        BOOLEAN,
    bufferType          INTEGER (0..31),
    epochDuration       Short,
    type                INTEGER (0..255),
    idx                 INTEGER (0..255),
    timeSyncData-list SEQUENCE (SIZE(0..5)) OF timeSyncData INTEGER (0..255),
    clockCount          Short,
    content             SEQUENCE {
 choice CHOICE {
     field      Field,
     expr       Expr,
     buf        BufInfo,
     eventName String (SIZE(0..8)),
```

```
157         numEpochs Short,
158         ttl       INTEGER (-128..127)
159   }
160       }
161 }
162
163 END
```

Curriculum Vitae

Privat

Dennis Pfisterer	Geboren am 23.12.1976 in Heidelberg, wohnhaft in Hamburg, ledig
Eltern	Werner und Ria Pfisterer (geb. Sickmüller)

Ausbildung

1983 – 1996	Internationale Gesamtschule Heidelberg und Helmholtz-Gymnasium (Abitur mit der Note 1,9)
10/1997 bis 08/2001	Studium der Nachrichtentechnik an der Fachhochschule Mannheim mit Abschluss als Diplom-Ingenieur (FH). Benotung der Diplomarbeit 1,0 und Diplomnote 1,2
11/2001	Preis der Karl-Völker-Stiftung für eine hervorragende Diplomarbeit des Studienjahres 2001
07/2003	Erwerb der Promotionszulassung für Fachhochschulabsolventen an der Technischen Universität Braunschweig

Beruflicher Werdegang

07/1996 – 04/1997	Wehrdienst bei der NATO-Einheit LANDCENT
09/1998 – 01/1999	Praxissemester bei der BASF AG, Mannheim
07/1999 – 08/1999	Praxissemester an der Nanyang Polytechnic, Singapur
06/2001 – 02/2005	Wissenschaftlicher Mitarbeiter bei der European Media Laboratory GmbH, Heidelberg
SS 2002	Lehrbeauftragter an der Fachhochschule Mannheim für die Vorlesung “Datenverarbeitung 1” und die zugehörige Übung
11/2003 – 12/2004	Wissenschaftlicher Mitarbeiter am Institut für Betriebssysteme und Rechnerverbund der TU Braunschweig
seit 01/2005	Wissenschaftlicher Mitarbeiter am Institut für Telematik der Universität zu Lübeck
07/2005	Mitbegründer der coalesenses GmbH als Spin-off der der Universität zu Lübeck

Personal Publications

[1] Buschmann, C., Fekete, S. P., Fischer, S., Kröller, A., and Pfisterer, D. Koordinatenfreies Lokationsbewusstsein. *it - Information Technology, Themenheft Sensornetze 47*, 4 (Apr. 2005), 70–77.

[2] Buschmann, C., and Pfisterer, D. iSense: A modular hardware and software platform for wireless sensor networks. Tech. rep., 6. Fachgespräch "Drahtlose Sensornetze" der GI/ITG-Fachgruppe "Kommunikation und Verteilte Systeme", 2007.

[3] Buschmann, C., Pfisterer, D., and Fischer, S. Experimenting with computer swarms: a mobile platform based on blimps. In *Proceedings of the Second International Conference on Mobile Systems, Applications, and Services (MobiSys' 04), Boston, Massachusetts, USA* (June 2004), ACM SIGMOBILE.

[4] Buschmann, C., Pfisterer, D., and Fischer, S. From diodes to insights. In *2nd Workshop on Sensor Networks, INFORMATIK 2005* (Sept. 2005).

[5] Buschmann, C., Pfisterer, D., and Fischer, S. Kommunikationsaufwand von Verfahren zur Errichtung eines konsistenten Zeitbewusstseins - Eine quantitative Analyse. *PIK - Praxis der Informationsverarbeitung und Kommunikation, Sonderheft Wireless Sensor Networks*, 4 (Apr. 2005), 74–79.

[6] Buschmann, C., Pfisterer, D., and Fischer, S. Estimating distances using neighborhood intersection. In *Proceedings of the 11th IEEE International Conference on Emerging Technologies and Factory Automation, Prague, Czech Republic (ETFA' 06)* (Sept. 2006), pp. 314–321.

[7] Buschmann, C., Pfisterer, D., and Fischer, S. On the relation of neighborhoods and distances. Tech. rep., 5. Fachgespräch "Drahtlose Sensornetze" der GI/ITG-Fachgruppe "Kommunikation und Verteilte Systeme", 2006.

[8] Buschmann, C., Pfisterer, D., Fischer, S., Fekete, S. P., and Kröller, A. SpyGlass: Taking a closer look into sensor networks. In *Proceedings of the 2nd international conference on Embedded networked sensor system (SenSys' 04)* (Nov. 2004), ACM Press New York, NY, USA, pp. 301–302.

[9] Buschmann, C., Pfisterer, D., Fischer, S., Fekete, S. P., and Kröller, A. SpyGlass: A wireless sensor network visualizer. *ACM*

SIGBED Review 2, 1 (Jan. 2005).

[10] Ding, Y., Litz, H., Malaka, R., and Pfisterer, D. On programming information agent systems - an integrated hotel reservation service as case study. In *Proceedings of the First German Conference on Multiagent System Technologies (MATES' 03), Lecture Notes in Computer Science 2831* (Sept. 2003), Springer.

[11] Ding, Y., Litz, H., and Pfisterer, D. A graphical single-authoring framework for building multi-platform user interfaces. In *Proceedings of the 9th International Conference on Intelligent User Interfaces (IUI' 04)* (Jan. 2004), ACM Press, pp. 235–237. Funchal, Madeira, Portugal.

[12] Ding, Y., Malaka, R., and Pfisterer, D. An open framework for load balanced multi-agent systems. In *Proceedings of the Workshop on Ubiquitous Agents on Embedded, Wearable, and Mobile Devices (AAMAS'02)* (July 2002).

[13] Ding, Y., Pfisterer, D., and Walther, U. Resource-adaptive video-streaming for mobility. In *Artificial Intelligence in Mobile Systems (AIMS' 02)* (Aug. 2002), pp. 1–8.

[14] Fekete, S., Kröller, A., Pfisterer, D., and Fischer, S. Algorithmic aspects of large sensor networks. In *Proceedings of Mobility and Scalability in Wireless Sensor Networks (MSWSN' 06)* (2006), CTI Press, pp. 141–152.

[15] Fekete, S. P., Kröller, A., Dennis Pfisterer, S. F., and Buschmann, C. Neighborhood-based topology recognition in sensor networks. In *Algosensors'04, Lecture Notes in Computer Science* (July 2004), vol. 3121, Springer, pp. 123–136.

[16] Fekete, S. P., Kröller, A., Fischer, S., and Pfisterer, D. Shawn: The fast, highly customizable sensor network simulator. In *Proceedings of the Fourth International Conference on Networked Sensing Systems (INSS' 07)* (June 2007).

[17] Hellbrück, H., Lipphardt, M., Pfisterer, D., Ransom, S., and Fischer, S. Praxiserfahrungen mit MarathonNet - Ein mobiles Sensornetz im Sport. *PIK - Praxis der Informationsverarbeitung und Kommunikation 29*, 4 (2006).

[18] Kröller, A., Fekete, S. P., Pfisterer, D., and Fischer, S. Deterministic boundary recognition and topology extraction for large sensor networks. In *Proceedings of the 17th ACM-SIAM Sympos. Discrete Algorithms (SODA' 06)* (2006), pp. 1000–1009.

[19] Kröller, A., Pfisterer, D., Buschmann, C., Fekete, S. P., and Fischer, S. Shawn: A new approach to simulating wireless sensor networks. In *Design, Analysis, and Simulation of Distributed Systems 2005 (DASD' 05)* (Apr. 2005), pp. 117–124.

[20] Lipphardt, M., Hellbrück, H., Pfisterer, D., Ransom, S., and Fischer, S. Practical experiences on mobile inter-body-area-networking. In *Proceedings of the Second International Conference on Body Area Networks (BodyNets'07)* (2007).

[21] Malaka, R., Haeussler, J., Aras, H., Merdes, M., Pfisterer, D., Joest, M., and Porzel, R. *SmartKom-Mobile: Intelligent Interaction with a Mobile System*. Springer Verlag, Heidelberg, Germany, 2006, pp. 505–522.

[22] Pfisterer, D. Resource aware multi-fidelity video streaming. Master's thesis, Mannheim University of Applied Sciences, 2001.

[23] Pfisterer, D., Buschmann, C., Hellbrück, H., and Fischer, S. Data-type centric middleware synthesis for wireless sensor network application development. In *Proceedings of the Fifth Annual Mediterranean Ad Hoc Networking Workshop (Med-Hoc-Net' 06)* (2006), pp. 329–336.

[24] Pfisterer, D., Buschmann, C., Hellbrück, H., and Fischer, S. Supporting WSN application development using data type-centric middleware synthesis. Tech. rep., 5. Fachgespräch "Drahtlose Sensornetze" der GI/ITG-Fachgruppe "Kommunikation und Verteilte Systeme", 2006.

[25] Pfisterer, D., Fischer, S., Kröller, A., and Fekete, S. Shawn: Ein alternativer Ansatz zur Simulation von Sensornetzwerken. Tech. rep., 4. Fachgespräch "Drahtlose Sensornetze" der GI/ITG-Fachgruppe "Kommunikation und Verteilte Systeme", Mar. 2005.

[26] Pfisterer, D., Hellbrück, H., and Fischer, S. Fabric: Towards data type-centric middleware synthesis. In *Proceedings of the Euro-American Workshop on Middleware for Sensor Networks in conjunction with the International Conference on Distributed Computing (DCOSS' 06)* (2006).

[27] Pfisterer, D., Lipphardt, M., Buschmann, C., Hellbrück, H., Fischer, S., and Sauselin, J. H. Marathonnet: Adding value to large scale sport events - a connectivity analysis. In *Proceedings of the International Conference on Integrated Internet Ad hoc and Sensor Networks (InterSense'06)* (May 2006), A. Press, Ed., p. 12.

[28] Pfisterer, D., Wegner, M., Hellbrück, H., Werner, C., and Fischer, S. Energy-optimized data serialization for heterogeneous WSNs using middleware synthesis. In *Proceedings of The Sixth Annual Mediterranean Ad Hoc Networking Workshop (Med-Hoc-Net' 07)* (June 2007), pp. 180–187.

List of Tables

List of Figures

List of Abbreviations

API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
ASN.1	Abstract Syntax Notation One
BAN	Body Area Network
BER	Basic Encoding Rules
CH	Cluster-head
CLI	Common Language Infrastructure
COCOMO	COnstructive COst Model
CORBA	Common Request Broker Architecture
CPU	Central Processing Unit
DLE	Data Link Escape
DTD	Document Type Definition
dVSIS	Distributed Virtual Shared Information Space
EMF	Eclipse Modeling Framework
ESB	Embedded Sensor Board
ETX	End of Text
GCC	Gnu Compiler Collection
GPRS	General Packet Radio Service
GPS	Global Positioning System
HTML	HyperText Markup Language
I2C	Inter-Integrated Circuit (pronounced I-squared-C)
IC	Integrated Circuit
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
JPEG	Joint Picture Expert Group
JVM	Java Virtual Machine
LED	Light-Emitting Diode
LiIon	Lithium Ion
LiPo	Lithium Polymer
MAC	Media Access Control
MDA	Model Driven Architecture
MPEG	Motion Picture Expert Group
MSB	Modular Sensor Board

nam ..... Network Animator
NFS ..... Network File System
NiMH ..... Nickel Metal Hydride
Ns-2 ..... Network Simulator-2
OMG ..... Object Management Group
OMNeT++ ..... Objective Modular Network Testbed
OS ..... Operating System
OSI ..... Open Systems Interconnection
PC ..... Personal Computer
PDA ..... Personal Digital Assistant
PER ..... Packed Encoding Rules
PNG ..... Portable Network Graphic
Q-UDG ..... Quasi-Unit Disk Graph
QName ..... Qualified Name
QoS ..... Quality of Service
RAM ..... Random Access Memory
RFC ..... Request For Comments
RIM ..... Radio Irregularity Model
RPC ..... Remote Procedure Call
RS232 ..... Recommended Standard 232
RTC ..... Real-Time Clock
SENSE ..... Sensor Network Simulator and Emulator
SGML ..... Standard Generalized Markup Language
SMS ..... Short Message Service
SNMS ..... Sensor Network Management System
SQL ..... Structured Query Language
STAX ..... Simple Typed API for XML
STX ..... Start of Text
TICTAC ..... Traffic Induced Control of Time and Communication
TOSSIM ..... TinyOS mote simulator
UDG ..... Unit Disk Graph
UDP ..... User Datagram Protocol
UML ..... Unified Modeling Language
UMTS ..... Universal Mobile Telecommunications System
URI ..... Uniform Resource Identifier
VM ..... Virtual Machine
W3C ..... World Wide Web Consortium
WSN ..... Wireless Sensor Network
XDR ..... External Data Representation

XER ..... XML Encoding Rules
XML ..... Extensible Markup Language

Bibliography

[1] Abramson, N. The ALOHA system - another alternative for computer communications. In *Proceedings of the AFIPS Conference* (1970), vol. 37, pp. 295–298.

[2] Adjie-Winoto, W., Schwartz, E., and Balakrishnan, H. The design and implementation of an intentional naming system. In *Proceedings of the Symposium on Operating Systems Principles, Charleston, SC* (1999).

[3] Amazon.com, Inc. Amazon E-Commerce Service, Nov. 2005. http://aws.amazon.com.

[4] Baer, M. The ultimate on-the-fly network. *Wired Magazine*, 12 (2003). http://www.wired.com/wired/archive/11.12/network.html.

[5] Bakshi, A., Ou, J., and Prasanna, V. K. Towards automatic synthesis of a class of application-specific sensor networks. In *CASES'02: Proceedings of the international conference on Compilers, architecture, and synthesis for embedded systems* (2002).

[6] Bakshi, A., and Prasanna, V. K. Algorithm design and synthesis for wireless sensor networks. In *International Conference on Parallel Processing (ICPP'04)* (2004).

[7] Bakshi, A., Prasanna, V. K., Reich, J., and Larner, D. The abstract task graph: A methodology for architecture-independent programming of networked sensor systems. In *Proc. of the Workshop on End-to-end, sense-and-respond systems, applications and services* (2005).

[8] Barriere, L., Fraigniaud, P., and Narayanan, L. Robust position-based routing in wireless ad hoc networks with unstable transmission ranges. In *DIALM '01* (New York, NY, USA, 2001), ACM Press, pp. 19–27. Q-UDG radio model.

[9] Bencomo, N., Blair, G., Coulson, G., and Batista, T. Towards a meta-modelling approach to configurable middleware. In *Proc. 2nd ECOOP 05 Workshop on Reflection, AOP and Meta-Data for Software Evolution, Glasgow, Scotland* (2005).

[10] Beutel, J. Fast-prototyping using the BTnode platform. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe* (3001 Leuven, Belgium, 2006), European Design and Automation Association, pp. 977–982.

[11] Beutel, J., Dyer, M., Hinz, M., Meier, L., and Ringwald, M. Next-generation prototyping of sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems* (2004).

[12] Beutel, J., Kasten, O., and Ringwald, M. Poster abstract: BTnodes – a distributed platform for sensor nodes. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems* (New York, NY, USA, 2003), ACM Press, pp. 292–293.

[13] Blumenthal, J., and Timmermann, D. Resource-aware service architecture for mobile services in wireless sensor networks. In *Proceedings of the International Conference on Wireless and Mobile Communications (ICWMC'06)* (2006), pp. 34–39.

[14] Boehm, B. W. *Software Engineering Economics*. Prentice Hall PTR, 1981.

[15] Boehm, B. W., Abts, C., and Brown, A. W. *Software Cost Estimation with Cocomo II*. Prentice Hall International, 1990.

[16] Braginsky, D., and Estrin, D. Rumor routing algorithm for sensor networks. In *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications* (2002).

[17] Burri, N., von Rickenbach, P., Wattenhofer, R., and Weber, Y. Topology control made practical: Increasing the performance of source routing. In *2nd International Conference on Mobile Ad-hoc and Sensor Networks (MSN), Hong Kong, China* (Dec. 2006).

[18] Buschmann, C., Fekete, S. P., Fischer, S., Kröller, A., and Pfisterer, D. Koordinatenfreies Lokationsbewusstsein. *it - Information Technology, Themenheft Sensornetze 47*, 4 (Apr. 2005), 70–77.

[19] Buschmann, C., and Fischer, S. Analyse des Kommunikationsaufwands für konsistentes Zeitbewusstsein. Tech. rep., 4. Fachgespräch Drahtlose Sensornetze, 2005.

[20] Buschmann, C., Fischer, S., Koberstein, J., and Luttenberger, N. Towards information centric application development for wireless sensor networks. In *Proceedings of the System Support for Ubiquitous Computing Workshop (UbiSys) at the 6th Annual Conference on Ubiquitous Computing* (Sept. 2004).

[21] Buschmann, C., Fischer, S., Luttenberger, N., and Reuter, F. Middleware for swarm-like collections of devices. *IEEE Pervasive Computing Magazine 2*, 3 (2003).

[22] Buschmann, C., Fischer, S., and Pfisterer, D. MarathonNet project. Funded by the Klaus Tschira Foundation, http://www.marathonnet.de.

[23] Buschmann, C., and Pfisterer, D. iSense: A modular hardware and software platform for wireless sensor networks. Tech. rep., 6. Fachgespräch "Drahtlose Sensornetze" der GI/ITG-Fachgruppe "Kommunikation und Verteilte Systeme", 2007.

[24] Buschmann, C., Pfisterer, D., and Fischer, S. Experimenting with computer swarms: a mobile platform based on blimps. In *Proceedings of the Second International Conference on Mobile Systems, Applications, and Services (MobiSys' 04), Boston, Massachusetts, USA* (June 2004), ACM SIGMOBILE.

[25] Buschmann, C., Pfisterer, D., and Fischer, S. From diodes to insights. In *2nd Workshop on Sensor Networks, INFORMATIK 2005* (Sept. 2005).

[26] Buschmann, C., Pfisterer, D., and Fischer, S. Kommunikationsaufwand von Verfahren zur Errichtung eines konsistenten Zeitbewusstseins - Eine quantitative Analyse. *PIK - Praxis der Informationsverarbeitung und Kommunikation, Sonderheft Wireless Sensor Networks*, 4 (Apr. 2005), 74–79.

[27] Buschmann, C., Pfisterer, D., and Fischer, S. Estimating distances using neighborhood intersection. In *Proceedings of the 11th IEEE International Conference on Emerging Technologies and Factory Automation, Prague, Czech Republic (ETFA' 06)* (Sept. 2006), pp. 314–321.

[28] Buschmann, C., Pfisterer, D., Fischer, S., Fekete, S. P., and Kröller, A. SpyGlass: Taking a closer look into sensor networks. In *Proceedings of the 2nd international conference on Embedded networked sensor system (SenSys' 04)* (Nov. 2004), ACM Press New York, NY, USA, pp. 301–302.

[29] Buschmann, C., Pfisterer, D., Fischer, S., Fekete, S. P., and Kröller, A. SpyGlass: A wireless sensor network visualizer. *ACM SIGBED Review 2*, 1 (Jan. 2005).

[30] Chang, K. K., and Gay, D. Language support for interoperable messaging in sensor networks. In *SCOPES '05: Proceedings of the 2005 workshop on Software and compilers for embedded systems* (New York, NY, USA, 2005), ACM Press, pp. 1–9.

[31] Costa, P., Mottola, L., Murphy, A. L., and Picco, G. P. TeenyLIME: transiently shared tuple space middleware for wireless sensor networks. In *MidSens '06: Proceedings of the international workshop on Middleware for sensor networks* (New York, NY, USA, 2006), ACM Press, pp. 43–48.

[32] Coulouris, G., Dollimore, J., and Kindberg, T. *Distributed Systems: Concepts and Design (4th Edition)*. Addison Wesley, 2005.

[33] Crossbow Technology Inc. Mica2Mote. http://www.xbow.com.

[34] Crossbow Technology Inc. Mote-VIEW Monitoring Software. http://www.xbow.com/Products/productsdetails.aspx?sid=88.

[35] Crossbow Technology Inc. Surge Network Viewer. http://www.xbow.com/Products/productsdetails.aspx?sid=86.

[36] Crossbow Technology Inc. MICAz wireless measurement system. http://www.xbow.com, June 2004.

[37] CST group, FU Berlin. Website of the Embedded Sensor Board ESB 430/2. http://www.inf.fu-berlin.de/inst/ag-tech/scatterweb_net/.

[38] Cugola, G., and Jacobsen, H.-A. Using publish/subscribe middleware for mobile systems. *The ACM SIGMOBILE Mobile Computing and Communications Review 6*, 4 (2002), 25–33.

[39] D. Jeff Dionne and Michael Durrant. uClinux – embedded linux/microcontroller project. http://www.uclinux.org/.

[40] Dunkels, A., Grönvall, B., and Voigt, T. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors 2004* (Nov. 2004).

[41] Elson, J., and Estrin, D. Time synchronization for wireless sensor networks. In *IPDPS '01: Proceedings of the 15th International Parallel and Distributed Processing Symposium* (2001).

[42] Elson, J., Girod, L., and Estrin, D. Fine-grained network time synchronization using reference broadcasts. *SIGOPS Oper. Syst. Rev. 36*, SI (2002), 147–163.

[43] Elson, J., and Römer, K. Wireless sensor networks: a new regime for time synchronization. *SIGCOMM Comput. Commun. Rev.* (2003).

[44] Estrin, D., Govindan, R., and Heidemann, J. Embedding the internet: introduction. *Commun. ACM 43*, 5 (2000), 38–41.

[45] Estrin, D., Govindan, R., Heidemann, J., and Kumar, S. Next century challenges: scalable coordination in sensor networks. In *MobiCom'99: Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking* (1999).

[46] Fekete, S. P., and Fischer, S. AUTONOMOS: A Distributed and Self-Regulating Approach for Organizing a Large System of Mobile Objects. DFG SPP 1183, http://www.auto-nomos.de/.

[47] Fekete, S. P., Fischer, S., Kröller, A., and Pfisterer, D. SwarmNet: Algorithmen und Protokolle füur Vernetzung und Betrieb großer Schwärme autonomer Kleinstprozessoren. DFG SPP 1126, http://www.swarmnet.de.

[48] Fekete, S. P., Kröller, A., Dennis Pfisterer, S. F., and

Buschmann, C. Neighborhood-based topology recognition in sensor networks. In *Algosensors'04, Lecture Notes in Computer Science* (July 2004), vol. 3121, Springer, pp. 123–136.

[49] Fekete, S. P., Kröller, A., Fischer, S., and Pfisterer, D. Shawn: The fast, highly customizable sensor network simulator. In *Proceedings of the Fourth International Conference on Networked Sensing Systems (INSS' 07)* (June 2007).

[50] Fischer, S., Luttenberger, N., Buschmann, C., and Koberstein, J. SWARMS: SoftWare Architecture for Radio-based Mobile Self-organizing Systems. DFG SPP 1140, http://www.swarms.de.

[51] Forsyth, W. S., and Safavi-Naini, R. Automated cryptanalysis of substitution ciphers. *Cryptologia 17*, 4 (Oct. 1993), 407–418.

[52] Frank, C., and Römer, K. Algorithms for generic role assignment in wireless sensor networks. In *SenSys'05: Proceedings of the 3rd international conference on Embedded networked sensor systems* (2005).

[53] Free Software Foundation, Inc. Gnu Compiler Collection (GCC), 1984. http://gcc.gnu.org/.

[54] Fung, W. F., Sun, D., and Gehrke, J. COUGAR: The network is the database. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data* (2002), pp. 621–621.

[55] Ganesan, D., Krishnamachari, B., Woo, A., Culler, D., Estrin, D., and Wicker, S. Complex behavior at scale: An experimental study of low-power wireless sensor networks. Tech. rep., UCLA, 2002.

[56] Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., and Culler, D. The nesC language: A holistic approach to networked embedded systems. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation* (New York, NY, USA, 2003), ACM Press, pp. 1–11.

[57] Gebhardt, N. The Irrlicht engine, 2003. http://irrlicht.sourceforge.net/.

[58] Girod, L., Stathopoulos, T., Ramanathan, N., Elson, J., Estrin, D., Osterweil, E., and Schoellhammer, T. A system for simulation, emulation, and deployment of heterogeneous sensor networks. In *Proceedings of the 2nd international conference on Embedded networked sensor system, SenSys 2004* (2004).

[59] Hadim, S., and Mohamed, N. Middleware challenges and approaches for wireless sensor networks. *IEEE Distributed Systems Online* (2006).

[60] Heidemann, J., Silva, F., Intanagonwiwat, C., Govindan, R., Estrin, D., and Ganesan, D. Building efficient wireless sensor networks with lowlevel naming. In *Proceedings of the Symposium on Operating Systems Principles, Banff, Canada* (2001).

[61] Heinzelman, W., Murphy, A., Carvalho, H., and Perillo, M. Middleware to support sensor network applications. *IEEE Network Magazine Special Issue* (2004).

[62] Heinzelman, W. R., Chandrakasan, A., and Balakrishnan, H. Energy-efficient communication protocol for wireless microsensor networks. In *HICSS'00: Proceedings of the 33rd Hawaii International Conference on System Sciences-Volume 8* (2000), p. 8020.

[63] Hellbrück, H., Lipphardt, M., Pfisterer, D., Ransom, S., and Fischer, S. Praxiserfahrungen mit MarathonNet - Ein mobiles Sensornetz im Sport. *PIK - Praxis der Informationsverarbeitung und Kommunikation 29*, 4 (2006).

[64] Hill, J., Horton, M., Kling, R., and Krishnamurthy, L. The platforms enabling wireless sensor networks. *Commun. ACM* (2004).

[65] Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., and Pister, K. System architecture directions for networked sensors. *SIGOPS Oper. Syst. Rev. 34*, 5 (2000), 93–104.

[66] Hill, J. L. *System architecture for wireless sensor networks*. PhD thesis, University of California, Berkeley, 2003. Adviser: David E. Culler.

[67] Hill, J. L., and Culler, D. E. Mica: A wireless platform for deeply embedded networks. *IEEE Micro 22*, 6 (Jan. 2002), 12–24.

[68] Huffman, D. A. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers 40*, 9 (Sept. 1952), 1098–1101.

[69] IEEE 802.11 Working Group. IEEE 802.11 Wireless local area networks. http://www.ieee802.org/11/.

[70] IEEE 802.15 Working Group. IEEE 802.15 WPAN Task Group 1 (TG1). http://www.ieee802.org/15/pub/TG1.html.

[71] IEEE 802.15 Working Group. IEEE 802.15 WPAN Task Group 4 (TG4). http://www.ieee802.org/15/pub/TG4.html.

[72] Institute of Electrical and Electronics Engineers (IEEE). IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std. 754-1985). http://grouper.ieee.org/groups/754/.

[73] Intanagonwiwat, C., Govindan, R., and Estrin, D. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *MobiCom '00: Proceedings of the 6th annual international conference on Mobile computing and networking* (2000), pp. 56–67.

[74] International, E. Standard ECMA-335: Common Language Infrastructure (CLI). Also: ISO/IEC 23271:2006.

[75] ISO Technical Committee 154. Data elements and interchange for-

mats - Information interchange - Representation of dates and times (ISO 8601:2004). Tech. rep., International Organization for Standardization (ISO), 2004. http://www.iso.org/.

[76] ISO/IEC 19757-2. RELAX NG. http://www.relaxng.org/, 2001.

[77] ITU-T Rec. X.680 and ISO/IEC 8824-1. Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation. http://www.itu.int/ITU-T/studygroups/com17/languages/, 2002.

[78] ITU-T Rec. X.690 (07/02). Information technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER). http://www.itu.int/ITU-T/studygroups/com17/languages/, 2002.

[79] ITU-T Rec. X.691 and ISO/IEC 8825-2. Information technology - ASN.1 encoding rules: Specification of Packed Encoding Rules (PER). http://www.itu.int/ITU-T/studygroups/com17/languages/, 2002.

[80] ITU-T Rec. X.693 (12/01). Information technology - ASN.1 encoding rules: XML encoding rules (XER). http://www.itu.int/ITU-T/studygroups/com17/languages/, 2001.

[81] J. Postel. User Datagram Protocol (RFC 768), Aug. 1980.

[82] Jaikaeo, C., Srisathapornphat, C., and Shen, C.-C. Querying and Tasking in Sensor Networks. In *SPIE's 14th Annual International Symposium on Aerospace/Defense Sensing, Simulation, and Control* (2000).

[83] Jiang, X., and Camp, T. A review of geocasting protocols for a mobile ad hoc network, 2002.

[84] Kahn, J. M., Katz, R. H., and Pister, K. S. J. Next century challenges: mobile networking for smart dust. In *MobiCom'99: Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking* (1999).

[85] Kaplan, E. D. *Understanding GPS: Principles and Applications*. Artech House, 1996.

[86] Karp, B., and Kung, H. T. GPSR: greedy perimeter stateless routing for wireless networks. In *Mobile Computing and Networking* (2000), pp. 243–254.

[87] Kasten, O., and Römer, K. Beyond event handlers: programming wireless sensors with attributed state machines. In *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks* (Piscataway, NJ, USA, 2005), IEEE Press, p. 7.

[88] Ko, Y. B., and Vaidya, N. H. Flooding-based geocasting protocols for mobile ad hoc networks. *Mob. Netw. Appl. 7*, 6 (2002), 471–480.

[89] Koberstein, J., Reuter, F., and Luttenberger, N. The XCast

approach for content-based flooding control in distributed virtual shared information spaces - design and evaluation. In *EWSN* (2004), pp. 188–203.

[90] Koshy, J., and Pandey, R. VMSTAR: synthesizing scalable runtime environments for sensor networks. In *SenSys 05: Proceedings of the 3rd international conference on Embedded networked sensor systems* (2005).

[91] Kröller, A., Fekete, S. P., Pfisterer, D., and Fischer, S. Deterministic boundary recognition and topology extraction for large sensor networks. In *Proceedings of the 17th ACM-SIAM Sympos. Discrete Algorithms (SODA' 06)* (2006), pp. 1000–1009.

[92] Kröller, A., Pfisterer, D., Buschmann, C., Fekete, S. P., and Fischer, S. Shawn: A new approach to simulating wireless sensor networks. In *Design, Analysis, and Simulation of Distributed Systems 2005 (DASD' 05)* (Apr. 2005), pp. 117–124.

[93] Kumar, V. Sensor: the atomic computing particle. *SIGMOD Rec. 32*, 4 (2003), 16–21.

[94] Kurkowski, S., Camp, T., Mushell, N., and Colagrosso, M. A visualization and analysis tool for Ns-2 wireless simulations: iNSpect. In *MASCOTS '05: Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 503–506.

[95] Landsiedel, O., Wehrle, K., and Götz, S. Accurate prediction of power consumption in sensor networks. In *Proceedings of the Second IEEE Workshop on Embedded Networked Sensors* (2005).

[96] Lee, I., and Sokolsky, O. Research challenges in embedded and hybrid systems. *SIGBED Rev. 1*, 1 (2004), 1–5.

[97] Levis, P., and Culler, D. Maté: A tiny virtual machine for sensor networks. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)* (2002).

[98] Levis, P., Gay, D., and Culler, D. Active sensor networks. In *Proceedings of the Second USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2005), Boston* (2005).

[99] Levis, P., Lee, N., Welsh, M., and Culler, D. TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In *Proceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003)* (2003). http://www.cs.berkeley.edu/~pal/research/tossim.html.

[100] Levis, P., Madden, S., Polastre, J., Szewczyk, R., Whitehouse, K., Woo, A., Gay, D., Hill, J., Welsh, M., Brewer, E., and

Culler, D. *TinyOS: An Operating System for Wireless Sensor Networks*. Springer, 2005, ch. 2.

[101] Li, N., and Hou, J. C. Topology control in heterogeneous wireless networks: problems and solutions. In *Proceedings of the 23st Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'04)* (2004).

[102] Lindholm, T., and Yellin, F. *The Java(TM) Virtual Machine Specification*. Prentice Hall PTR; 2 edition, 1999.

[103] Lipphardt, M., Hellbrück, H., Pfisterer, D., Ransom, S., and Fischer, S. Practical experiences on mobile inter-body-area-networking. In *Proceedings of the Second International Conference on Body Area Networks (BodyNets'07)* (2007).

[104] Liu, T., and Martonosi, M. Impala: A middleware system for managing autonomic, parallel sensor systems. *SIGPLAN Not. 38*, 10 (2003), 107–118.

[105] Lysecky, S., and Vahid, F. Automated generation of basic custom sensor-based embedded computing systems guided by end-user optimization criteria. In *Ubicomp* (2006).

[106] Madden, S., Franklin, M. J., Hellerstein, J. M., and Hong, W. Tag: a tiny aggregation service for ad-hoc sensor networks. In *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation* (New York, NY, USA, 2002), ACM Press, pp. 131–146.

[107] Madden, S. R., Franklin, M. J., Hellerstein, J. M., and Hong, W. TinyDB: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.* (2005).

[108] Mainwaring, A., Culler, D., Polastre, J., Szewczyk, R., and Anderson, J. Wireless sensor networks for habitat monitoring. In *WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications* (New York, NY, USA, 2002), ACM Press, pp. 88–97.

[109] Manjeshwar, A., and Agrawal, D. P. TEEN: A routing protocol for enhanced efficiency in wireless sensor networks. In *IPDPS'01: Proceedings of the 15th International Parallel and Distributed Processing Symposium* (2001), p. 189.

[110] Mannion, R., Hsieh, H., Cotterell, S., and Vahid, F. System synthesis for networks of programmable blocks. In *DATE'05 Proceedings of the Conference on Design, Automation and Test in Europe* (2005).

[111] Mauve, M., Widmer, J., and Hartenstein, H. A survey on position-based routing in mobile ad hoc networks. *Network, IEEE 15*, 6 (2001), 30–39.

[112] Mhatre, V., and Rosenberg, C. Homogeneous vs heterogeneous clustered sensor networks: a comparative study. In *IEEE International Conference on Communications (ICC 2004)* (2004).

[113] Microsoft Corporation. Windows Embedded CE. http://www.microsoft.com/windows/embedded/.

[114] Miyashita, M., Nesterenko, M., Shah, R., and Vora, A. Visualizing wireless sensor networks: An experience report. In *Proceedings of the 2005 International Conference on Wireless Networks, ICWN 2005, Las Vegas, Nevada, USA, June 27-30, 2005* (2005), pp. 412–419.

[115] Molla, M. M., and Ahamed, S. I. A survey of middleware for sensor network and challenges. In *ICPPW '06: Proceedings of the 2006 International Conference Workshops on Parallel Processing* (Washington, DC, USA, 2006), vol. 0, IEEE Computer Society, pp. 223–228.

[116] Moteiv Corporation. Tmote Sky. http://www.moteiv.com/products-tmotesky.php, 2005.

[117] Naval Research Laboratory. ns2sensors: NRL's sensor network extension to Ns-2. http://nrlsensorsim.pf.itd.nrl.navy.mil/.

[118] Object Management Group., Inc. (OMG). Common Object Request Broker Architecture (CORBA). http://www.omg.org/technology/documents/corba_spec_catalog.htm, 1991.

[119] Object Management Group., Inc. (OMG). A discussion of the object management architecture. http://www.omg.org, 1997.

[120] Object Management Group, Inc. (OMG). Unified Modeling Language. http://www.uml.org/, 1997.

[121] Object Management Group, Inc. (OMG). Model Driven Architecture (MDA). http://www.omg.org/mda/, 2001.

[122] Pan, J., Hou, Y. T., Cai, L., Shi, Y., and Shen, S. X. Topology control for wireless sensor networks. In *MobiCom '03: Proceedings of the 9th annual international conference on Mobile computing and networking* (New York, NY, USA, 2003), ACM Press, pp. 286–299.

[123] Park, S., Savvides, A., and Srivastava, M. B. SensorSim: A simulation framework for sensor networks. In *Proceedings of MSWiM 2000* (2000). http://nesl.ee.ucla.edu/projects/sensorsim/.

[124] Parr, T. ANTLR - ANother Tool for Language Recognition, 1989. http://www.antlr.org/.

[125] Pfisterer, D., Buschmann, C., Hellbrück, H., and Fischer, S. Data-type centric middleware synthesis for wireless sensor network application development. In *Proceedings of the Fifth Annual Mediterranean Ad Hoc Networking Workshop (Med-Hoc-Net' 06)* (2006), pp. 329–336.

[126] Pfisterer, D., Buschmann, C., Hellbrück, H., and Fischer, S. Supporting WSN application development using data type-centric middleware synthesis. Tech. rep., 5. Fachgespräch "Drahtlose Sensornetze" der GI/ITG-Fachgruppe "Kommunikation und Verteilte Systeme", 2006.

[127] Pfisterer, D., Fischer, S., Kröller, A., and Fekete, S. Shawn: Ein alternativer Ansatz zur Simulation von Sensornetzwerken. Tech. rep., 4. Fachgespräch "Drahtlose Sensornetze" der GI/ITG-Fachgruppe "Kommunikation und Verteilte Systeme", Mar. 2005.

[128] Pfisterer, D., Hellbrück, H., and Fischer, S. Fabric: Towards data type-centric middleware synthesis. In *Proceedings of the Euro-American Workshop on Middleware for Sensor Networks in conjunction with the International Conference on Distributed Computing (DCOSS 06)* (2006).

[129] Pfisterer, D., Lipphardt, M., Buschmann, C., Hellbrueck, H., Fischer, S., and Sauselin, J. H. Marathonnet: Adding value to large scale sport events - a connectivity analysis. In *Proceedings of the International Conference on Integrated Internet Ad hoc and Sensor Networks (InterSense'06)* (May 2006), A. Press, Ed., p. 12.

[130] Pfisterer, D., Wegner, M., Hellbrück, H., Werner, C., and Fischer, S. Energy-optimized data serialization for heterogeneous WSNs using middleware synthesis. In *Proceedings of The Sixth Annual Mediterranean Ad Hoc Networking Workshop (Med-Hoc-Net' 07)* (June 2007), pp. 180–187.

[131] Picco, G. P., Murphy, A. L., and Roman, G.-C. LIME: Linda meets mobility. In *ICSE '99: Proceedings of the 21st international conference on Software engineering* (1999), pp. 368–377.

[132] Pinto, J., Sousa, A., Lebres, P., Goncalves, G. M., and Sousa, J. MonSense - application for deployment, monitoring and control of wireless sensor networks. In *Proceedings of the REALWSN'06: Workshop on Real-World Wireless Sensor Networks* (2006).

[133] Polastre, J., Hill, J., and Culler, D. Versatile low power media access for wireless sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems* (New York, NY, USA, Nov. 2004), ACM Press, pp. 95–107.

[134] Polastre, J., Szewczyk, R., and Culler, D. E. Telos: enabling ultra-low power wireless research. In *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks* (2005), pp. 364–369.

[135] Ramanathan, N., Chang, K., Kapur, R., Girod, L., Kohler, E., and Estrin, D. Sympathy for the sensor network debugger. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems* (New York, NY, USA, 2005), ACM Press,

pp. 255–267.

[136] Ratnasamy, S., Karp, B., Shenker, S., Estrin, D., Govindan, R., Yin, L., and Yu, F. Data-centric storage in sensornets with GHT, a geographic hash table. *Mob. Netw. Appl.* (2003).

[137] Reichenbach, F., Born, A., Timmermann, D., and Bill, R. A distributed linear least squares method for precise localization with low complexity in wireless sensor networks. In *2nd IEEE International Conference on Distributed Computing in Sensor Systems* (Juni 2006), Springer, pp. 514–528. San Francisco, USA.

[138] Reichenbach, F., and Timmermann, D. On improving the precision of localization with minimum resource allocation. In *16th International Conference on Computer Communications and Networks, International Workshop on Wireless Mesh and Ad Hoc Networks* (August 2007). Honolulu, Hawaii, USA.

[139] Ringwald, M., and Römer, K. BitMAC: A deterministic, collision-free, and robust MAC protocol for sensor networks. In *Proceedings of 2nd European Workshop on Wireless Sensor Networks (EWSN 2005)* (Istanbul, Turkey, Jan. 2005), pp. 57–69.

[140] Ringwald, M., Römer, K., and Vitaletti, A. Passive inspection of sensor networks. In *Proceedings of the 3rd IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS' 2007)* (Santa Fe, New Mexico, USA, June 2007).

[141] Ringwald, M., and Römer, K. Monitoring and debugging of deployed sensor networks. 2. GI/ITG KuVS Fachgespräch Systemsoftware für Pervasive Computing, Arbeitsberichte des Instituts für Informatik, vol. 38/5, Oct. 2005.

[142] Römer, K., and Mattern, F. The design space of wireless sensor networks. *IEEE Wireless Communications 11*, 6 (Dec. 2004), 54–61.

[143] Römer, K., Kasten, O., and Mattern, F. Middleware challenges for wireless sensor networks. *SIGMOBILE Mob. Comput. Commun. Rev. 6*, 4 (2002), 59–61.

[144] Sack, J., and Urrutia, J. *Handbook of Computational Geometry.* North Holland; 1st edition, 2000.

[145] Sayood, K. *Introduction to data compression (2nd ed.).* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[146] ScatterWeb GmbH. ScatterNode. http://www.scatterweb.com/, 2005.

[147] Schiller, J., Liers, A., Ritter, H., Winter, R., and Voigt, T. Scatterweb - low power sensor nodes and energy aware routing. In *HICSS '05: Proceedings of the Proceedings of the 38th Annual Hawaii International Conference on System Sciences (HICSS'05) - Track 9* (Washington,

DC, USA, 2005), IEEE Computer Society, p. 286.3.

[148] Schmid, U., and Schossmaier, K. Interval-based clock synchronization. *Real-Time Syst.* (1997).

[149] Shannon, C. A mathematical theory of communication. *Bell System Technical Journal 27* (July 1948), 379–423.

[150] Shen, C.-C., Srisathapornphat, C., and Jaikaeo, C. Sensor information networking architecture and applications. *IEEE Personel Communication Magazine 8*, 4 (Aug. 2001), 52–59.

[151] Shepler, S., Callaghan, B., Robinson, D., Thurlow, R., Beame, C., Eisler, M., and Noveck, D. Network File System (NFS) version 4 Protocol (RFC 3530), Apr. 2003. http://www.ietf.org/rfc/rfc3530.txt.

[152] Shnayder, V., Hempstead, M., Chen, B.-R., Allen, G. W., and Welsh, M. Simulating the power consumption of large-scale sensor network applications. In *Proceedings of the 2nd international conference on Embedded networked sensor systems* (2004), ACM Press, pp. 188–200.

[153] Sivaharan, T., Blair, G., and Coulson, G. GREEN: A configurable and re-configurable publish-subscribe middleware for pervasive computing. In *Proc. Distributed Objects and Applications (DOA'05), Cyprus* (2005).

[154] Souto, E., Guimares, G., Vasconcelos, G., Vieira, M., Rosa, N., and Ferraz, C. A message-oriented middleware for sensor networks. In *Proceedings of the 2nd workshop on Middleware for pervasive and ad-hoc computing* (2004).

[155] Spirakis, P. Foundations of Adaptive Networked Societies of Tiny Artefacts (FRONTS). Research Academic Computer Technology Institute, 7th Framework Programme on Research, Technological Development and Demonstration.

[156] Srinivasan, R. XDR: External Data Representation Standard. Request for Comments (RFC) 1832, 1995.

[157] Stankovic, J. A. Research challenges for wireless sensor networks. *SIGBED Rev. 1*, 2 (2004), 9–12.

[158] Starner, T. Power and heat in ubiquitous computing. Summer School on Ubiquitous and Pervasive Computing, 2002. http://www.vs.inf.ethz.ch/events/dag2002/program/lectures.html.

[159] Starner, T. Web technologies - Thick clients for personal wireless devices. *IEEE Computer 35*, 1 (2002), 133–135.

[160] Su, W., and Akyildiz, I. F. Time-diffusion synchronization protocol for wireless sensor networks. *IEEE/ACM Trans. Netw. 13*, 2 (2005), 384–397.

[161] Sun, K., Ning, P., and Wang, C. TinySeRSync: secure and resilient time synchronization in wireless sensor networks. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security* (New York, NY, USA, 2006), ACM Press, pp. 264–277.

[162] Sun Microsystems, Inc. Java 2D Application Programmer Interface. http://java.sun.com/products/java-media/2D/.

[163] Sun Microsystems, Inc. The K virtual machine (KVM). http://java.sun.com/products/cldc/wp/.

[164] Sun Microsystems, Inc. RPC: Remote Procedure Call Protocol Specification Version 2 (RFC 1057), June 1988. http://www.ietf.org/rfc/rfc1057.txt.

[165] Szewczyk, R., Mainwaring, A., Polastre, J., Anderson, J., and Culler, D. An analysis of a large scale habitat monitoring application. In *Proceedings of the 2nd international conference on Embedded networked sensor system, SenSys 2004* (Nov. 2004).

[166] Szewczyk, R., Polastre, J., Mainwaring, A., and Culler, D. Lessons from a sensor network expedition. In *Proceedings of the First European Workshop on Sensor Networks (EWSN)* (Jan. 2004).

[167] Szymanski, B. K., Chen, G., Branch, J. W., and Zhu, L. SENSE: Sensor network simulator and emulator. http://www.cs.rpi.edu/~cheng3/sense/.

[168] T. Berners-Lee. Informational: Universal Resource Identifiers in WWW (RFC 1630), June 1994.

[169] Tanenbaum, A. S., and Steen, M. V. *Distributed Systems: Principles and Paradigms*. Prentice Hall PTR, 2001.

[170] Terfloth, K., Wittenburg, G., and Schiller, J. FACTS - a rule-based middleware architecture for wireless sensor networks. In *Proceedings of the First IEEE International Conference on Communication System Software and Middleware (COMSWARE 2006)* (2006).

[171] The Eclipse Foundation. Eclipse Modeling Framework Project (EMF). http://www.eclipse.org/modeling/emf/.

[172] The Eclipse Foundation. Eclipse - an open development platform, 2001. http://www.eclipse.org/.

[173] The TinyOS 2.x Working Group. TinyOS 2.0. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems* (2005).

[174] Tolle, G., and Culler, D. Design of an application-cooperative management system for wireless sensor networks. In *EWSN '05: Second European Workshop on Wireless Sensor Networks* (2005).

[175] University of Southern California, Information Sciences Institute (ISI). Ns-2: Network simulator-2. http://www.isi.edu/nsnam/ns/.

[176] van Greunen, J., and Rabaey, J. Lightweight time synchronization for sensor networks. In *WSNA '03: Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications* (New York, NY, USA, 2003), ACM Press, pp. 11–19.

[177] Varga, A. OMNeT++: Objective modular network testbed in C++. http://www.omnetpp.org.

[178] Vlajic, N., and Xia, D. Wireless sensor networks: To cluster or not to cluster? In *WOWMOM '06: Proceedings of the 2006 International Symposium on on World of Wireless, Mobile and Multimedia Networks* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 258–268.

[179] Walther, U. *Service Support for Virtual Groups in Mobile Environments*. PhD thesis, Technical University Carolo-Wilhelmina at Brunswick, Oct. 2003.

[180] Wattenhofer, M., Wattenhofer, R., and Widmayer, P. Geometric routing without geometry. In *12th Colloquium on Structural Information and Communication Complexity (SIROCCO), Le Mont Saint-Michel, France* (May 2005).

[181] Wattenhofer, R., Li, L., Bahl, P., and Wang, Y.-M. Distributed topology control for power efficient operation in multihop wireless ad hoc networks. In *Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM), Anchorage, Alaska* (Apr. 2001).

[182] Werner, C. *Optimierte Protokolle für Web Services mit begrenzten Datenraten*. Logos, Berlin, 2007.

[183] Werner, C., Buschmann, C., and Fischer, S. Advanced data compression techniques for SOAP web services. In *Modern Technologies in Web Services Research*, L.-J. Zhang, Ed. IGI Publishing, 2007, pp. 76–97.

[184] Wind River. VxWorks. http://www.windriver.com.

[185] Wittenburg, G., Terfloth, K., Villafuerte, F. L., Naumowicz, T., Ritter, H., and Schiller, J. Fence monitoring - experimental evaluation of a use case for wireless sensor networks. In *Proceedings of the 4th European Conference on Wireless Sensor Networks (EWSN'07)* (Delft, The Netherlands, Jan. 2007).

[186] Wolenetz, M., Kumar, R., Shin, J., and Ramachandran, U. Middleware guidelines for future sensor networks. In *First workshop on broadband advanced sensor networks (BaseNets)* (Oct. 2004).

[187] World Wide Web Consortium (W3C). Recommendation: Extensi-

ble Markup Language (XML) 1.0, Feb. 1998.

[188] World Wide Web Consortium (W3C). Recommendation: Namespaces in XML, Jan. 1999.

[189] World Wide Web Consortium (W3C). Member submission: Web services description language (WSDL) 1.1, Mar. 2001.

[190] World Wide Web Consortium (W3C). Recommendation: SOAP Version 1.2 Part 0: Primer, June 2003.

[191] World Wide Web Consortium (W3C). Recommendation: XML Schema Part 1 – Structures, Second Edition, Oct. 2004.

[192] World Wide Web Consortium (W3C). Recommendation: XML Schema Part 2 – Datatypes, Second Edition, Oct. 2004.

[193] Xu, Y., Bien, S., Mori, Y., Heidemann, J., and Estrin, D. Topology control protocols to conserve energy in wireless ad hoc networks. Tech. Rep. 6, University of California, Los Angeles, Center for Embedded Networked Computing, Jan. 2003.

[194] Xu, Y., Heidemann, J., and Estrin, D. Geography-informed energy conservation for ad hoc routing. In *Proceedings of the 7th annual international conference on Mobile computing and networking* (2001).

[195] Ye, W., Heidemann, J., and Estrin, D. An energy-efficient MAC protocol for wireless sensor networks. In *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE* (2002), vol. 3, pp. 1567–1576.

[196] Yoneki, E. Mobile applications with a middleware system in publish-subscribe paradigm. In *Proceedings of the 3rd Workshop on Applications and Services in Wireless Networks, Bern, Switzerland* (2003).

[197] Younis, M., Youssef, M., and Arisha, K. Energy-aware routing in cluster-based sensor networks. In *10th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'02)* (2002).

[198] Zhang, P., Sadler, C. M., Lyon, S. A., and Martonosi, M. Hardware design experiences in ZebraNet. In *Proceedings of the 2nd international conference on Embedded networked sensor systems* (2004), ACM Press, pp. 227–238.

[199] Zhou, G., He, T., Krishnamurthy, S., and Stankovic, J. A. Impact of radio irregularity on wireless sensor networks. In *MobiSys '04: Proceedings of the 2nd international conference on Mobile systems, applications, and services* (2004), pp. 125–138.

[200] Zhou, G., He, T., Krishnamurthy, S., and Stankovic, J. A. Models and solutions for radio irregularity in wireless sensor networks. *ACM Trans. Sen. Netw. 2*, 2 (2006), 221–262.

[201] ZigBee Alliance. The ZigBee 1.0 specification. http://www.zigbee.org/.