Chapter 1

Introduction

More than a decade ago, the idea of wireless sensor networks (WSNs) was first introduced. These networks consist of small, resource-constrained devices, only equipped with a low-end microcontroller and a radio transceiver for wireless communication. The devices are usually battery driven, and monitor their environment with the help of attached sensors. They perform distributed in-network data processing, in contrast to centralized approaches were devices are controlled by a single unit. This way, algorithms become more scalable, and communication costs can be reduced by sending only relevant data to a central station. WSNs have become a well-established research area, which led to a large number of available algorithms, hardware platforms, and operating systems.

In recent years, however, the focus has shifted to the *Internet of Things* (IoT), which aims at the interconnection of every-day devices such as smartphones or radio-frequency identification (RFID) chips. With the IoT, even more new and different heterogeneous platforms were introduced.

Unfortunately, this diversity of hardware platforms and operating systems has not been successfully addressed yet. Operating systems such as Contiki or TinyOS cover a number of hardware architectures, but by far not all available systems. Adding support for a new platform is usually difficult and requires low-level development and a profound understanding of the underlying hardware; it is simply impossible to run such an operating system on a smartphone. Interoperability is also only rarely achieved: Even two nodes of the same kind cannot communicate with each other when running different operating systems.

The situation is similarly desperate when considering standard algorithm implementations; each operating system usually uses its own—very system-specific—implementations of routing, localization, or time-synchronization algorithms. Consequently, there are many implementations of the same algorithm, each with its own peculiarities. Even worse, each implementation is also tested and evaluated separately, which is unnecessary overhead that is better avoided.

Together with the separate solutions in hardware platforms, operating systems, and algorithm implementations, also very specialized testbed installations are used. Usu-

Dieses Werk ist copyrightgeschützt und darf in keiner Form vervielfältigt werden noch an Dritte weitergegeben werden.
Es gilt nur für den persönlichen Gebrauch.

ally, testbeds consist of only homogeneous nodes. First, because maintenance costs are considerably lower when dealing only with one kind of node. Second, an institution that sets up a testbed generally uses only a single operating system; hence, multiple hardware platforms are not required.

The current state of isolated hardware and software platforms as well as present testbed installations do not adequately reflect the arisen demands on testing algorithms in large networks with various node architectures. Heterogeneity plays an essential role for ongoing and future research. It is no longer sufficient to address only particular subareas of the target domain; instead, the entire problem space must be considered at once. Already several approaches are available, mostly in terms of generic operating systems or middleware solutions, but none of these systems fully solves the problem.

We state three fundamental factors that are essential to successfully address heterogeneity in distributed, embedded systems:

- **New programming paradigms.** While modern programming paradigms such as object-oriented software development and generic programming techniques are already well established for desktop computers, this is not the case for embedded systems. The dominant programming language is still C, although more promising and also more efficient results can be achieved by a well-considered use of C++.

- **Heterogeneous testbed design.** When building a testbed for distributed embedded systems, heterogeneity must be the main factor in all fundamental design decisions. This is of utmost importance to sufficiently address the demands of future research. Heterogeneity, in this context, does not only cover the purely wireless nodes, but also deployed sensors and—if any—actuators.

- **Novel communication channels.** With the increasing importance of heterogeneity, using only standard communication channels is no longer sufficient. We may deal with devices that cannot communicate with others because of incompatible radios or distant positioning. Hence, the default type of communication must be extended to more flexible and dynamic approaches.

We address each of these topics separately in detail, and provide promising solutions to be used in current and future installations of distributed heterogeneous embedded systems. We address the full cycle of software development for these systems. Beginning with general programming paradigms suitable for an efficient usage even on low-end microcontrollers, we thereon present the design and final installation of a heterogeneous testbed, capable of dealing with many future requirements especially arising in the context of the IoT. Finally, we develop novel communication links, which enable advanced evaluation and testing techniques, for example unit testing for networks of distributed embedded systems.

Dieses Werk ist copyrightgeschützt und darf in keiner Form vervielfältigt werden noch an Dritte weitergegeben werden.
Es gilt nur für den persönlichen Gebrauch.

Organization of this Work. In Chapter 2 we describe successful software design techniques to develop algorithms and applications for heterogeneous embedded systems. We therefore study several existent approaches, and see that none of them is sufficiently generic and efficient at the same time. We show that it is possible to transfer modern programming techniques—using C++ and templates—to embedded systems; such a design is already successfully used in several software libraries for desktop computers, but so far it has not been used on low-end microcontrollers.

In Chapter 3 we present the *Wiselib*, a successful implementation of the programming paradigms introduced before. The Wiselib is an algorithms library for embedded systems, completely developed in C++, and running on top of various standard operating systems such as Contiki and TinyOS. The idea is to develop an algorithm—for instance, routing or time-synchronization—once, and compile it for different platforms without changing a single line of algorithm code. We also show that by using a modern programming language, the compiler is able to generate very efficient code, both regarding code size and execution time. At best, potential glue code is completely removed by the compiler, and the algorithm is built as if compiled directly for the corresponding platform.

In Chapter 4 we present the design and installation of a heterogeneous testbed for distributed embedded systems. We installed load sensors beneath the floor of a hallway at our university, and passive infrared sensors (PIRs) at the walls; both sensor types are connected to wireless sensor nodes, allowing the development of sophisticated in-network data-processing algorithms such as target tracking. Furthermore, actuators were installed, each consisting of a light-emitting diode (LED) and a speaker unit to play sound samples. Employing these units, the sensor nodes can interact with passers-by, providing possibilities to, for example, playing interactive games.

In Chapter 5 we study selected applications that were implemented on the hallway testbed. We give an example of target tracking, showing the hallway's suitability for security applications or the field of Ambient Assisted Living (AAL). We also present an example related to medical testing, where the hallway can be used to assist a physician in an examination, providing more reliable results than possible by current standard methods. Furthermore, we give an outlook on future applications, where the hallway is used for advanced algorithm development.

In Chapter 6 we introduce a novel mechanism of connecting nodes that cannot communicate directly with each other. Examples for such an inability are distant testbeds or radios using different communication frequency bands. We describe the idea of virtualized communication, where messages are sent both via the physical radio of a node and the Internet using a gateway; in the latter case, messages are injected at the destination node as if received over ordinary radio. Hence, the virtualization is fully transparent to the upper layer. We also connect real nodes with simulated ones using this mechanism, which considerably enhances general debugging possibilities for distributed embedded systems.

Dieses Werk ist copyrightgeschützt und darf in keiner Form vervielfältigt werden noch an Dritte weitergegeben werden.
Es gilt nur für den persönlichen Gebrauch.

The papers forming the foundation of this work were prepared in collaboration with other people. Above all, **Sándor P. Fekete** and **Alexander Kröller** contributed ideas and thoughts to almost all aspects of this work.

Many people—almost all partners from the EU-projects WISEBED [WIS11] and FRONTS [FRO11], as well as several students from our university—contributed implementations to the Wiselib. Most notably, we had a close collaboration in the preparation of the fundamental Wiselib paper [BCF^{+}10] with **Ioannis Chatzigiannakis** and **Christos Koninis**.

The construction of the hallway testbed was mainly supported by three colleagues from our institute: **Henning Hasemann**, **Tom Kamphans**, and **Max Pagel** contributed much time and effort to successfully build the initial setup and maintain the running installation.

The part about virtual communication was done with partners from the corresponding EU-project WISEBED. **Ioannis Chatzigiannakis** and **Christos Koninis** contributed to the foundations, focusing on flexible interconnection of distant testbeds, and re-configuration for changing topologies in a network. **Dennis Pfisterer** and **Daniel Bimschas** developed the underlying software framework for gateway computers, building the base for testbed inter-connection.

Dieses Werk ist copyrightgeschützt und darf in keiner Form vervielfältigt werden noch an Dritte weitergegeben werden.
Es gilt nur für den persönlichen Gebrauch.

Part I

Generic Algorithm Development

Dieses Werk ist copyrightgeschützt und darf in keiner Form vervielfältigt werden noch an Dritte weitergegeben werden.
Es gilt nur für den persönlichen Gebrauch.

Chapter 2

Software Design Techniques for Heterogeneous Embedded Systems

This chapter discusses software design techniques for the development of generic applications for distributed, heterogeneous embedded systems by taking additional targets such as simulation environments or emulators into account. Since we deal with tiny embedded systems—for instance, microcontrollers with only a few kilobytes of program memory—the main objective is an efficient and generic concept, in which the cost of generality regarding code space or runtime overhead is as small as possible. We learn that by using the modern programming language C++ and only standard language features, it is possible to create a software library that fulfills all of our requirements.

2.1 Problem Statement

In the last decade, WSNs have become an important and well-studied research field. We have seen a tremendous amount of algorithm developments, especially designed for low-end microcontrollers and focusing on the limiting resource constraints of these platforms. Concurrently, also a variety of sensor nodes were designed—platforms equipped with tiny microcontrollers and a radio transceiver, usually driven by a battery. In recent time, the focus has changed to the IoT, covering even more platforms, and establishing the vision of joining very different electronic devices together to appear to the outside world as a single homogeneous system.

Software development for heterogeneous embedded systems is a highly challenging task. With the diversity of available platforms, there is also a correspondingly high amount of different operating systems in use, considerably complicating the issue of finding a generic solution that adapts perfectly well to the individualities of each available system. This diversity, with respect to both the availability of operating systems and the wide differences in hardware capabilities, makes it difficult to find a solution that is adopted by the various communities. Figure 2.1 shows potential target platforms from the wireless embedded world. Our targets range from tiny sensor nodes

Dieses Werk ist copyrightgeschützt und darf in keiner Form vervielfältigt werden noch an Dritte weitergegeben werden.
Es gilt nur für den persönlichen Gebrauch.

(a) iSense sensor node. (b) TelosB sensor node. (c) iMote2 sensor node.

(d) Ordinary wireless router. (e) Android-based smartphone. (f) Apple's iPod Touch.

Figure 2.1: Potential target platforms. *Various target platforms for a software library for distributed, embedded systems. Platforms range from tiny sensor nodes over ordinary wireless routers to powerful smartphones.*

up to smartphones or wireless routers. We see that the capabilities of the selected platforms—even in this small subset—are very diverse.

Furthermore, each of these systems is usually equipped with a radio transceiver, leading to wirelessly connected distributed systems. Hence, an important design factor for a software architecture is the possibility of communication between the various nodes: Keeping in mind that two platforms may be totally distinct, with different word lengths, different byte ordering, and a different hardware architecture. The final vision is a framework, where any algorithm can be implemented once, and then run on any supported platform. Even more, such an algorithm can be put on heterogeneous nodes, where it may be fully transparent to the algorithm that it runs on different kinds of hardware and that it may communicate wirelessly with entirely different hardware architectures.

Dieses Werk ist copyrightgeschützt und darf in keiner Form vervielfältigt werden noch an Dritte weitergegeben werden.
Es gilt nur für den persönlichen Gebrauch.

There are several possibilities to develop a system that fulfills our requirements. Beginning with the design of a new embedded operating system, up to a middleware approach that runs on top of existing systems. Since there are already a lot of different operating systems available, many of them advantageous for specific hardware platforms, it may be too complex to develop a generic solution from scratch. Moreover, such newly designed systems may not be adopted by developers who are already familiar with an existing system. Finally, porting to a new platform can be a very complex task, involving the understanding of low-level platform development. Hence, a middleware-like approach is the most promising one: A framework that runs on top of existing operating systems. However, again it is important to not force developers to learn the handling of a completely new system. Keeping hold of as much as possible well-known system behavior is a crucial factor for the adoption of a new software architecture by the general public.

Another issue is that we have basically two potential target groups dealing with distributed, embedded systems: Theorists, mainly developing provable optimal algorithms, but often do not evaluate their ideas in practice; and practitioners, keen in the development of algorithms on hardware, but often using simple protocols due to complex problems when dealing with physical characteristics of real testbeds. An interesting opportunity for a newly designed software framework is to bridge the gap between these groups. When such a framework creates a simple development environment where algorithms can be easily implemented, it would be adopted by theorists, resulting in implementations of mathematically profound algorithms and protocols that can be used on real hardware. On the other hand, practitioners can benefit from this progress, having more and most notably more reliable implementations available. However, for a long-term success story of such a collaboration, it is important that both parties work closely together.

Putting all the requirements together, the development of a generic software architecture for heterogeneous embedded systems is a sophisticated task. Even on ordinary desktop systems, where restrictions concerning code size or runtime issues are usually very lax, it is not obligatory to support different operating systems. This is due to several reasons. First of all, we usually deal with different compiler manufacturers—for example, Visual Studio on Windows systems versus the wide-spread GNU Compiler Collection (GCC)[1]. Next, even if the same compiler is used, one must still deal with different library versions. This is especially problematic with Linux, since library versions are often even different under several distributions. Furthermore, it is usually more convenient to develop for an operating system when using native functionality. A prominent example is multi-threading, which is very system-specific for Windows and Unix systems; if not using platform-independent libraries such as Boost to hide these peculiarities.

There have been attempts to circumvent these problems. Well-known examples are platform independent solutions such as Java or the .NET framework. However, even

[1]The GCC is mostly used on Linux or Apple's MAC OS, but is also available for Windows systems.

Dieses Werk ist copyrightgeschützt und darf in keiner Form vervielfältigt werden noch an Dritte weitergegeben werden.
Es gilt nur für den persönlichen Gebrauch.

those approaches show certain disadvantages. While the .NET framework runs only on Windows systems, alternative implementations such as Mono are still not fully compatible. Java, on the other hand, is much more promising, while still not completely successful. When developing Web or database applications, the code is usually fully platform independent; serial communication with attached systems, in contrast, is surprisingly problematic. The RXTX library, which is commonly used for such tasks—for example, communication with a device connected over a FTDI chip—comes with native implementations via JNI, and provides pre-compiled libraries for various platforms. This complicates the transfer of applications to other systems, because these libraries must be distributed separately.

These problems known from ordinary desktop systems are even more existent in the embedded world. Not only that the operation systems vary in the used programming language and basic design issues (i.e., event-driven vs. multi-threaded), but also the hardware capabilities are very different. There are even nodes available with only a few kilobytes of program memory, or only one kilobyte of volatile memory. Hence, a crucial factor for a successful software framework in the embedded world is efficiency. This incorporates both code space and runtime issues. The former leads to a demand for a slim solution, in contrast to present middleware approaches, where the middleware itself often takes most of the available memory. The latter—runtime efficiency—is far more important on tiny microcontrollers than on ordinary desktop systems, since processor speed can differ in a magnitude of thousands.

Possibilities in software design are considerably influenced by the chosen programming language. Today, most applications for embedded systems are written in pure C, while recently there is also the use of modern programming languages such as C++ observable. Java, on the other hand, is only very sporadically visible due to the requirement of the availability of an entire virtual machine—which is too much overhead for tiny microcontrollers. We analyze the impact of choosing the object-oriented programming language C++ and evaluate potential overhead when only certain features of the language are carefully selected. If it is possible to use such a modern programming language for the development of a software architecture on embedded systems, the resulting approach can have formidable advantages over existing solutions: A clear and type-safe design that is easily usable, while generating very efficient code.

On desktop systems, there are already well-known libraries established, written in C++. Examples are the Standard Template Library (STL) [Jos99], the Computational Geometry Algorithms Library (CGAL) [Ket99], and Boost [Kar05]. They share a prominent programming concept, resulting in highly efficient code: C++ templates. With the aid of this concept, one can develop generic and exchangeable applications, where the price of generality is paid at compile time. If adapted to embedded systems, it would result in a powerful software architecture that can cover very different hardware and software platforms, and a compiled application can adapt perfectly well to the corresponding needs.

Section 2.2 discusses the problem space by exploring potential target platforms of a software architecture for heterogeneous embedded systems. In Section 2.3, an

Dieses Werk ist copyrightgeschützt und darf in keiner Form vervielfältigt werden noch an Dritte weitergegeben werden.
Es gilt nur für den persönlichen Gebrauch.

Table 2.1: Evaluation of potential target platforms. *The columns refer to the sensor node type, the standard operating system, the type of microcontroller, the programming language for it, what kind of dynamic memory is available, the amount of ROM and RAM, and the word length.*

Hardware	**CPU**	**Firmware/OS**	**Language**	**Dyn Mem**	**ROM**	**RAM**	**Bits**
iSense	Jennic JN5139	iSense FW	C++	Physical	128kB	96kB[a]	32
iSense v2	Jennic JN5148	iSense FW	C++	Physical	512kB	128kB[b]	32
SCW MSB-A2	NXP LPC2387	μkleos	C	None	512kB	98kB	32
SCW MSB430	MSP430	SCW2, Contiki	C	None	48kB	10kB	16
TelosB[c]	MSP430	Contiki,TinyOS	C, nesC	Physical	48kB	10kB	16
G-Node	MSP430	TinyOS	nesC	Physical	116kB	8kB	16
INGA	ATmega1284P	Contiki	C	Physical	128kB	16kB	8
MicaZ	ATmega128L	Contiki,TinyOS	C, nesC	Physical	128kB	4kB	8
Waspmote	ATmega1281	Waspmote API	C/C++	Physical	128kB	8kB	8
Arduino Nano 2.3	ATmega168	Arduino SW	C/C++	Physical	16 kB	1kB	8
Arduino Nano 3.0	ATmega328	Arduino SW	C/C++	Physical	32 kB	2kB	8
iMote2	Intel XScale	TinyOS, Linux	nesC,C/C++	Physical	32MB	32MB	32
iPhone, iPod	ARM	iOS	C++, Obj-C	Virtual	$\geq$8GB	$\geq$128MB	32
Smartphone	ARM	Android	C/C++, Java	Virtual	$\geq$2GB	$\geq$64MB	32
Desktop PC	various	Shawn	C++	Virtual	unlimited	unlimited	32/64
Desktop PC	(ATmega128L)	TOSSIM	nesC	(Physical)	unlimited	unlimited	(8)
Desktop PC	various	Win/Linux/MAC	C/C++	Virtual	unlimited	unlimited	32/64

[a] shared for program and data
[b] shared for program and data
[c] identical in construction with Tmote Sky

overview on existing approaches is presented, both in the embedded world and successful libraries from desktop systems, followed by a discussion of the possibilities of using C++ on embedded systems in Section 2.4. Finally, a promising software design is presented in Section 2.5.

2.2 Problem Space

When developing a software architecture for heterogeneous embedded systems, one must deal with a great variety of different hardware and software platforms. Table 2.1 shows an overview of platforms that are important and interesting both in the area of sensor networks and the upcoming field of IoT.

The hardware platforms vary from sensor nodes such as INGA [BKW11, KBPW11], iSense [BP07], or TelosB [PSC05] over smartphones and simulation environments to ordinary desktop computers. The latter are taken into account to be able to integrate such systems directly into existing networks—for instance, as a sink for sensed data or as a high-performance computation node, useful for certain kinds of algorithms. These platforms come with very different kinds of processor architectures, ranging from tiny microcontrollers such as the MSP430 or the Atmel ATmega to powerful processors such as Intel XScale or ARMs. As with the hardware, there are also various operating systems in use. There are both system-specific implementations—for

Dieses Werk ist copyrightgeschützt und darf in keiner Form vervielfältigt werden noch an Dritte weitergegeben werden.
Es gilt nur für den persönlichen Gebrauch.