Mälardalen University Licentiate Thesis
No.45

Towards TCP/IP for Wireless Sensor Networks

Adam Dunkels

March 2005

Swedish Institute of Computer Science
Stockholm, Sweden

Department of Computer Science and Electronics
Mälardalen University
Västerås, Sweden

ISSN 1651-9256
ISBN 91-88834-96-4
Printed by Arkitektkopia, Västerås, Sweden
Distribution: Mälardalen University Press

Abstract

Wireless sensor networks are composed of large numbers—up to thousands—of tiny radio-equipped sensors. Every sensor has a small microprocessor with enough power to allow the sensors to autonomously form networks through which sensor information is gathered. Wireless sensor networks makes it possible to monitor places like nuclear disaster areas or volcano craters without requiring humans to be immediately present. Many wireless sensor network applications cannot be performed in isolation; the sensor network must somehow be connected to monitoring and controlling entities.

This thesis investigates a novel approach for connecting sensor networks to existing networks: by using the TCP/IP protocol suite in the sensor network, the sensors can be directly connected to an outside network without the need for special proxy servers or protocol converters.

Bringing TCP/IP to wireless sensor networks is a challenging task, however. First, because of their limited physical size and low cost, sensors are severely constrained in terms of memory and processing power. Traditionally, these constraints have been considered too limiting for a sensor to be able to use the TCP/IP protocols. In this thesis, I show that even tiny sensors can communicate using TCP/IP. Second, the harsh communication conditions make TCP/IP perform poorly in terms of both throughput and energy efficiency. With this thesis, I suggest a number of optimizations that are intended to increase the performance of TCP/IP for sensor networks.

The results of the work presented in this thesis has had a significant impact on the embedded TCP/IP networking community. The software developed as part of the thesis has become widely known in the community. The software is mentioned in books on embedded systems and networking, is used in academic courses on embedded systems, is the focus of articles in professional magazines, is incorporated in embedded operating systems, and is used in a large number of embedded devices.

To Castor, Morgan and Maria

Acknowledgements

First of all, I would like to thank my colleague and co-advisor Thiemo Voigt for being such an inspiring person to work with. This thesis would not have been possible without Thiemo's support and encouragement! Many thanks also go to my advisor Mats Björkman, for being a positive individual and for having the attitude that all problems are small problems. I am also grateful to Juan Alonso who not only was the driving force behind the sensor network research at SICS, but also is a very nice person to work with. My thanks also go to Bengt Ahlgren, lab manager of the Computer and Network Architectures lab at the Swedish Institute of Computer Science, for giving me the opportunity to do research at SICS.

My gratitude also goes to Henrik Abrahamsson for being a wonderful discussion partner on subjects ranging from science, mathematics, and research methods, to music, TV, and culture, as well as life in general. Thanks also go to all the members CNA lab for contributing to an inspiring work environment: Lars Albertsson, Anders Andersson, Laura Feeney, Björn Grönvall, and Ian Marsh. Other notable persons at SICS are Sverker Janson, Joakim Eriksson, Niclas Finne, and Martin Nilsson. Thanks also go to Joe Armstrong and to Hartmut Ritter for interesting discussions.

Many thanks to the everyone who have taken part of the development of my software. In particular, this includes Ullrich von Bassewitz, who have been a great discussion partner during the development of uIP and Contiki, Oliver Schmidt who is actively working on Contiki together with me, and Leon Woestenberg, who are heading the continued development of the lwIP stack. Also, everyone who have contributed code, patches, and bug reports to the software also deserves a big "thank you"!

Thanks also go to my mother Kerstin for being supportive and for taking a general interest in my work. I am also endlessly grateful to my late father Andrejs who, despite no longer being with us, still manages to guide me as a

thinker, researcher, and in being a father.

Finally, I would like to thank my wife Maria for making life outside of work even more enjoyable than the work itself and for being a great mother for our sons Morgan and Castor.

This work in this thesis is in part supported by VINNOVA, Ericsson, SITI, SSF, the European Commission under the Information Society Technology priority within the 6th Framework Programme, and the European Commission's 6th Framework Programme under contract number IST-004536. The Swedish Institute of Computer Science is sponsored by TeliaSonera, Ericsson, SaabTech, FMV, Green Cargo, ABB, and Bombardier Transportation AB.

Contents

I

Thesis

Chapter 1

Introduction

Wireless sensor networks consist of large numbers of sensors equipped with a small microprocessor, a radio transceiver, and an energy source, typically a battery. The sensors nodes autonomously form networks through which sensor readings are transported. Applications of wireless sensor networks can be found in such diverse areas as wild-life habitat monitoring [59], forest fire detection [76], alarm systems [31], medicine [84], and monitoring of volcanic eruptions [85].

In order to make large scale networks feasible, the sensor nodes are required to be physically small and inexpensive. These requirements severely constraints the available resources on each sensor node in terms of memory size, communication bandwidth, computation speed, and energy.

Many wireless sensor network applications do not work well in isolation; the sensor network must somehow be connected to monitoring and controlling entities. Since communication within the sensor network is done using short-range radios, a straightforward approach to connecting the sensors with the controlling entities is to deploy the controlling entities physically close to the sensor network. In many cases however, placing those entities close to the sensors, and hence to the phenomenon being observed, is not practical. Instead, by connecting the sensor network and the controlling entities to a common network infrastructure the sensors and the controlling entities can communicate without being physically close to each other.

Because of the success of the Internet, the TCP/IP protocols have become the de-facto standard protocol stack for large scale networking. However, conventional wisdom states that TCP/IP is inherently unsuitable for communica-

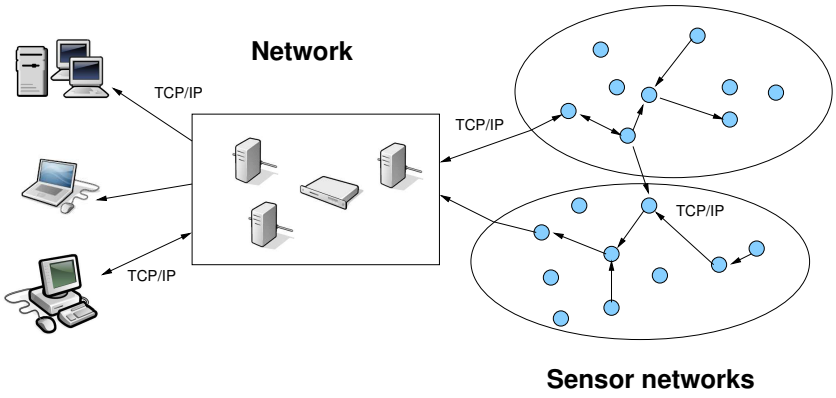

Figure 1.1: Using TCP/IP both outside of and inside the wireless sensor network

tion within wireless sensor networks, because of the extreme communication conditions in sensor networks. Hence, a large number of protocols specifically tailored for sensor networks have been developed. While it is unquestionably true that the TCP/IP protocols were not designed to run in the kind of environments where sensor networks are envisioned, the claim that TCP/IP is inherently unsuitable for wireless sensor networks has not been verified.

The purpose of this licentiate thesis is to lay the groundwork for exploring the use of TCP/IP for wireless sensor networks. Using TCP/IP for sensor networks allows connecting the sensor networks directly to IP network infrastructures, as shown in Figure 1.1. In a set of four papers I present the software for an experimental platform, describe the problem area, and propose a set of mechanisms that are intended to allow TCP/IP to be efficiently used in wireless sensor networks. The software platform consists of a lightweight implementation of the TCP/IP protocol stack and an equally lightweight and flexible operating system. Both the operating system and the protocol implementation are specifically designed to run on resource constrained sensor nodes.

1.1 Method

In order to explain and motivate the work in this thesis, I use two perspectives: the *engineering perspective* and the *research perspective*. Engineering

is about finding solutions to complex problems, within a given set of limitations. Research is about developing understanding. In experimental computer science [21], this understanding commonly is developed by producing artifacts and solving complex problems—*doing engineering*—and drawing conclusions from the solutions. A single solution may not be possible to generalize, but taken together, a number of solutions can be said to span a *solution space* to a particular problem. Exploring, characterizing, and analyzing this solution space develops understanding for the character of the problem. This classification of engineering and research is based on definitions from Brooks' [15] and Phillips and Pugh [66].

The research in this thesis has mostly been exploratory. The problem area was not defined in advance, but has been developed as part of the thesis work. The exploratory method starts with finding an interesting question to answer. The question usually involves an interesting problem to solve. The problem is then solved in a set of different ways, using either different tools or methods or variations of the same method. Based on observations of the solutions, or of the process leading to the solutions, an initial answer to the question can be formulated. From the answer and the solutions to the problems, it might be possible to generalize the question into a hypothesis. This hypothesis can then be tested using experimentation in order to validate or invalidate it. The process of testing the hypothesis typically leads to a number of questions that need to be answered. Thus the research process is iterative in that a research question leads to a hypothesis, which leads to further questions.

In this thesis, the initial question was if the TCP/IP protocol stack could be implemented so that it would fit in a severely memory constrained system. After twice solving the problem of implementing TCP/IP with limited resources, the question could be answered: the TCP/IP protocol stack can be implemented using very small amounts of memory. This observation lead to the generalized question if TCP/IP could be useful is wireless sensor networks. This generalization was made because of the similarities of parts of the problem domains—sensor network nodes have severely limited memory resources—as well as intuition developed when answering the initial question. The event-driven nature of sensor networks seemed to fit the event-driven design of the small TCP/IP implementations. Furthermore, it appeared that many of the problems with TCP/IP in sensor networks could be solved with relatively straight-forward mechanisms. These observations lead to the hypothesis that TCP/IP could be a viable alternative for wireless sensor networks. This thesis takes the first steps towards validating or invalidating this hypothesis.

1.2 Research Issues

This thesis takes the first steps towards the use of the TCP/IP protocol suite in wireless sensor networks. This section summarizes the research issues that are identified and treated in this thesis.

Many of these issues are of the engineering kind: a problem that needs a solution that is not only correct, but also is able to work within the available limitations. These issues are the primary focus of papers A and B. Papers A and B solve the specific problems of implementing TCP/IP on a limited device and on designing an operating system for sensor nodes that allows rapid prototyping and experimentation.

Papers C and D focus on the research challenges involved in TCP/IP for wireless sensor networks. The formulation of these challenges are based on the software artifacts developed in paper A. Paper B presents a software framework designed to support future experimentation.

1.2.1 TCP/IP on a Limited Device

The TCP/IP protocol suite, which forms the basis of the Internet, is often perceived to be "heavy-weight" in that an implementation of the protocols requires large amounts of resources in terms of memory and processing power. This perception can be corroborated by measuring the memory requirements of popular TCP/IP implementations, such as the one in the Linux kernel [43] or in the BSD operating system [62]. The TCP/IP implementations in these systems require many hundreds of kilobytes of random access memory (RAM).

For most embedded systems, cost typically is a limiting factor. This constrains the available resources such as RAM and processor capabilities[1]. Consequently, many embedded systems do not have more than a few kilobytes of RAM. Within the constraints of such a small embedded system, it is impossible to run the TCP/IP implementations from Linux or BSD.

Within this thesis, I investigate the solution space to the problem of running TCP/IP within constrained memory limits. By developing a very small TCP/IP implementation that is able to run even on a system with very small amounts of memory, I demonstrate that the solution space of the problem is larger than previously shown. While this is not an exhaustive investigation of the solution

[1] Price and, hence, cost are not technical limitations, but functions of business models. It is therefore out of scope of this thesis to discuss these matters in any detail. For simplicity, I assume that cost is proportional to memory size and processor resources, but at the same time note that this is a gross oversimplification.

space, it does show that the solution space is large enough to accommodate even small embedded devices.

1.2.2 Operating Systems for Wireless Sensor Networks

The resource limitations and application characteristics of wireless sensor networks place specific requirements on the operating systems running on the sensor nodes. The applications are typically event-based: the application performs most of its work in response to external events. Resources are typically severely limited: memory is on the order of a few kilobytes, processing speed on the order of a few MHz, and limited energy from a battery or some other non-renewable energy source.

Early research into operating systems for sensor networks [44] identified the requirements and proposed a system, called TinyOS, that solved many of the problems. The TinyOS designers did, however, decide to leave out a set of features commonly found in larger operating systems, such as multithreading and run-time module loading.

In this thesis, I argue that multithreading and run-time loading of modules are desirable features of an operating system for sensor network nodes. I have implemented an operating system that includes said features and runs within the resource limitations of a sensor node, and thereby show that these features are feasible for sensor node operating systems.

1.2.3 Connecting Sensor Networks and IP Networks

A number of practical problems manifest themselves when doing a real-world deployment of a wireless sensor network. One of these is how to get data into and out of the sensor network, which may be deployed in a remote location. One way to solve this problem is to connect the sensor networks to an existing network infrastructure as an access network to the sensor network. Today most network infrastructures, including the global Internet, use the Internet Protocol (IP) [67] as its base technology. It is therefore interesting to investigate how wireless sensor networks can be connected to IP network infrastructures.

From the engineering perspective, the problem of connecting a sensor network with an IP network can easily be solved. In many cases, it is possible to simply place a PC inside or on the border of the sensor network and connect the PC both to the IP network and to the sensor network. The PC then acts as the gateway between the sensor network and the IP network. There are also many other possibilities, such as using a special-purpose device that connects the two

networks [17], or using satellite access to a special base station connected to the sensor network [59].

From the research perspective, however, the problem still has opportunities for investigation. Paper C is a first step towards characterizing the solution space. It presents three different types of solutions to the problem: proxy architectures, overlay networking, and direct connection by using TCP/IP in the sensor network. This thesis focuses on the last solution: connecting sensor networks and IP networks by using the TCP/IP protocols inside the sensor network as in the outside IP network.

1.2.4 TCP/IP for Wireless Sensor Networks

From the research perspective, investigating the use of TCP/IP in wireless sensor networks is of importance because the intersection of the TCP/IP protocol suite, the dominating communication protocol suite today, and wireless sensor networks, a new area in computer networking research, has not been previously studied. In general, the purpose of research is to provide understanding of problems and to gain new knowledge. Within this particular problem, we can develop new understanding of the interactions between wireless sensor networks and wired network infrastructures by identifying, solving, and studying the problems with TCP/IP in sensor networks.

From the engineering perspective, however, using the TCP/IP protocol suite inside the wireless sensor network may not be the "best" approach to solving the problem of connecting wireless sensor networks to IP networks, for some arbitrary definition of "best". There may be many other solutions to the problem that perform better both in a quantitative sense, e.g. that provide higher throughput or better energy efficiency, and in a qualitative sense, e.g. that provide a better security architecture. Prior to this thesis, however, no research has—to the best of my knowledge—been carried out to support claims in either way.

There are a number of problems with TCP/IP for wireless sensor networks. An enumeration of the problems, which are identified in paper D, follows.

IP Addressing Architecture

In ordinary IP networks, each network interface attached to a network is given its own unique IP address. The addresses are assigned either statically by human configuration, or dynamically using mechanisms such as DHCP [28]. This does not fit well with the sensor network paradigm. For sensor networks, the

addresses of the individual sensors are not interesting as such. Rather, the data generated by the sensors is the main interest. It is therefore advantageous to be able to loosen the requirement that each sensor has a unique address.

Address Centric Routing

Packet routing in IP networks is *address centric*, i.e., based on the addresses of the hosts and networks. The application specific nature of sensor networks makes *data centric* routing mechanisms [54] preferable. Data centric routing uses node attributes and the data contained in the packets to route packets towards a destination. Additionally, data centric mechanisms are naturally adopted to in-network data fusion [42].

Header Overhead

The protocols in the TCP/IP suite have a high overhead in terms of protocol header size, particularly for small packets. For small data packets, the header overhead is over 95%. Since energy conservation is of prime importance in sensor networks, transmission of unnecessary or redundant packet header fields should be avoided.

TCP Performance and Energy Efficiency

The reliable byte-stream protocol TCP has serious performance problems in wireless networks, both in terms of throughput [7] and in terms of energy efficiency. To be able to use TCP as a reliable transport protocol in wireless sensor networks, methods must be developed to increase the performance of TCP in the specific setting of sensor networks.

The end-to-end acknowledgment and retransmission scheme employed by TCP is not energy efficient enough to be useful in wireless sensor networks. A single dropped packet requires an expensive retransmission from the original source. Because sensor networks often are designed to be multi-hop, a single retransmission will incur transmission and reception costs at every hop through which the retransmitted packet will travel.

Limited Nodes

Sensor nodes are typically limited in terms of memory size and processing power. Any algorithm developed for sensor networks must therefore take these limitations into consideration.

Chapter 2

Contributions and Results

The main scientific contributions of this thesis are:

- The design and implementation of the uIP and the lwIP TCP/IP stacks that demonstrate that TCP/IP can be implemented on systems with very limited memory resources, without sacrificing interoperability or compliance.
- The formulation of initial solutions to the problems with TCP/IP for sensor networks, which point towards the feasibility of using TCP/IP for wireless sensor networks. This opens up opportunities for new research.
- The design and implementation of the Contiki operating system that has a number of features currently not found in other operating systems for the same class of hardware platforms. These features enable rapid experimentation for further research into the area of this thesis.

The work presented in this thesis has had a visible impact on networking for embedded systems and, to a lesser degree, on sensor networks. Less than a year after paper D was published, the 6lowpan IETF workgroup [63] was established. The focus of the workgroup is on standardizing transmission of IP packets over IEEE 802.15.4 [40], a sensor networking radio technology. The workgroup charter explicitly cites paper D and the uIP stack presented in paper A.

The work in this thesis is mentioned in books on embedded systems and networking [53, 61] and cited in numerous academic papers (e.g. [3, 11, 14, 29, 32, 34, 41, 51, 60, 64, 65, 69, 77]). Articles in professional magazines

have been written—by others—on using the uIP software for wireless sensor networks [8]. The software has been used in academic projects [52, 82], in courses e.g. at University of California, Los Angeles (UCLA) [74] and Stanford University [39], as well as in laboratory exercises [18, 79]. Finally, the software is being used in embedded operating systems [1, 81], and in a large number of embedded products (e.g. [12, 20, 25, 27, 33, 36, 47, 48, 75]).

Chapter 3

Summary of the Papers and Their Contributions

This thesis is a collection of four papers which all have been published at peer-reviewed international conferences. The first two papers, A and B, describe the necessary software platform for running the TCP/IP protocol suite in wireless sensor networks, whereas papers C and D focus on the protocol aspects of running TCP/IP inside a wireless sensor network.

3.1 Paper A: Full TCP/IP for 8-Bit Architectures

Adam Dunkels. Full TCP/IP for 8-bit architectures. In *Proceedings of The First International Conference on Mobile Systems, Applications, and Services (MOBISYS '03)*, May 2003.

Summary. The TCP/IP protocol suite is the family of protocols used for communication over the global Internet, and is often used in private networks such as local-area networks and corporate intranets. In order to attach a device to the network, the device must be able to use the TCP/IP protocols for communication.

This paper presents two small implementations of the TCP/IP protocol stack with slightly different designs; *lwIP*, which is designed in a modular and generic fashion, similar to how large-scale protocol implementations are designed, and *uIP* which is designed in a minimalistic fashion and only containing the absolute minimum set of features required to fulfill the protocol

standards. In order to reduce the code size and the memory requirements, the uIP implementation uses an event-based API which is fundamentally different from the most common TCP/IP API, the BSD socket API.

As was expected, measurements from an actual system running the implementations show that the smaller uIP implementation provides a very low throughput, particularly when sending data to a PC host. It must be noted, however, that small systems running uIP usually do not produce enough data for the performance degradation to become a serious problem.

Contribution. The main contribution of this paper is that it refutes the common conception that the TCP/IP protocol suite is too "heavy-weight" to be possible to fully implement on tiny devices. At the time this paper was written, most TCP/IP protocol stack implementations were designed for workstations and server-class systems, where communication performance was the primary concern. This caused a wide-spread belief that tiny devices such as sensor network nodes would be too constrained to be able to fully implement the TCP/IP stack. There were also a number of commercial implementations of the TCP/IP stack intended for embedded devices, where the protocols in the TCP/IP suite had been modified in order to reduce the code size and memory usage of their implementation. Such implementations are problematic as they may cause interoperability problems with other TCP/IP implementations. This paper shows that it is possible to implement the TCP/IP stack in a very small code size and with a very small memory usage, without modifying the protocols.

There is also a strong contribution made by the artifacts, the two TCP/IP implementations described in the paper. Both implementations have become wide-spread in academia as well as in the industry and are currently used in academic courses as well as in numerous embedded devices.

3.2 Paper B: Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors

Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors. In *First IEEE Workshop on Embedded Networked Sensors*, November 2004.

Summary. When performing experiments with sensor networks that are larger than a few nodes, having the ability to reprogram the network using the radio significantly reduces the development time. This paper presents Contiki, a lightweight and flexible operating system for tiny networked embedded

devices. Contiki has the ability to selectively load and unload individual programs while still being small enough to be usable on small sensor nodes.

Contiki supports two kinds of concurrency mechanisms: an event-driven interface and a preemptive multi-threading interface. The advantages of an event-driven model is that it is possible to implement using very small amounts of memory. Preemptive multi-threading, on the other hand, requires comparatively large amounts of memory to hold per-thread stacks. Furthermore, there are types of programs that are unsuited for the event-driven model but work well with preemptive multi-threading. Computationally intensive programs such as encryption algorithms are typical examples of this.

Unlike other operating systems, Contiki leverages both models by basing the system on an event-driven kernel and implementing preemptive multi-threading as an optional application library. This allows using preemptive multi-threading on a per-program basis. Experiments show that a Contiki system is able to continue to respond to events in a timely manner while performing a long-running computation as a preemptible thread.

Contribution. The main contribution of this paper is that it shows that preemptive multi-threading can be provided in an otherwise event-driven system. This leads the way to implementing more complex algorithms such as encryption mechanisms even in small sensor systems.

My contribution. I designed and implemented the system and wrote most of the text for the paper. The idea of providing preemptive multi-threading as an application library on top of an event-driven system was formed in cooperation with Björn Grönvall.

3.3 Paper C: Connecting Wireless Sensornets with TCP/IP Networks

Adam Dunkels, Thiemo Voigt, Juan Alonso, Hartmut Ritter, and Jochen Schiller. Connecting Wireless Sensornets with TCP/IP Networks. In *Proceedings of the Second International Conference on Wired/Wireless Internet Communications (WWIC2004)*, Frankfurt (Oder), Germany, February 2004.

Summary. Many sensor network applications require the sensor network to be connected to an external networks. Since TCP/IP has become the de-facto standard for networking, this paper focuses on the specific problem of connecting sensor networks to TCP/IP networks. We discuss three fundamentally different methods for connecting sensornets to TCP/IP networks: proxy architectures, Delay Tolerant Networking overlays, and directly using the TCP/IP

protocol suite in the sensor network. The paper concludes that the three methods are in some ways orthogonal and that combinations of the methods are possible.

Contribution. To the best of our knowledge, this paper is the first that explicitly discusses the issues of connecting sensor networks to TCP/IP networks. The contribution of the paper is the identification and characterization of the problem areas. The paper does not contain any simulation results or measurements, but focuses on the discussion of advantages and drawbacks of each of the presented methods.

My contribution. I wrote most of the text for the paper. The classification of proxies into front-end and relay proxies were done by me. The ideas and thoughts about TCP/IP for sensor networks and the comparison between the three different methods are mine.

3.4 Paper D: Making TCP/IP Viable for Wireless Sensor Networks

Adam Dunkels, Thiemo Voigt, and Juan Alonso. Making TCP/IP Viable for Sensor Networks. In *Proceedings of the First European Workshop on Wireless Sensor Networks (EWSN2004), work-in-progress session*, January 2004.

Summary. This paper addresses the specific problems of making TCP/IP a viable protocol stack for wireless networks of resource constrained sensors. The paper identifies five problem areas for which solutions are proposed: IP address assignment, TCP/IP header overhead, address centric routing, node limitations, and TCP performance and energy efficiency. The proposed solutions are: a *spatial IP address assignment* mechanism which lets sensor nodes construct semi-unique addresses from their spatial location; *shared context header compression*, that takes advantage of the shared context nature of sensor networks; *application overlay routing*, which enables implementation of data centric routing and data aggregation as application layer mechanisms; and a *distributed TCP caching* mechanism for improving TCP performance and energy efficiency.

Preliminary results indicate large energy savings compared to unoptimized TCP/IP.

Contribution. This paper is the first to address the challenges with TCP/IP for wireless sensor networks. It introduces the idea of using the TCP/IP protocol stack in wireless sensor networks, despite the specialized and resource constrained communication conditions. The contribution of this paper is that it

for the first time tries to provide a set of optimizations that enables the use of TCP/IP for wireless sensor networks.

My contribution. I formulated the idea of using TCP/IP for wireless sensor networks and worked out the ideas of spatial IP address assignment, shared context header compression, and application overlay routing. The idea of distributed TCP caching was conceived by Thiemo Voigt, and further refined by Thiemo and me in close cooperation. I wrote most of the text for the paper.

Chapter 4

Related Work

This chapter presents related work. The discussion is divided into four sections: small TCP/IP implementations, operating systems for sensor networks, connecting IP networks with sensor networks, and TCP/IP for sensor networks.

4.1 Small TCP/IP Implementations

There are several small TCP/IP implementations that fit the limitations of small embedded systems. Many of those implementations does, however, refrain from implementing certain protocol mechanisms in order to reduce the complexity of the implementation. The resulting implementation may therefore not be fully compatible with other TCP/IP implementations. Hence, communication may not be possible.

Many small TCP/IP implementations are tailored for a specific application, such as running a web server. This makes it possible to significantly reduce the implementation complexity, but does not provide a general communications mechanism that can be used for other applications. The PICmicro stack [10] is an example of such a TCP/IP implementation. Unlike such implementations, the uIP and lwIP implementations are not designed for a specific application.

Other implementations rely on the assumption that the small embedded device always will be communicating with a full-scale TCP/IP implementation running on a PC or work-station class device. Under this assumption it is possible to remove certain mechanisms that are required for full compatibility. Specifically, support for IP fragment reassembly and for TCP segment

size variation are two mechanisms that often are left out. Examples of such implementations are Texas Instrument's MSP430 TCP/IP stack [24] and the TinyTCP code [19]. Neither the uIP or the lwIP stack are designed under this assumption.

In addition to the TCP/IP implementation for small embedded systems, there is a large class of TCP/IP implementations for embedded systems with less constraining limitations. Typically, such implementations are based on the TCP/IP implementation from the BSD operating system [62]. These implementations do not suffer from the same problems as the tailored implementations. Such implementations does, however, in general require too large amount of resources to be feasible for small embedded systems. Typically, such implementations are orders of magnitude larger than the uIP implementation.

4.2 Operating Systems for Sensor Networks

TinyOS [44] is probably the earliest operating system that directly targets the specific applications and limitations of sensor devices. TinyOS is built around a lightweight event scheduler where all program execution is performed in tasks that run to completion. TinyOS uses a special description language for composing a system of smaller components [37] which are statically linked with the kernel to a complete image of the system. After linking, modifying the system is not possible [55]. The Contiki system is also designed around a lightweight event-scheduler, but is designed to allow loading, unloading, and replacing modules at run-time.

In order to provide run-time reprogramming for TinyOS, Levis and Culler have developed Maté [55], a virtual machine for TinyOS devices. Code for the virtual machine can be downloaded into the system at run-time. The virtual machine is specifically designed for the needs of typical sensor network applications. Similarly, the MagnetOS [9] system uses a virtual Java machine to distribute applications across the sensor network. The advantages of using a virtual machine instead of native machine code is that the virtual machine code can be made smaller, thus reducing the energy consumption of transporting the code over the network. One of the drawbacks is the increased energy spent in interpreting the code—for long running programs the energy saved during the transport of the binary code is instead spent in the overhead of executing the code. Contiki does not suffer from the executional overhead as modules loaded into Contiki are compiled to native machine code.

SensorWare [13] provides an abstract scripting language for programming sensors, but their target platforms are not as resource constrained as ours. Similarly, the EmStar environment [38] is designed for less resource constrained systems. Reijers and Langendoen [71] use a patch language to modify parts of the binary image of a running system. This works well for networks where all nodes run the exact same binary code but soon gets complicated if sensors run slightly different programs or different versions of the same software.

The Mantis system [2] uses a traditional preemptive multi-threaded model of operation. Mantis enables reprogramming of both the entire operating system and parts of the program memory by downloading a program image onto EEPROM, from where it can be burned into flash ROM. Due to the multi-threaded semantics, every Mantis program must have stack space allocated from the system heap, and locking mechanisms must be used to achieve mutual exclusion of shared variables. In Contiki, only such programs that explicitly require multi-threading needs to allocate an extra stack.

4.3 Connecting IP Networks with Sensor Networks

At the time of publication of paper C, there was very little work done in the area of connecting wireless sensor networks and IP networks. Recently, however, a number of papers on the subject has been published.

Ho and Fall [45] have presented an application of Delay Tolerant Networking (DTN) mechanisms to sensor networks. Their work is similar to that presented in paper C, but is more focused on the specifics of the DTN architecture.

The overlay architecture presented by Dai and Han [23] unifies the Internet and sensor networks by providing a sensor network overlay layer on top of the Internet. While this work is similar in scope to the work in this thesis, it explores a slightly different path: this thesis explores the interconnectivity in a lower layer of the protocol stack.

The FLexible Interconnection Protocol (FLIP) [78] provides interconnectivity between IP networks and sensor networks, but relies on protocol converters at the border of the sensor network. This thesis investigates an architecture where no explicit protocol converters are required.

Finally, the Plutarch architecture [22] changes the communication architecture of the Internet in a way that is able to accommodate natural inclusion of sensor networks in the new communication architecture. This work is orthogonal to the work in this thesis. The intention with this thesis is to investigate how sensor networks can be connected with today's IP network infrastructures.

4.4 TCP/IP for Wireless Sensor Networks

While I am not aware of any previous work on TCP/IP for wireless sensor networks, the area of mobile ad-hoc networks (MANETs) is the area which is most closely related to the area of TCP/IP for wireless sensor networks. MANETs typically use the TCP/IP protocol suite for communication both within the MANETs and with outside networks. There are, however, a number of differences between sensor networks and MANETs that affect the applicability of TCP/IP. First, MANET nodes typically has significantly more resources in terms of memory and processing power than sensor network nodes. Furthermore, MANET nodes are operated by human users, whereas sensor networks are intended to be autonomous. The user-centricity of MANETs makes throughput the primary performance metric, while the per-node throughput in sensor networks is inherently low because of the limited capabilities of the nodes. Instead, energy consumption is the primary concern in sensor networks. Finally, TCP throughput is reduced by mobility [46], but nodes in sensor networks are usually not as mobile as MANET nodes.

While the specific area of TCP/IP for wireless sensor networks has not been previously explored, there are a number of adjacent areas that are relevant to this licentiate thesis. The following sections presents the related work in those areas.

4.4.1 Reliable Sensor Network Transport Protocols

Reliable data transmission in sensor networks have attained very little research attention, mostly because many sensor network applications do not require reliable data transmission. Nevertheless, a few protocols for reliable data transport have been developed. Those protocols target both the problem of reliable transmission of sensor data from sensors to a "sink" node, and the problem of reliable transmission of data from a central sink node to a sensor. Potential uses of reliable data transmission is transport of important sensor data from one or more sensors to a sink node, transmission of sensor node configuration from a central server to one or more sensors, program downloads to sensor nodes, and other administrative tasks. Most protocols for reliable transport in sensor networks are designed specifically for sensor networks and therefore cannot be readily used for e.g. downloading data from an external IP network, without protocol converters or proxy servers.

Reliable Multi-Segment Transport (RMST) [80] provides a reliable transport protocol for bounded messages on top of the Distributed Diffusion routing

paradigm [49]. RMST uses either hop-by-hop reliability through negative acknowledgments and local retransmissions, or end-to-end reliability by using positive acknowledgments and end-to-end retransmissions. The authors provide simulation results and conclude that reliable transport for sensor networks is best implemented on the MAC layer. The results provided rely on the fact that the Directed Diffusion routing substrate is able to find relatively good paths through the network, however.

Pump Slowly Fetch Quickly (PSFQ) [83] is a reliable transport protocol that focuses on one-to-many communication situations and uses hop-by-hop reliability. In PSFQ, data is slowly *pumped* towards the receivers, one fragment at a time. If a nodes along the path towards the receiver notices that a data fragment has been lost, it issues a *fetch* request to the closest node on the backward path. The number of fetch requests for a single fragment is bounded and fetch requests are issued only within the time frame between two data fragments are pumped.

Event-to-Sink Reliable Transport (ESRT) [73] is a transport protocol that provides a semi-reliable transport in only one direction. Data that is sent from sensors to a sink is given a certain amount of reliability. The sink node, which is assumed to have more computational resources than the sensors, computes a suitable reporting frequency for the nodes.

4.4.2 Header Compression

Header compression is a technique that reduces packet header overhead by refraining from transmitting header fields that do not change between consecutive packets. The header compressor and the decompressor shares the state of streams that pass over them. This shared state is called the header compression *context*. The compression works by not transmitting full headers, but only the delta values for such header fields that change in a predictable way. Early variants of header compression for TCP were developed for low speed serial links [50] and are able to compress most headers down to only 10% of their original size.

Early header compression schemes did not work well over lossy links since they could not recover from the loss of a header update. A missed header update will cause subsequent header updates to be incorrect because of the context mismatch between the compressor and the decompressor. The early methods did not try to detect incorrectly decompressed headers. Rather, these methods trusted recipients to drop packets with erroneous headers and relied on retransmissions from the sender to repair the context mismatch.

Degermark et al. [26, 57] have presented a method for compressing headers for both TCP/IP and for a set of real-time data protocols. The method is robust in the sense that it is able to recover from a context mismatch by using feedback from the header decompressor. The feedback information is piggybacked on control packets such as acknowledgments that travel on the reverse-path. Furthermore, authors introduces the TWICE algorithm. The algorithm is able to adapt to a single lost header delta value by applying the received delta value twice. Incorrectly decompressed headers are identified by computing the checksum of the decompressed packet. If the checksum is found to be incorrect by the decompressor, a full header is requested from the compressor, thus synchronizing the header compression context.

Sridharan et al. [70] have presented Routing-Assisted Header Compression (RAHC), a header compression scheme that is particularly well-suited for multi-hop networks. Unlike other header compression schemes, the RAHC algorithm works end-to-end across a number of routing hops. The algorithm utilizes information from the underlying routing protocol in order to detect route changes and multiple paths.

4.4.3 TCP over Wireless Media

TCP [68] was designed for wired networks where congestion is the predominant source of packet drops. TCP reduces its sending rate detecting packet loss in order to avoid overloading the network. This behavior has shown to be problematic when running TCP over wireless links that have potentially high bit error rates. Packet loss due to bit errors will be interpreted by TCP as a sign of congestion and TCP will reduce its sending rate. TCP connections running over wireless links may therefore see very large reductions in throughput. A number of mechanisms for solving these problems have been studied.

Wireless TCP enhancements can be divided into three types [6]: *split-connection*, *end-to-end*, and *link-layer*. The split-connection approach, as exemplified by Indirect TCP [5] and M-TCP [16], splits each TCP connections into two parts: one over the wired network and one over the wireless link. Connections are terminated at a base station to allow a specially tuned protocol to be used between the base station and the wireless host.

TCP snoop [7] is a link-layer approach that is designed to work in a scenario where the last hop is over a wireless medium. TCP snoop uses a program called the *snoop agent* that is running on the base station before the last hop. The snoop agent intercepts TCP segments and caches them. If it detects a failed transmission, it will immediately retransmit the lost segment and pre-

vent duplicate acknowledgements to be sent towards the original sender of the segment.

A-TCP [56] is primarily designed for wireless ad-hoc networks and is an example of the end-to-end approach. A-TCP inserts a conceptual layer inbetween IP and TCP that deals with packet losses because of transmission errors and unstable routes. Unlike the other approaches, A-TCP requires modifications to the end-host.

4.4.4 Addressing in Sensor Networks

Addressing in sensor networks is different from addressing in other computer networks in that the sensors do not necessarily need to have individual addresses [42]. Instead, many sensor network applications benefit from seeing the *data* sensed by the network the primary addressing object [35]. This allows routing to be *data-centric* rather than the traditional address-centric. One of the earliest data-centric routing protocols is Directed Diffusion [49] which propagates an information interest through the network. When a sensor obtains information for which an interest has been registered, it transmits the information back towards the source of the information interest.

A different approach is taken by TinyDB [58] where the sensor network is viewed as a distributed data base. The data base is queried with an SQL-like language. Query strings are processed by a base station, and compressed and optimized queries are disseminated through the sensor network. Results are distributed back through the routing tree that was formed when the query was propagated. This is an addressing scheme where the data is explicitly addressed and where individual nodes are not possible to address directly.

Chapter 5

Conclusions and Future Work

This licentiate thesis takes the first steps towards the use of the TCP/IP protocol suite in wireless sensor networks. It builds the framework in which the use of TCP/IP can be further investigated, identifies the problems with TCP/IP for sensor networks, and formulates initial solutions to the problems. The contribution of this work is that it for the first time brings TCP/IP, the dominant protocol stack, together with wireless sensor networks.

The results of the work presented in this thesis have had a significant impact on the embedded TCP/IP networking community. The software developed as part of the thesis has become widely known in the community. The software is used in academic research projects, academic courses, as well as a large number of embedded devices.

I will continue this work with experimental studies of the use of TCP/IP in wireless sensor networks. Further investigation must be made before the hypothesis that TCP/IP is a viable protocol suite for wireless sensor networks can be validated or invalidated. We have already made simulation studies of the Distributed TCP Caching mechanism [30] and are designing a MAC layer that will support DTC. We intend to evaluate the energy efficiency of TCP/IP for sensor networks by using the method described by Ritter et al. [72]. While this method has been developed to experimentally evaluate a model of life-time bounds [4], it also is useful for comparing the energy efficiency of communication protocols.

I will also continue to investigate software construction for memory con-

strained systems, based on the findings in papers A and B. This work consists of developing mechanisms and methods for implementing computer programs for resource limited embedded systems and sensor nodes. I am currently working on a lightweight mechanism called *protothreads* that provides sequential flow of control for event-driven systems.

Bibliography

[1] eCos Embedded Configurable Operating System. Web page. URL: http://sources.redhat.com/ecos/

[2] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, and R. Han. Mantis: system support for multimodal networks of in-situ sensors. In *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, pages 50–59, 2003.

[3] P. Agrawal, T.S. Teck, and A.L. Ananda. A lightweight protocol for wireless sensor networks. March 2003.

[4] J. Alonso, A. Dunkels, and T. Voigt. Bounds on the energy consumption of routings in wireless sensor networks. In *Proceedings of the 2nd WiOpt, Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks*, Cambridge, UK, March 2004.

[5] A. Bakre and B. R. Badrinath. I-TCP: Indirect TCP for mobile hosts. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, May 1995.

[6] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, and R. H. Katz. A comparison of mechanisms for improving TCP performance over wireless links. *IEEE/ACM Trans. Netw.*, 5(6):756–769, 1997.

[7] H. Balakrishnan, S. Seshan, E. Amir, and R. Katz. Improving TCP/IP performance over wireless networks. In *Proceedings of the first ACM Conference on Mobile Communications and Networking*, Berkeley, California, November 1995.

[8] D. Barnett and A. J. Massa. Inside the uIP Stack. *Dr. Dobb's Journal*, February 2005.

[9] R. Barr, J. C. Bicket, D. S. Dantas, B. Du, T. W. D. Kim, B. Zhou, and E. Sirer. On the need for system-level support for ad hoc and sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(2):1–5, 2002.

[10] J. Bentham. *TCP/IP Lean: Web servers for embedded systems*. CMP Books, October 2000.

[11] S. Beyer, K. Mayes, and B. Warboys. Application-compliant networking on embedded systems. In *Proceedings of the 5th IEEE Internation Workshop on Networked Appliances*, pages 53–58, October 2002.

[12] C. Borrelli. TCP/IP on Virtex-II Pro Devices Using IwIP. XAPP 663, Xilinx Inc., August 2003.

[13] A. Boulis, C. Han, and M. B. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. In *Proceedings of The First International Conference on Mobile Systems, Applications, and Services (MOBISYS '03)*, May 2003.

[14] M. Britton, V. Shum, L. Sacks, and H. Haddadi. A biologically-inspired approach to designing wireless sensor networks. In *Proceedings of the Second European Workshop on Wireless Sensor Networks*, Istanbul, Turkey, 2005.

[15] F. P. Brooks Jr. The computer scientist as a toolsmith II. *Communications of the ACM*, 39(3):61–68, March 1996.

[16] K. Brown and S. Singh. M-TCP: TCP for mobile cellular networks. *ACM Computer Communications Review*, 27(5):19–43, October 1997.

[17] P. Buonadonna, D. Gay, J. Hellerstein W. Hong, and S. Madden. TASK: Sensor Network in a Box. In *Proceedings of the Second European Workshop on Sensor Networks*, 2005.

[18] B. F. Cockburn. CMPE 401 - Computer Interfacing. Web page. URL: http://www.ece.ualberta.ca/~cmpe401/

[19] G. H. Cooper. TinyTCP. Web page. 2002-10-14. URL: http://www.csonline.net/bpaddock/tinytcp/

[20] Nu Horizons Electronics Corp. TCP/IP Development Kit. Web page. URL: http://www.nuhorizons.com/

[21] National Research Council. *Academic Careers for Experimental Computer Scientists and Engineers*. National Academy Press, 1994.

[22] J. Crowcroft, S. Hand, R. Mortier, T. Roscoe, and A. Warfield. Plutarch: an argument for network pluralism. In *FDNA '03: Proceedings of the ACM SIGCOMM workshop on Future directions in network architecture*, pages 258–266. ACM Press, 2003. ISBN: 1-58113-748-0

[23] H. Dai and R. Han. Unifying micro sensor networks with the internet via overlay networking. In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks (LCN'04)*, pages 571–572, November 2004.

[24] A. Dannenberg. MSP430 internet connectivity. SLAA 137, November 2001. Avalible from www.ti.com.

[25] Microtronix Datacom. MicroC/OS-II Development Kit. Web page. URL: http://www.microtronix.com/

[26] M. Degermark, M. Engan, B. Nordgren, and S. Pink. Low-loss TCP/IP header compression for wireless networks. *ACM/Baltzer Journal on Wireless Networks*, 3(5), 1997.

[27] Axon Digital Designs. Synapse: Modular Broadcast System. Web page. URL: http://www.axon.tv

[28] R. Droms. Dynamic Host Configuration Protocol. RFC 2131, Internet Engineering Task Force, March 1997.

[29] W. Drytkiewicz, I. Radusch, S. Arbanowski, and R. Popescu-Zeletin. pREST: A REST-based Protocol for Pervasive Systems. In *Proceedings of the 1st IEEE International Conference on Mobile Ad-hoc and Sensor Systems (MASS)*, Ft. Lauderdale, USA, October 2004.

[30] A. Dunkels, T. Voigt, J. Alonso, and H. Ritter. Distributed TCP Caching for Wireless Sensor Networks. In *Proceedings of the Third Annual Mediterranean Ad Hoc Networking Workshop*, June 2004.

[31] A. Dunkels, T. Voigt, N. Bergman, and M. Jönsson. An IP-based sensor network as a rapidly deployable building security system. In *Swedish National Computer Networking Workshop*, Karlstad, Sweden, November 2004.

[32] K. Elphinstone and S. Götz. Initial evaluation of a user-level device driver framework. In *Proceedings of the Ninth Asia-Pacific Computer Systems Architecture Conference*, pages 131–136, Beijing, China, September 7–9 2004.

[33] emWare. emWare: Remote Device Management. Web page. URL: http://www.emware.com/

[34] M. Engel and B. Freisleben. A lightweight communication infrastructure for spontaneously networked devices with limited resources. In *Proceedings of the 2002 International Conference on Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 22–40, Erfurt, Germany, 2002.

[35] D. Estrin, R. Govindan, J. S. Heidemann, and S. Kumar. Next century challenges: Scalable coordination in sensor networks. In *Mobile Computing and Networking*, pages 263–270, 1999.

[36] Gavitech AG. VisionSpy 6300 User's manual. Available from http://www.gavitech.com/.

[37] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1–11, 2003.

[38] L. Girod, J. Elson, A. Cerpa, T. Stathopoulos, N. Ramanathan, and D. Estrin. EmStar: A Software Environment for Developing and Deploying Wireless Sensor Networks. In *Proceedings of the USENIX Annual Technical Conference*, 2004.

[39] Stanford High-Performance Networking Group. CS344: Building an Internet Router. Web page. Visited 2004-09-30. URL: http://yuba.stanford.edu/cs344_public/

[40] J. A. Gutierrez, M. Naeve, E. Callaway, M. Bourgeois, V. Mitter, and B. Heile. IEEE 802.15.4: A developing standard for low-power low-cost wireless personal area networks. *IEEE Network*, 15(5):12–19, September/October 2001.

[41] J. Hallberg, S. Svensson, A. Östmark, P. Lindgren, K. Synnes, and J. Delsing. Enriched media-experience of sport events. In *Sixth IEEE*

Workshop on Mobile Computing Systems and Applications (WMCSA'04), Lake District National Park, United Kingdom, December 2004.

[42] J. Heidemann, F. Silva, C. Intanagonwiwat, R. Govindan, D. Estrin, and D. Ganesan. Building efficient wireless sensor networks with low-level naming. In *Proceedings of the Symposium on Operating Systems Principles*, pages 146–159, Chateau Lake Louise, Banff, Alberta, Canada, October 2001. ACM.

[43] T. F. Herbert. *The Linux TCP/IP Stack: Networking For Embedded Systems*. Charles River Media, 2004.

[44] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[45] M. Ho and K. Fall. Delay tolerant networking for sensor networks. Poster, October 2004.

[46] G. Holland and N. H. Vaidya. Analysis of TCP performance over mobile ad hoc networks. In *MOBICOM '99*, August 1999.

[47] Analog Devices Inc. ADI Blackfin Software Development Kit. Web page. Visited 2005-02-17. URL: http://www.analog.com/

[48] Pumpkin Inc. CubeSat kit. Web page. Visited 2005-02-17. URL: http://www.cubesatkit.com/

[49] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *Mobile Computing and Networking*, pages 56–67, 2000.

[50] V. Jacobson. Compressing TCP/IP headers for low-speed serial links. RFC 1144, Internet Engineering Task Force, February 1990.

[51] A. Jamieson, S. Breslin, P. Nixon, and D. Smeed. Mipos - the mote indoor positioning system. In *1st International Workshop on Wearable and Implantable Body Sensor Networks*, April 2004.

[52] A. Jamieson, D. Smeed, and P. Nixon. Construction guidelines: Hardware infrastructure design. Technical Report D12, Global smart spaces IST-2000-2607, 2003.

[53] M. T. Jones. *TCP/IP Application Layer Protocols for Embedded Systems*. Charles River Media, June 2002.

[54] B. Krishnamachari, D. Estrin, and S. Wicker. The impact of data aggregation in wireless sensor networks. In *Proceedings of International Workshop on Distributed Event-Based Systems*, 2002.

[55] P. Levis and D. Culler. Mate: A tiny virtual machine for sensor networks. In *Proceedings of ASPLOS-X*, San Jose, CA, USA, October 2002.

[56] J. Liu and S. Singh. ATCP: TCP for mobile ad hoc networks. *IEEE Journal on Selected Areas in Communications*, 19(7):1300–1315, 2001.

[57] S. Pink M. Degermark, B. Nordgren. IP header compression. RFC 2507, Internet Engineering Task Force, February 1999.

[58] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 491–502. ACM Press, 2003. ISBN: 1-58113-634-X

[59] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless sensor networks for habitat monitoring. In *First ACM Workshop on Wireless Sensor Networks and Applications (WSNA 2002)*, Atlanta, GA, USA, September 2002.

[60] K. Mansley. Engineering a user-level tcp for the clan network. In *NICELI '03: Proceedings of the ACM SIGCOMM workshop on Network-I/O convergence*, pages 228–236. ACM Press, 2003. ISBN: 0-123-45678-0

[61] A. J. Massa. *Embedded Software Development with eCos*. Prentice Hall, November 2002.

[62] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison-Wesley, 1996.

[63] G. Mulligan, N. Kushalnagar, and G. Montenegro. IPv6 over IEEE 802.15.4 BOF (6lowplan). Web page. URL: http://www.ietf.org/ietf/04nov/6lowpan.txt

[64] V. K. Nandivada and J. Palsberg. Timing analysis of TCP servers for surviving denial-of-service attacks. In *Proceedings of RTAS'05, 11th IEEE Real-Time and Embedded Technology and Applications Symposium*, San Francisco, March 2005.

[65] C. Pearce, V. Yin Man Ma, and P. Bertok. A secure communication protocol for ad-hoc wireless sensor networks. In *IEEE International Conference on Intelligent Sensors, Sensor Networks and Information Processings*, December 2004.

[66] E. M. Phillips and D. S. Pugh. *How to get a PhD : a handbook for students and their supervisors*. Open University Press, Buckingham, England, 1994.

[67] J. Postel. Internet protocol. RFC 791, Internet Engineering Task Force, September 1981.

[68] J. Postel. Transmission control protocol. RFC 793, Internet Engineering Task Force, September 1981.

[69] S. Pérez, J. Vila, J. A. Alegre, and J. V. Sala. A CORBA Based Architecture for Distributed Embedded Systems Using the RTLinux-GPL Platform. In *Proceedings of the Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'04)*, Vienna, Austria, May 2004.

[70] S. Mishra R. Sridharan, R. Sridhar. A robust header compression technique for wireless ad hoc networks. In *MobiHoc 2003*, Annapolis, MD, USA, June 2003.

[71] N. Reijers and K. Langendoen. Efficient code distribution in wireless sensor networks. In *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, pages 60–67, 2003.

[72] H. Ritter, J. Schiller, T. Voigt, A. Dunkels, and J. Alonso. Experimental Evaluation of Lifetime Bounds for Wireless Sensor Networks. In *Proceedings of the Second European Workshop on Sensor Networks (EWSN2005)*, Istanbul, Turkey, January 2005.

[73] Y. Sankarasubramaniam, O. Akan, and I. Akyildiz. ESRT : Event-to-Sink Reliable Transport in Wireless Sensor Networks. In *Proceedings of the 4th ACM international symposium on Mobile ad hoc networking and computing (MobiHOC 2003)*, 2003.

[74] P. Schaumont. EE201A - VLSI Architectures and Design Methods. Web page. URL: http://www.ee.ucla.edu/~schaum/ee201a/

[75] GE Security. Intermodal Container Security CommerceGuard(TM) - Protecting and Tracking Global Trade. Web page. Visited 2005-02-17. URL: http://www.gesecurity.com/csd/

[76] S. N. Simic and S. Sastry. Distributed environmental monitoring using random sensor networks. In *Proceedings of the 2nd International Workshop on Information Processing in Sensor Networks*, pages 582–592, Palo Alto, California, 2003.

[77] A. Sinha, S. Sarat, and J. S. Shapiro. Network subsystems reloaded: A high-performance, defensible network subsystem. In *Proceedings of the 2004 USENIX Annual Technical Conference*, Boston, MA, USA, June 2004.

[78] I. Solis and K. Obraczka. Flip: a flexible interconnection protocol for heterogeneous internetworking. *Mob. Netw. Appl.*, 9(4):347–361, 2004. ISSN: 1383-469X

[79] M. Srivastava. EE202A, Embedded and Real-time Systems. Web page. URL: http://nesl.ee.ucla.edu/courses/ee202a/2003f/

[80] F. Stann and J. Heidemann. RMST: Reliable Data Transport in Sensor Networks. In *Proceedings of the First International Workshop on Sensor Net Protocols and Applications*, pages 102–112, Anchorage, Alaska, USA, April 2003. IEEE.

[81] J. T. Taylor. eXtreme Minimal Kernel. Shift-Right Technologies LLC. URL: http://www.shift-right.com/xmk/

[82] C. Tschudin, R. Gold, O. Rensfelt, and O. Wibling. LUNAR: a Lightweight Underlay Network Ad-hoc Routing Protocol and Implementation. In *Proceedings of The Next Generation Teletraffic and Wired/Wireless Advanced Networking (NEW2AN'04)*, 2004.

[83] C.Y. Wan, A. T. Campbell, and L. Krishnamurthy. PSFQ: A Reliable Transport Protocol For Wireless Sensor Networks. In *First ACM International Workshop on Wireless Sensor Networks and Applications (WSNA 2002)*, Atlanta, September 2002.

[84] M. Welsh, D. Myung, M. Gaynor, and S. Moulton. Resuscitation monitoring with a wireless sensor network. In *Circulation 108:1037: Journal of the American Heart Association, Resuscitation Science Symposium.*, October 2003.

[85] G. Werner-Allen, J. Johnson, M. Ruiz, J. Lees, and M. Welsh. Monitoring volcanic eruptions with a wireless sensor network. In *Proceedings of the Second European Workshop on Sensor Networks*, 2005.

II

Included Papers

Chapter 6

Paper A: Full TCP/IP for 8-Bit Architectures

Adam Dunkels. Full TCP/IP for 8-bit architectures. In *Proceedings of The First International Conference on Mobile Systems, Applications, and Services (MOBISYS '03)*, May 2003.

Abstract

We describe two small and portable TCP/IP implementations fulfilling the subset of RFC1122 requirements needed for full host-to-host interoperability. Our TCP/IP implementations do not sacrifice any of TCP's mechanisms such as urgent data or congestion control. They support IP fragment reassembly and the number of multiple simultaneous connections is limited only by the available RAM. Despite being small and simple, our implementations do not require their peers to have complex, full-size stacks, but can communicate with peers running a similarly light-weight stack. The code size is on the order of 10 kilobytes and RAM usage can be configured to be as low as a few hundred bytes.

6.1 Introduction

With the success of the Internet, the TCP/IP protocol suite has become a global standard for communication. TCP/IP is the underlying protocol used for web page transfers, e-mail transmissions, file transfers, and peer-to-peer networking over the Internet. For embedded systems, being able to run native TCP/IP makes it possible to connect the system directly to an intranet or even the global Internet. Embedded devices with full TCP/IP support will be first-class network citizens, thus being able to fully communicate with other hosts in the network.

Traditional TCP/IP implementations have required far too much resources both in terms of code size and memory usage to be useful in small 8 or 16-bit systems. Code size of a few hundred kilobytes and RAM requirements of several hundreds of kilobytes have made it impossible to fit the full TCP/IP stack into systems with a few tens of kilobytes of RAM and room for less than 100 kilobytes of code.

TCP [21] is both the most complex and the most widely used of the transport protocols in the TCP/IP stack. TCP provides reliable full-duplex byte stream transmission on top of the best-effort IP [20] layer. Because IP may reorder or drop packets between the sender and the receiver, TCP has to implement sequence numbering and retransmissions in order to achieve reliable, ordered data transfer.

We have implemented two small generic and portable TCP/IP implementations, *lwIP* (lightweight IP) and *uIP* (micro IP), with slightly different design goals. The lwIP implementation is a full-scale but simplified TCP/IP implementation that includes implementations of IP, ICMP, UDP and TCP and is modular enough to be easily extended with additional protocols. lwIP has support for multiple local network interfaces and has flexible configuration options which makes it suitable for a wide variety of devices.

The uIP implementation is designed to have only the absolute minimal set of features needed for a full TCP/IP stack. It can only handle a single network interface and does not implement UDP, but focuses on the IP, ICMP and TCP protocols.

Both implementations are fully written in the C programming language. We have made the source code available for both lwIP [7] and uIP [8]. Our implementations have been ported to numerous 8- and 16-bit platforms such as the AVR, H8S/300, 8051, Z80, ARM, M16c, and the x86 CPUs. Devices running our implementations have been used in numerous places throughout the Internet.

We have studied how the code size and RAM usage of a TCP/IP implementation affect the features of the TCP/IP implementation and the performance of the communication. We have limited our work to studying the implementation of TCP and IP protocols and the interaction between the TCP/IP stack and the application programs. Aspects such as address configuration, security, and energy consumption are out of the scope of this work.

The main contribution of our work is that we have shown that is it possible to implement a full TCP/IP stack that is small enough in terms of code size and memory usage to be useful even in limited 8-bit systems.

Recently, other small implementations of the TCP/IP stack have made it possible to run TCP/IP in small 8-bit systems. Those implementations are often heavily specialized for a particular application, usually an embedded web server, and are not suited for handling generic TCP/IP protocols. Future embedded networking applications such as peer-to-peer networking require that the embedded devices are able to act as first-class network citizens and run a TCP/IP implementation that is not tailored for any specific application.

Furthermore, existing TCP/IP implementations for small systems assume that the embedded device always will communicate with a full-scale TCP/IP implementation running on a workstation-class machine. Under this assumption, it is possible to remove certain TCP/IP mechanisms that are very rarely used in such situations. Many of those mechanisms are essential, however, if the embedded device is to communicate with another equally limited device, e.g., when running distributed peer-to-peer services and protocols.

This paper is organized as follows. After a short introduction to TCP/IP in Section 6.2, related work is presented in Section 6.3. Section 6.4 discusses RFC standards compliance. How memory and buffer management is done in our implementations is presented in Section 6.5 and the application program interface is discussed in Section 6.6. Details of the protocol implementations is given in Section 6.7 and Section 6.8 comments on the performance and maximum throughput of our implementations, presents throughput measurements from experiments and reports on the code size of our implementations. Section 6.9 gives ideas for future work. Finally, the paper is summarized and concluded in Section 6.10.

6.2 TCP/IP overview

From a high level viewpoint, the TCP/IP stack can be seen as a black box that takes incoming packets, and demultiplexes them between the currently active

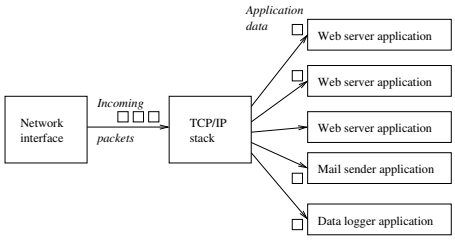

Figure 6.1: TCP/IP input processing.

connections. Before the data is delivered to the application, TCP sorts the packets so that they appear in the order they were sent. The TCP/IP stack will also send acknowledgments for the received packets.

Figure 6.1 shows how packets come from the network device, pass through the TCP/IP stack, and are delivered to the actual applications. In this example there are five active connections, three that are handled by a web server application, one that is handled by the e-mail sender application, and one that is handled by a data logger application.

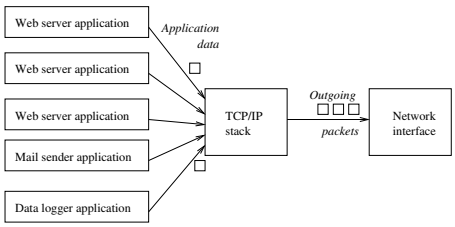

Figure 6.2: TCP/IP output processing.

A high level view of the output processing can be seen in Figure 6.2. The TCP/IP stack collects the data sent by the applications before it is actually sent onto the network. TCP has mechanisms for limiting the amount of data that is sent over the network, and each connection has a queue on which the data is held while waiting to be transmitted. The data is not removed from the queue until the receiver has acknowledged the reception of the data. If no acknowledgment is received within a specific time, the data is retransmitted.

Data arrives asynchronously from both the network and the application,

and the TCP/IP stack maintains queues in which packets are kept waiting for service. Because packets might be dropped or reordered by the network, incoming packets may arrive out of order. Such packets have to be queued by the TCP/IP stack until a packet that fills the gap arrives. Furthermore, because TCP limits the rate at which data that can be transmitted over each TCP connection, application data might not be immediately sent out onto the network.

The full TCP/IP suite consists of numerous protocols, ranging from low level protocols such as ARP which translates IP addresses to MAC addresses, to application level protocols such as SMTP that is used to transfer e-mail. We have concentrated our work on the TCP and IP protocols and will refer to upper layer protocols as "the application". Lower layer protocols are often implemented in hardware or firmware and will be referred to as "the network device" that are controlled by the network device driver.

TCP provides a reliable byte stream to the upper layer protocols. It breaks the byte stream into appropriately sized segments and each segment is sent in its own IP packet. The IP packets are sent out on the network by the network device driver. If the destination is not on the physically connected network, the IP packet is forwarded onto another network by a router that is situated between the two networks. If the maximum packet size of the other network is smaller than the size of the IP packet, the packet is fragmented into smaller packets by the router. If possible, the size of the TCP segments are chosen so that fragmentation is minimized. The final recipient of the packet will have to reassemble any fragmented IP packets before they can be passed to higher layers.

6.3 Related work

There are numerous small TCP/IP implementations for embedded systems. The target architectures range from small 8-bit microcontrollers to 32-bit RISC architectures. Code size varies from a few kilobytes to hundreds of kilobytes. RAM requirements can be as low as 10 bytes up to several megabytes.

Existing TCP/IP implementations can roughly be divided into two categories; those that are adaptations of the Berkeley BSD TCP/IP implementation [18], and those that are written independently from the BSD code. The BSD implementation was originally written for workstation-class machines and was not designed for the limitations of small embedded systems. Because of that, implementations that are derived from the BSD code base are usually suited for larger architectures than our target. An example of a BSD-

derived implementation is the InterNiche NicheStack [11], which needs around 50 kilobytes of code space on a 32-bit ARM system.

Many of the independent TCP/IP implementations for embedded processors use a simplified model of the TCP/IP stack which makes several assumptions about the communication environment. The most common assumption is that the embedded system always will communicate with a system such as a PC that runs a full scale, standards compliant TCP/IP implementation. By relying on the standards compliance of the remote host, even an extremely simplified, uncompliant, TCP/IP implementation will be able to communicate. The communication may very well fail, however, once the system is to communicate with another simplified TCP/IP implementation such as another embedded system of the same kind. We will briefly cover a number of such simplifications that are used by existing implementations.

One usual simplification is to tailor the TCP/IP stack for a specific application such as a web server. By doing this, only the parts of the TCP/IP protocols that are required by the application need to be implemented. For instance, a web server application does not need support for urgent data and does not need to actively open TCP connections to other hosts. By removing those mechanisms from the implementation, the complexity is reduced.

The smallest TCP/IP implementations in terms of RAM and code space requirements are heavily specialized for serving web pages and use an approach where the web server does not hold any connection state at all. For example, the iPic match-head sized server [26] and Jeremy Bentham's PICmicro stack [1] require only a few tens of bytes of RAM to serve simple web pages. In such an implementation, retransmissions cannot be made by the TCP module in the embedded system because nothing is known about the active connections. In order to achieve reliable transfers, the system has to rely on the remote host to perform retransmissions. It is possible to run a very simple web server with such an implementation, but there are serious limitations such as not being able to serve web pages that are larger than the size of a single TCP segment, which typically is about one kilobyte.

Other TCP/IP implementations such as the Atmel TCP/IP stack [5] save code space by leaving out certain vital TCP mechanisms. In particular, they often leave out TCP's congestion control mechanisms, which are used to reduce the sending rate when the network is overloaded. While an implementation with no congestion control might work well when connected to a single Ethernet segment, problems can arise when communication spans several networks. In such cases, the intermediate nodes such as switches and routers may be overloaded. Because congestion primarily is caused by the amount of pack-

ets in the network, and not the size of these packets, even small 8-bit systems are able to produce enough traffic to cause congestion. A TCP/IP implementation lacking congestion control mechanisms should not be used over the global Internet as it might contribute to congestion collapse [9].

Texas Instrument's MSP430 TCP/IP stack [6] and the TinyTCP code [4] use another common simplification in that they can handle only one TCP connection at a time. While this is a sensible simplification for many applications, it seriously limits the usefulness of the TCP/IP implementation. For example, it is not possible to communicate with two simultaneous peers with such an implementation. The CMX Micronet stack [27] uses a similar simplification in that it sets a hard limit of 16 on the maximum number of connections.

Yet another simplification that is used by LiveDevices Embedinet implementation [12] and others is to disregard the maximum segment size that a receiver is prepared to handle. Instead, the implementation will send segments that fit into an Ethernet frame of 1500 bytes. This works in a lot of cases due to the fact that many hosts are able to receive packets that are 1500 bytes or larger. Communication will fail, however, if the receiver is a system with limited memory resources that is not able to handle packets of that size.

Finally, the most common simplification is to leave out support for reassembling fragmented IP packets. Even though fragmented IP packets are quite infrequent [25], there are situations in which they may occur. If packets travel over a path which fragments the packets, communication is impossible if the TCP/IP implementation is unable to correctly reassemble them. TCP/IP implementations that are able to correctly reassemble fragmented IP packets, such as the Kadak KwikNET stack [22], are usually too large in terms of code size and RAM requirements to be practical for 8-bit systems.

6.4 RFC-compliance

The formal requirements for the protocols in the TCP/IP stack is specified in a number of RFC documents published by the Internet Engineering Task Force, IETF. Each of the protocols in the stack is defined in one more RFC documents and RFC1122 [2] collects all requirements and updates the previous RFCs.

The RFC1122 requirements can be divided into two categories; those that deal with the host to host communication and those that deal with communication between the application and the networking stack. An example of the first kind is *"A TCP MUST be able to receive a TCP option in any segment"* and an example of the second kind is *"There MUST be a mechanism for reporting soft*

Table 6.1: TCP/IP features implemented by uIP and lwIP

Feature	uIP	lwIP
IP and TCP checksums	x	x
IP fragment reassembly	x	x
IP options		
Multiple interfaces		x
UDP		x
Multiple TCP connections	x	x
TCP options	x	x
Variable TCP MSS	x	x
RTT estimation	x	x
TCP flow control	x	x
Sliding TCP window		x
TCP congestion control	Not needed	x
Out-of-sequence TCP data		x
TCP urgent data	x	x
Data buffered for rexmit		x

TCP error conditions to the application." A TCP/IP implementation that violates requirements of the first kind may not be able to communicate with other TCP/IP implementations and may even lead to network failures. Violation of the second kind of requirements will only affect the communication within the system and will not affect host-to-host communication.

In our implementations, we have implemented all RFC requirements that affect host-to-host communication. However, in order to reduce code size, we have removed certain mechanisms in the interface between the application and the stack, such as the soft error reporting mechanism and dynamically configurable type-of-service bits for TCP connections. Since there are only very few applications that make use of those features, we believe that they can be removed without loss of generality. Table 6.1 lists the features that uIP and lwIP implements.

6.5 Memory and buffer management

In our target architecture, RAM is the most scarce resource. With only a few kilobytes of RAM available for the TCP/IP stack to use, mechanisms used in

traditional TCP/IP cannot be directly applied.

Because of the different design goals for the lwIP and the uIP implementations, we have chosen two different memory management solutions. The lwIP implementation has dynamic buffer and memory allocation mechanisms where memory for holding connection state and packets is dynamically allocated from a global pool of available memory blocks. Packets are contained in one or more dynamically allocated buffers of fixed size. The size of the packet buffers is determined by a configuration option at compile time. Buffers are allocated by the network device driver when an incoming packet arrives. If the packet is larger than one buffer, more buffers are allocated and the packet is split into the buffers. If the incoming packet is queued by higher layers of the stack or the application, a reference counter in the buffer is incremented. The buffer will not be deallocated until the reference count is zero.

The uIP stack does not use explicit dynamic memory allocation. Instead, it uses a single global buffer for holding packets and has a fixed table for holding connection state. The global packet buffer is large enough to contain one packet of maximum size. When a packet arrives from the network, the device driver places it in the global buffer and calls the TCP/IP stack. If the packet contains data, the TCP/IP stack will notify the corresponding application. Because the data in the buffer will be overwritten by the next incoming packet, the application will either have to act immediately on the data or copy the data into a secondary buffer for later processing. The packet buffer will not be overwritten by new packets before the application has processed the data. Packets that arrive when the application is processing the data must be queued, either by the network device or by the device driver. Most single-chip Ethernet controllers have on-chip buffers that are large enough to contain at least 4 maximum sized Ethernet frames. Devices that are handled by the processor, such as RS-232 ports, can copy incoming bytes to a separate buffer during application processing. If the buffers are full, the incoming packet is dropped. This will cause performance degradation, but only when multiple connections are running in parallel. This is because uIP advertises a very small receiver window, which means that only a single TCP segment will be in the network per connection.

Outgoing data is also handled differently because of the different buffer schemes. In lwIP, an application that wishes to send data passes the length and a pointer to the data to the TCP/IP stack as well as a flag which indicates whether the data is volatile or not. The TCP/IP stack allocates buffers of suitable size and, depending on the volatile flag, either copies the data into the buffers or references the data through pointers. The allocated buffers contain space for the TCP/IP stack to prepend the TCP/IP and link layer headers.

After the headers are written, the stack passes the buffers to the network device driver. The buffers are not deallocated when the device driver is finished sending the data, but held on a retransmission queue. If the data is lost in the network and have to be retransmitted, the buffers on retransmission queue will be retransmitted. The buffers are not deallocated until the data is known to be received by the peer. If the connection is aborted because of an explicit request from the local application or a reset segment from the peer, the connection's buffers are deallocated.

In uIP, the same global packet buffer that is used for incoming packets is also used for the TCP/IP headers of outgoing data. If the application sends dynamic data, it may use the parts of the global packet buffer that are not used for headers as a temporary storage buffer. To send the data, the application passes a pointer to the data as well as the length of the data to the stack. The TCP/IP headers are written into the global buffer and once the headers have been produced, the device driver sends the headers and the application data out on the network. The data is not queued for retransmissions. Instead, the application will have to reproduce the data if a retransmission is necessary.

The total amount of memory usage for our implementations depends heavily on the applications of the particular device in which the implementations are to be run. The memory configuration determines both the amount of traffic the system should be able to handle and the maximum amount of simultaneous connections. A device that will be sending large e-mails while at the same time running a web server with highly dynamic web pages and multiple simultaneous clients, will require more RAM than a simple Telnet server. It is possible to run the uIP implementation with as little as 200 bytes of RAM, but such a configuration will provide extremely low throughput and will only allow a small number of simultaneous connections.

6.6 Application program interface

The Application Program Interface (API) defines the way the application program interacts with the TCP/IP stack. The most commonly used API for TCP/IP is the BSD socket API which is used in most Unix systems and has heavily influenced the Microsoft Windows WinSock API. Because the socket API uses stop-and-wait semantics, it requires support from an underlying multitasking operating system. Since the overhead of task management, context switching and allocation of stack space for the tasks might be too high in our target architecture, the BSD socket interface is not suitable for our purposes.

Instead, we have chosen an event driven interface where the application is invoked in response to certain events. Examples of such events are data arriving on a connection, an incoming connection request, or a poll request from the stack. The event based interface fits well in the event based structure used by operating systems such as TinyOS [10]. Furthermore, because the application is able to act on incoming data and connection requests as soon as the TCP/IP stack receives the packet, low response times can be achieved even in low-end systems.

6.7 Protocol implementations

The protocols in the TCP/IP protocol suite are designed in a layered fashion where each protocol performs a specific function and the interactions between the protocol layers are strictly defined. While the layered approach is a good way to design protocols, it is not always the best way to implement them. For the lwIP implementation, we have chosen a fully modular approach where each protocol implementation is kept fairly separate from the others. In the smaller uIP implementation, the protocol implementations are tightly coupled in order to save code space.

6.7.1 Main control loop

The lwIP and uIP stacks can be run either as a task in a multitasking system, or as the main program in a singletasking system. In both cases, the main control loop (Figure 6.3) does two things repeatedly:

1. Check if a packet has arrived from the network.

2. Check if a periodic timeout has occurred.

If a packet has arrived, the input handler of the TCP/IP stack is invoked. The input handler function will never block, but will return at once. When it returns, the stack or the application for which the incoming packet was intended may have produced one or more reply packets which should be sent out. If so, the network device driver is called to send out these packets.

Periodic timeouts are used to drive TCP mechanisms that depend on timers, such as delayed acknowledgments, retransmissions and round-trip time estimations. When the main control loop infers that the periodic timer should fire, it invokes the timer handler of the TCP/IP stack. Because the TCP/IP stack may

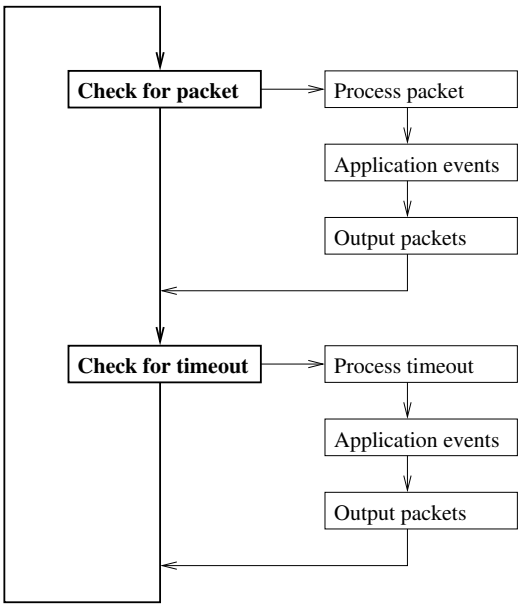

Figure 6.3: The main control loop.

perform retransmissions when dealing with a timer event, the network device driver is called to send out the packets that may have been produced.

This is similar to how the BSD implementations drive the TCP/IP stack, but BSD uses software interrupts and a task scheduler to initiate input handlers and timers. In our limited system, we do not depend on such mechanisms being available.

6.7.2 IP — Internet Protocol

When incoming packets are processed by lwIP and uIP, the IP layer is the first protocol that examines the packet. The IP layer does a few simple checks such as if the destination IP address of the incoming packet matches any of the local IP address and verifies the IP header checksum. Since there are no IP options that are strictly required and because they are very uncommon, both lwIP and uIP drop any IP options in received packets.

IP fragment reassembly

In both lwIP and uIP, IP fragment reassembly is implemented using a separate buffer that holds the packet to be reassembled. An incoming fragment is copied into the right place in the buffer and a bit map is used to keep track of which fragments have been received. Because the first byte of an IP fragment is aligned on an 8-byte boundary, the bit map requires a small amount of memory. When all fragments have been reassembled, the resulting IP packet is passed to the transport layer. If all fragments have not been received within a specified time frame, the packet is dropped.

The current implementation only has a single buffer for holding packets to be reassembled, and therefore does not support simultaneous reassembly of more than one packet. Since fragmented packets are uncommon, we belive this to be a reasonable decision. Extending our implementation to support multiple buffers would be straightforward, however.

Broadcasts and multicasts

IP has the ability to broadcast and multicast packets on the local network. Such packets are addressed to special broadcast and multicast addresses. Broadcast is used heavily in many UDP based protocols such as the Microsoft Windows file-sharing SMB protocol. Multicast is primarily used in protocols used for multimedia distribution such as RTP. TCP is a point-to-point protocol and does not use broadcast or multicast packets.

Because lwIP supports applications using UDP, it has support for both sending and receiving broadcast and multicast packets. In contrast, uIP does not have UDP support and therefore handling of such packets has not been implemented.

6.7.3 ICMP — Internet Control Message Protocol

The ICMP protocol is used for reporting soft error conditions and for querying host parameters. Its main use is, however, the echo mechanism which is used by the `ping` program.

The ICMP implementations in lwIP and uIP are very simple as we have restricted them to only implement ICMP echo messages. Replies to echo messages are constructed by simply swapping the source and destination IP addresses of incoming echo requests and rewriting the ICMP header with the Echo-Reply message type. The ICMP checksum is adjusted using standard techniques [23].

Since only the ICMP echo message is implemented, there is no support for Path MTU discovery or ICMP redirect messages. Neither of these is strictly required for interoperability; they are performance enhancement mechanisms.

6.7.4 TCP — Transmission Control Protocol

The TCP implementations in lwIP and uIP are driven by incoming packets and timer events. IP calls TCP when a TCP packet arrives and the main control loop calls TCP periodically.

Incoming packets are parsed by TCP and if the packet contains data that is to be delivered to the application, the application is invoked by the means of a function call. If the incoming packet acknowledges previously sent data, the connection state is updated and the application is informed, allowing it to send out new data.

Listening connections

TCP allows a connection to listen for incoming connection requests. In our implementations, a listening connection is identified by the 16-bit port number and incoming connection requests are checked against the list of listening connections. This list of listening connections is dynamic and can be altered by the applications in the system.

Sending data

When sending data, an application will have to check the number of available bytes in the send window and adjust the number of bytes to send accordingly. The size of the send window is dictated by the memory configuration as well as the buffer space announced by the receiver of the data. If no buffer space is available, the application has to defer the send and wait until later.

Buffer space becomes available when an acknowledgment from the receiver of the data has been received. The stack informs the application of this event, and the application may then repeat the sending procedure.

Sliding window

Most TCP implementations use a sliding window mechanism for sending data. Multiple data segments are sent in succession without waiting for an acknowledgment for each segment.

The sliding window algorithm uses a lot of 32-bit operations and because 32-bit arithmetic is fairly expensive on most 8-bit CPUs, uIP does not implement it. Also, uIP does not buffer sent packets and a sliding window implementation that does not buffer sent packets will have to be supported by a complex application layer. Instead, uIP allows only a single TCP segment per connection to be unacknowledged at any given time. lwIP, on the other hand, implements TCP's sliding window mechanism using output buffer queues and therefore does not add additional complexity to the application layer.

It is important to note that even though most TCP implementations use the sliding window algorithm, it is not required by the TCP specifications. Removing the sliding window mechanism does not affect interoperability in any way.

Round-trip time estimation

TCP continuously estimates the current Round-Trip Time (RTT) of every active connection in order to find a suitable value for the retransmission time-out.

We have implemented the RTT estimation using TCP's periodic timer. Each time the periodic timer fires, it increments a counter for each connection that has unacknowledged data in the network. When an acknowledgment is received, the current value of the counter is used as a sample of the RTT. The sample is used together with the standard TCP RTT estimation function [13] to calculate an estimate of the RTT. Karn's algorithm [14] is used to ensure that retransmissions do not skew the estimates.

Retransmissions

Retransmissions are driven by the periodic TCP timer. Every time the periodic timer is invoked, the retransmission timer for each connection is decremented. If the timer reaches zero, a retransmission should be made.

The actual retransmission operation is handled differently in uIP and in lwIP. lwIP maintains two output queues: one holds segments that have not yet been sent, the other holds segments that have been sent but not yet been acknowledged by the peer. When a retransmission is required, the first segment on the queue of segments that has not been acknowledged is sent. All other segments in the queue are moved to the queue with unsent segments.

As uIP does not keep track of packet contents after they have been sent by the device driver, uIP requires that the application takes an active part in performing the retransmission. When uIP decides that a segment should be re-

transmitted, it calls the application with a flag set indicating that a retransmission is required. The application checks the retransmission flag and produces the same data that was previously sent. From the application's standpoint, performing a retransmission is not different from how the data originally was sent. Therefore the application can be written in such a way that the same code is used both for sending data and retransmitting data. Also, it is important to note that even though the actual retransmission operation is carried out by the application, it is the responsibility of the stack to know when the retransmission should be made. Thus the complexity of the application does not necessarily increase because it takes an active part in doing retransmissions.

Flow control

The purpose of TCP's flow control mechanisms is to allow communication between hosts with wildly varying memory dimensions. In each TCP segment, the sender of the segment indicates its available buffer space. A TCP sender must not send more data than the buffer space indicated by the receiver.

In our implementations, the application cannot send more data than the receiving host can buffer. Before sending data, the application checks how many bytes it is allowed to send and does not send more data than the other host can accept. If the remote host cannot accept any data at all, the stack initiates the zero window probing mechanism.

The application is responsible for controlling the size of the window size indicated in sent segments. If the application must wait or buffer data, it can explicitly close the window so that the sender will not send data until the application is able to handle it.

Congestion control

The congestion control mechanisms limit the number of simultaneous TCP segments in the network. The algorithms used for congestion control [13] are designed to be simple to implement and require only a few lines of code.

Since uIP only handles one in-flight TCP segment per connection, the amount of simultaneous segments cannot be further limited, thus the congestion control mechanisms are not needed. lwIP has the ability to have multiple in-flight segments and therefore implements all of TCP's congestion control mechanisms.

Urgent data

TCP's urgent data mechanism provides an application-to-application notification mechanism, which can be used by an application to mark parts of the data stream as being more urgent than the normal stream. It is up to the receiving application to interpret the meaning of the urgent data.

In many TCP implementations, including the BSD implementation, the urgent data feature increases the complexity of the implementation because it requires an asynchronous notification mechanism in an otherwise synchronous API. As our implementations already use an asynchronous event based API, the implementation of the urgent data feature does not lead to increased complexity.

Connection state

Each TCP connection requires a certain amount of state information in the embedded device. Because the state information uses RAM, we have aimed towards minimizing the amount of state needed for each connection in our implementations.

The uIP implementation, which does not use the sliding window mechanism, requires far less state information than the lwIP implementation. The sliding window implementation requires that the connection state includes several 32-bit sequence numbers, not only for keeping track of the current sequence numbers of the connection, but also for remembering the sequence numbers of the last window updates. Furthermore, because lwIP is able to handle multiple local IP addresses, the connection state must include the local IP address. Finally, as lwIP maintains queues for outgoing segments, the memory for the queues is included in the connection state. This makes the state information needed for lwIP nearly 60 bytes larger than that of uIP which requires 30 bytes per connection.

6.8 Results

6.8.1 Performance limits

In TCP/IP implementations for high-end systems, processing time is dominated by the checksum calculation loop, the operation of copying packet data and context switching [15]. Operating systems for high-end systems often have multiple protection domains for protecting kernel data from user processes and

user processes from each other. Because the TCP/IP stack is run in the kernel, data has to be copied between the kernel space and the address space of the user processes and a context switch has to be performed once the data has been copied. Performance can be enhanced by combining the copy operation with the checksum calculation [19]. Because high-end systems usually have numerous active connections, packet demultiplexing is also an expensive operation [17].

A small embedded device does not have the necessary processing power to have multiple protection domains and the power to run a multitasking operating system. Therefore there is no need to copy data between the TCP/IP stack and the application program. With an event based API there is no context switch between the TCP/IP stack and the applications.

In such limited systems, the TCP/IP processing overhead is dominated by the copying of packet data from the network device to host memory, and checksum calculation. Apart from the checksum calculation and copying, the TCP processing done for an incoming packet involves only updating a few counters and flags before handing the data over to the application. Thus an estimate of the CPU overhead of our TCP/IP implementations can be obtained by calculating the amount of CPU cycles needed for the checksum calculation and copying of a maximum sized packet.

6.8.2 The impact of delayed acknowledgments

Most TCP receivers implement the delayed acknowledgment algorithm [3] for reducing the number of pure acknowledgment packets sent. A TCP receiver using this algorithm will only send acknowledgments for every other received segment. If no segment is received within a specific time-frame, an acknowledgment is sent. The time-frame can be as high as 500 ms but typically is 200 ms.

A TCP sender such as uIP that only handles a single outstanding TCP segment will interact poorly with the delayed acknowledgment algorithm. Because the receiver only receives a single segment at a time, it will wait as much as 500 ms before an acknowledgment is sent. This means that the maximum possible throughput is severely limited by the 500 ms idle time.

Thus the maximum throughput equation when sending data from uIP will be $p = s/(t+t_d)$ where s is the segment size and t_d is the delayed acknowledgment timeout, which typically is between 200 and 500 ms. With a segment size of 1000 bytes, a round-trip time of 40 ms and a delayed acknowledgment timeout of 200 ms, the maximum throughput will be 4166 bytes per second. With

the delayed acknowledgment algorithm disabled at the receiver, the maximum throughput would be 25000 bytes per second.

It should be noted, however, that since small systems running uIP are not very likely to have large amounts of data to send, the delayed acknowledgment throughput degradation of uIP need not be very severe. Small amounts of data sent by such a system will not span more than a single TCP segment, and would therefore not be affected by the throughput degradation anyway.

The maximum throughput when uIP acts as a receiver is not affected by the delayed acknowledgment throughput degradation.

6.8.3 Measurements

For our experiments we connected a 450 MHz Pentium III PC running FreeBSD 4.7 to an Ethernut board [16] through a dedicated 10 megabit/second Ethernet network. The Ethernut board is a commercially available embedded system equipped with a RealTek RTL8019AS Ethernet controller, an Atmel Atmega128 AVR microcontroller running at 14.7456 MHz with 128 kilobytes of flash ROM for code storage and 32 kilobytes of RAM. The FreeBSD host was configured to run the Dummynet delay emulator software [24] in order to facilitate controlled delays for the communication between the PC and the embedded system.

In the embedded system, a simple web server was run on top of the uIP and lwIP stacks. Using the `fetch` file retrieval utility, a file consisting of null bytes was downloaded ten times from the embedded system. The reported throughput was logged, and the mean throughput of the ten downloads was calculated. By redirecting file output to `/dev/null`, the file was immediately discarded by the FreeBSD host. The file size was 200 kilobytes for the uIP tests, and 200 megabytes for the lwIP tests. The size of the file made it impossible to keep it all in the memory of the embedded system. Instead, the file was generated by the web server as it was sent out on the network.

The total TCP/IP memory consumption in the embedded system was varied by changing the send window size. For uIP, the send window was varied between 50 bytes and the maximum possible value of 1450 bytes in steps of 50 bytes. The send window configuration translates into a total RAM usage of between 400 bytes and 3 kilobytes. The lwIP send window was varied between 500 and 11000 bytes in steps of 500 bytes, leading to a total RAM consumption of between 5 and 16 kilobytes.

Figure 6.4 shows the mean throughput of the ten file downloads from the web server running on top of uIP, with an additional 10 ms delay created by

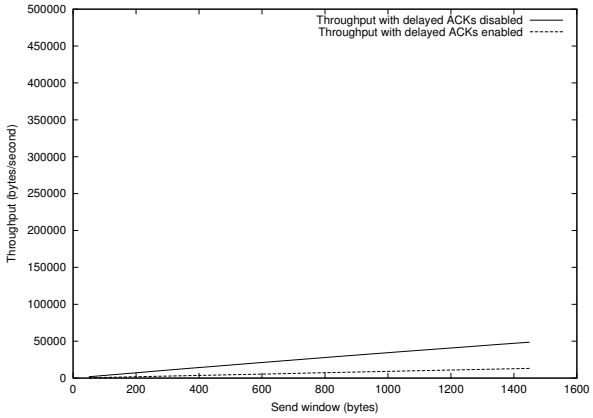

Figure 6.4: uIP sending data with 10 ms emulated delay.

the Dummynet delay emulator. The two curves show the measured throughput with the delayed acknowledgment algorithm disabled and enabled at the receiving FreeBSD host, respectively. The performance degradation caused by the delayed acknowledgments is evident.

Figure 6.5 shows the same setup, but without the 10 ms emulated delay. The lower curve, showing the throughput with delayed acknowledgments enabled, is very similar to the lower one in Figure 6.4. The upper curve, however, does not show the same linear relation as the previous figure, but shows an increasing throughput where the increase declines with increasing send window size. One explanation for the declining increase of throughput is that the round-trip time increases with the send window size because of the increased per-packet processing time. Figure 6.6 shows the round-trip time as a function of packet size. These measurements were taken using the `ping` program and therefore include the cost for the packet copying operation twice; once for packet input and once for packet output.

The throughput of lwIP shows slightly different characteristics. Figure 6.7 shows three measured throughput curves, without emulated delay, and with emulated delays of 10 ms and 20 ms. For all measurements, the delayed acknowledgment algorithm is enabled at the FreeBSD receiver. We see that for small send window sizes, lwIP also suffers from the delayed acknowledgment throughput degradation. With a send window larger than two maximum TCP

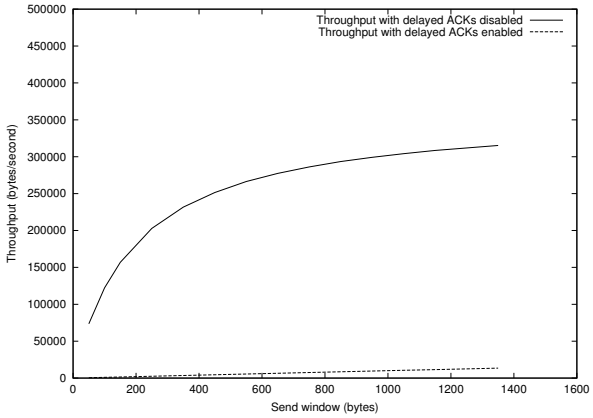

Figure 6.5: uIP sending data without emulated delay.

segment sizes (3000 bytes), lwIP is able to send out two TCP segments per round-trip time and thereby avoids the delayed acknowledgments throughput degradation. Without emulated delay, the throughput quickly reaches a maximum of about 415 kilobytes per second. This limit is likely to be the processing limit of the lwIP code in the embedded system and therefore is the maximum possible throughput for lwIP in this particular system.

The maximum throughput with emulated delays is lower than without delay emulation, and the similarity of the two curves suggests that the throughput degradation could be caused by interaction with the Dummynet software.

6.8.4 Code size

The code was compiled for the 32-bit Intel x86 and the 8-bit Atmel AVR platforms using gcc [28] versions 2.95.3 and 3.3 respectively, with code size optimization turned on. The resulting size of the compiled code can be seen in Tables 6.2 to 6.5. Even though both implementations support ARP and SLIP and lwIP includes UDP, only the protocols discussed in this paper are presented. Because the protocol implementations in uIP are tightly coupled, the individual sizes of the implementations are not reported.

There are several reasons for the dramatic difference in code size between lwIP and uIP. In order to support the more complex and configurable TCP im-

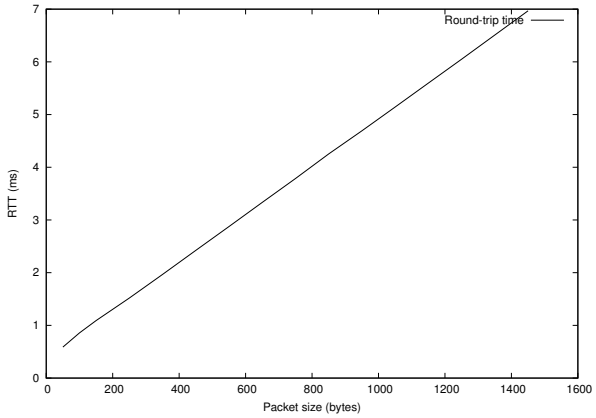

Figure 6.6: Round-trip time as a function of packet size.

Table 6.2: Code size for uIP (x86)

Function	Code size (bytes)
Checksumming	464
IP, ICMP and TCP	4724
Total	5188

plementation, lwIP has significantly more complex buffer and memory management than uIP. Since lwIP can handle packets that span several buffers, the checksum calculation functions in lwIP are more complex than those in uIP. The support for dynamically changing network interfaces in lwIP also contributes to the size increase of the IP layer because the IP layer has to manage multiple local IP addresses. The IP layer in lwIP is further made larger by the fact that lwIP has support for UDP, which requires that the IP layer is able handle broadcast and multicast packets. Likewise, the ICMP implementation in lwIP has support for UDP error messages which have not been implemented in uIP.

The TCP implementation is lwIP is nearly twice as large as the full IP, ICMP and TCP implementation in uIP. The main reason for this is that lwIP implements the sliding window mechanism which requires a large amount of

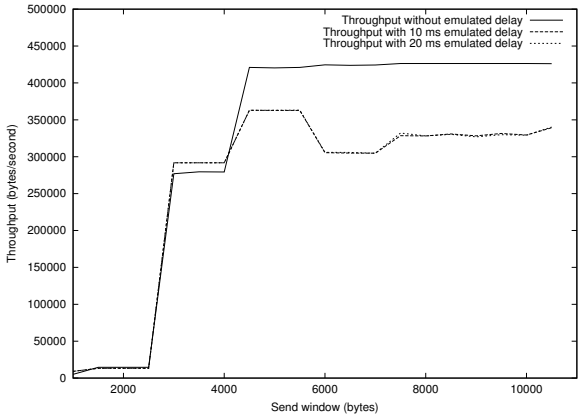

Figure 6.7: lwIP sending data with and without emulated delays.

Table 6.3: Code size for uIP (AVR)

Function	Code size (bytes)
Checksumming	712
IP, ICMP and TCP	4452
Total	5164

buffer and queue management functionality that is not required in uIP.

The different memory and buffer management schemes used by lwIP and uIP have implications on code size, mainly in 8-bit systems. Because uIP uses a global buffer for all incoming packets, the absolute memory addresses of the protocol header fields are known at compile time. Using this information, the compiler is able to generate code that uses absolute addressing, which on many 8-bit processors requires less code than indirect addressing.

Is it interesting to note that the size of the compiled lwIP code is larger on the AVR than on the x86, while the uIP code is of about the same size on the two platforms. The main reason for this is that lwIP uses 32-bit arithmetic to a much larger degree than uIP and each 32-bit operation is compiled into a large number of machine code instructions.

Table 6.4: Code size for lwIP (x86)

Function	Code size (bytes)
Memory management	2512
Checksumming	504
Network interfaces	364
IP	1624
ICMP	392
TCP	9192
Total	14588

Table 6.5: Code size for lwIP (AVR)

Function	Code size (bytes)
Memory management	3142
Checksumming	1116
Network interfaces	458
IP	2216
ICMP	594
TCP	14230
Total	21756

6.9 Future work

Prioritized connections. It is advantageous to be able to prioritize certain connections such as Telnet connections for manual configuration of the device. Even in a system that is under heavy load from numerous clients, it should be possible to remotely control and configure the device. In order to do provide this, different connection types could be given different priority. For efficiency, such differentiation should be done as far down in the system as possible, preferably in the device driver.

Security aspects. When connecting systems to a network, or even to the global Internet, the security of the system is very important. Identifying levels of security and mechanisms for implementing security for embedded devices is crucial for connecting systems to the global Internet.

Address auto-configuration. If hundreds or even thousands of small em-

bedded devices should be deployed, auto-configuration of IP addresses is advantageous. Such mechanisms already exist in IPv6, the next version of the Internet Protocol, and are currently being standardized for IPv4.

Improving throughput. The throughput degradation problem caused by the poor interaction with the delayed acknowledgment algorithm should be fixed. By increasing the maximum number of in-flight segments from one to two, the problem will not appear. When increasing the amount of in-flight segments, congestion control mechanisms will have to be employed. Those mechanisms are trivial, however, when the upper limit is two simultaneous segments.

Performance enhancing proxy. It might be possible to increase the performance of communication with the embedded devices through the use of a proxy situated near the devices. Such a proxy would have more memory than the devices and could assume responsibility for buffering data.

6.10 Summary and conclusions

We have shown that it is possible to fit a full scale TCP/IP implementation well within the limits of an 8-bit microcontroller, but that the throughput of such a small implementation will suffer. We have not removed any TCP/IP mechanisms in our implementations, but have full support for reassembly of IP fragments and urgent TCP data. Instead, we have minimized the interface between the TCP/IP stack and the application.

The maximum achievable throughput for our implementations is determined by the send window size that the TCP/IP stack has been configured to use. When sending data with uIP, the delayed ACK mechanism at the receiver lowers the maximum achievable throughput considerably. In many situations however, a limited system running uIP will not produce so much data that this will cause problems. lwIP is not affected by the delayed ACK throughput degradation when using a large enough send window.

6.11 Acknowledgments

Many thanks go to Martin Nilsson, who has provided encouragement and been a source of inspiration throughout the preparation of this paper. Thanks also go to Deborah Wallach for comments and suggestions, the anonymous reviewers whose comments were highly appreciated, and to all who have contributed bugfixes, patches and suggestions to the lwIP and uIP implementations.

Bibliography

[1] J. Bentham. *TCP/IP Lean: Web servers for embedded systems*. CMP Books, October 2000.

[2] R. Braden. Requirements for internet hosts – communication layers. RFC 1122, Internet Engineering Task Force, October 1989.

[3] D. D. Clark. Window and acknowledgement strategy in TCP. RFC 813, Internet Engineering Task Force, July 1982.

[4] G. H. Cooper. TinyTCP. Web page. 2002-10-14. URL: http://www.csonline.net/bpaddock/tinytcp/

[5] Atmel Corporation. Embedded web server. AVR 460, January 2001. Avalible from www.atmel.com.

[6] A. Dannenberg. MSP430 internet connectivity. SLAA 137, November 2001. Avalible from www.ti.com.

[7] A. Dunkels. lwIP - a lightweight TCP/IP stack. Web page. 2002-10-14. URL: http://www.sics.se/~adam/lwip/

[8] A. Dunkels. uIP - a TCP/IP stack for 8- and 16-bit microcontrollers. Web page. 2003-10-21. URL: http://dunkels.com/adam/uip/

[9] S. Floyd and K. Fall. Promoting the use of end-to-end congestion control in the internet. *IEEE/ACM Transactions on Networking*, August 1999.

[10] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[11] InterNiche Technologies Inc. NicheStack portable TCP/IP stack. Web page. 2002-10-14. URL: http://www.iniche.com/products/tcpip.htm

[12] LiveDevices Inc. Embedinet - embedded internet software products. Web page. 2002-10-14. URL: http://www.livedevices.com/net_products/embedinet.shtml

[13] V. Jacobson. Congestion avoidance and control. In *Proceedings of the SIGCOMM '88 Conference*, Stanford, California, August 1988.

[14] P. Karn and C. Partridge. Improving round-trip time estimates in reliablie transport protocols. In *Proceedings of the SIGCOMM '87 Conference*, Stowe, Vermont, August 1987.

[15] J. Kay and J. Pasquale. The importance of non-data touching processing overheads in TCP/IP. In *Proceedings of the ACM SIGCOMM '93 Symposium*, pages 259–268, September 1993.

[16] H. Kipp. Ethernut embedded ethernet. Web page. 2002-10-14. URL: http://www.ethernut.de/en/

[17] P. E. McKenney and K. F. Dove. Efficient demultiplexing of incoming TCP packets. In *Proceedings of the SIGCOMM '92 Conference*, pages 269–279, Baltimore, Maryland, August 1992.

[18] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison-Wesley, 1996.

[19] C. Partridge and S. Pink. A faster UDP. *IEEE/ACM Transactions in Networking*, 1(4):429–439, August 1993.

[20] J. Postel. Internet protocol. RFC 791, Internet Engineering Task Force, September 1981.

[21] J. Postel. Transmission control protocol. RFC 793, Internet Engineering Task Force, September 1981.

[22] Kadak Products. Kadak KwikNET TCP/IP stack. Web page. 2002-10-14. URL: http://www.kadak.com/html/kdkp1030.htm

[23] A. Rijsinghani. Computation of the internet checksum via incremental update. RFC 1624, Internet Engineering Task Force, May 1994.

[24] Luigi Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1):31–41, 1997.

[25] C. Shannon, D. Moore, and K. Claffy. Beyond folklore: Observations on fragmented traffic. *IEEE/ACM Transactions on Networking*, 10(6), December 2002.

[26] H. Shrikumar. IPic - a match head sized web-server. Web page. 2002-10-14. URL: http://www-ccs.cs.umass.edu/~shri/iPic.html

[27] CMX Systems. CMX-MicroNet true TCP/IP networking. Web page. 2002-10-14. URL: http://www.cmx.com/micronet.htm

[28] The GCC Team. The GNU compiler collection. Web page. 2002-10-14. URL: http://gcc.gnu.org/

Chapter 7

Paper B: Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors

Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors. In *First IEEE Workshop on Embedded Networked Sensors*, November 2004.

Abstract

Wireless sensor networks are composed of large numbers of tiny networked devices that communicate untethered. For large scale networks it is important to be able to dynamically download code into the network. In this paper we present Contiki, a lightweight operating system with support for dynamic loading and replacement of individual programs and services. Contiki is built around an event-driven kernel but provides optional preemptive multithreading that can be applied to individual processes. We show that dynamic loading and unloading is feasible in a resource constrained environment, while keeping the base system lightweight and compact.

7.1 Introduction

Wireless sensor networks are composed of large numbers of tiny sensor devices with wireless communication capabilities. The sensor devices autonomously form networks through which sensor data is transported. The sensor devices are often severely resource constrained. An on-board battery or solar panel can only supply limited amounts of power. Moreover, the small physical size and low per-device cost limit the complexity of the system. Typical sensor devices [1, 2, 5] are equipped with 8-bit microcontrollers, code memory on the order of 100 kilobytes, and less than 20 kilobytes of RAM. Moore's law predicts that these devices can be made significantly smaller and less expensive in the future. While this means that sensor networks can be deployed to greater extents, it does not necessarily imply that the resources will be less constrained.

For the designer of an operating system for sensor nodes, the challenge lies in finding lightweight mechanisms and abstractions that provide a rich enough execution environment while staying within the limitations of the constrained devices. We have developed Contiki, an operating system developed for such constrained environments. Contiki provides dynamic loading and unloading of individual programs and services. The kernel is event-driven, but the system supports preemptive multi-threading that can be applied on a per-process basis. Preemptive multi-threading is implemented as a library that is linked only with programs that explicitly require multi-threading.

Contiki is implemented in the C language and has been ported to a number of microcontroller architectures, including the Texas Instruments MSP430 and the Atmel AVR. We are currently running it on the ESB platform [5]. The ESB uses the MSP430 microcontroller with 2 kilobytes of RAM and 60 kilobytes of ROM running at 1 MHz. The microcontroller has the ability to selectively reprogram parts of the on-chip flash memory.

The contributions of this paper are twofold. Our first contribution is that we show the feasibility of loadable programs and services even in a constrained sensor device. The possibility to dynamically load individual programs leads to a very flexible architecture, which still is compact enough for resource constrained sensor nodes. Our second contribution is more general in that we show that preemptive multi-threading does not have to be implemented at the lowest level of the kernel but that it can be built as an application library on top of an event-driven kernel. This allows for thread-based programs running on top of an event-based kernel, without the overhead of reentrancy or multiple stacks in all parts of the system.

7.1.1 Downloading code at run-time

Wireless sensor networks are envisioned to be large scale, with hundreds or even thousands of nodes per network. When developing software for such a large sensor network, being able to dynamically download program code into the network is of great importance. Furthermore, bugs may have to be patched in an operational network [9]. In general, it is not feasible to physically collect and reprogram all sensor devices and in-situ mechanisms are required. A number of methods for distributing code in wireless sensor networks have been developed [21, 8, 17]. For such methods it is important to reduce the number of bytes sent over the network, as communication requires a large parts of the available node energy.

Most operating systems for embedded systems require that a complete binary image of the entire system is built and downloaded into each device. The binary includes the operating system, system libraries, and the actual applications running on top of the system. In contrast, Contiki has the ability to load and unload individual applications or services at run-time. In most cases, an individual application is much smaller than the entire system binary and therefore requires less energy when transmitted through a network. Additionally, the transfer time of an application binary is less than that of an entire system image.

7.1.2 Portability

As the number of different sensor device platforms increases (e.g. [1, 2, 5]), it is desirable to have a common software infrastructure that is portable across hardware platforms. The currently available sensor platforms carry completely different sets of sensors and communication devices. Due to the application specific nature of sensor networks, we do not expect that this will change in the future. The single unifying characteristic of today's platforms is the CPU architecture which uses a memory model without segmentation or memory protection mechanisms. Program code is stored in reprogrammable ROM and data in RAM. We have designed Contiki so that the only abstraction provided by the base system is CPU multiplexing and support for loadable programs and services. As a consequence of the application specific nature of sensor networks, we believe that other abstractions are better implemented as libraries or services and provide mechanisms for dynamic service management.

7.1.3 Event-driven systems

In severely memory constrained environments, a multi-threaded model of operation often consumes large parts of the memory resources. Each thread must have its own stack and because it in general is hard to know in advance how much stack space a thread needs, the stack typically has to be over provisioned. Furthermore, the memory for each stack must be allocated when the thread is created. The memory contained in a stack can not be shared between many concurrent threads, but can only be used by the thread to which is was allocated. Moreover, a threaded concurrency model requires locking mechanisms to prevent concurrent threads from modifying shared resources.

To provide concurrency without the need for per-thread stacks or locking mechanisms, event-driven systems have been proposed [15]. In event-driven systems, processes are implemented as event handlers that run to completion. Because an event handler cannot block, all processes can use the same stack, effectively sharing the scarce memory resources between all processes. Also, locking mechanisms are generally not needed because two event handlers never run concurrently with respect to each other.

While event-driven system designs have been found to work well for many kinds of sensor network applications [18] they are not without problems. The state driven programming model can be hard to manage for programmers [17]. Also, not all programs are easily expressed as state machines. One example is the lengthy computation required for cryptographic operations. Typically, such operations take several seconds to complete on CPU constrained platforms [22]. In a purely event-driven operating system a lengthy computation completely monopolizes the CPU, making the system unable to respond to external events. If the operating system instead was based on preemptive multi-threading this would not be a problem as a lengthy computation could be preempted.

To combine the benefits of both event-driven systems and preemptible threads, Contiki uses a hybrid model: the system is based on an event-driven kernel where preemptive multi-threading is implemented as an application library that is *optionally* linked with programs that *explicitly* require it.

The rest of this paper is structured as follows. Section 7.2 reviews related work and Section 7.3 presents an overview of the Contiki system. We describe the design of the Contiki kernel in Section 7.4. The Contiki service concept is presented in Section 7.5. In the following section, we describe how Contiki handles libraries and communication support is discussed in Section 7.7. We present the implementation of preemptive multi-threading in Section 7.8 and

our experiences with using the system is discussed in Section 7.9. Finally, the paper is concluded in Section 7.10.

7.2 Related work

TinyOS [15] is probably the earliest operating system that directly targets the specific applications and limitations of sensor devices. TinyOS is also built around a lightweight event scheduler where all program execution is performed in tasks that run to completion. TinyOS uses a special description language for composing a system of smaller components [12] which are statically linked with the kernel to a complete image of the system. After linking, modifying the system is not possible [17]. In contrast, Contiki provides a dynamic structure which allows programs and drivers to be replaced during run-time and without relinking.

In order to provide run-time reprogramming for TinyOS, Levis and Culler have developed Maté [17], a virtual machine for TinyOS devices. Code for the virtual machine can be downloaded into the system at run-time. The virtual machine is specifically designed for the needs of typical sensor network applications. Similarly, the MagnetOS [7] system uses a virtual Java machine to distribute applications across the sensor network. The advantages of using a virtual machine instead of native machine code is that the virtual machine code can be made smaller, thus reducing the energy consumption of transporting the code over the network. One of the drawbacks is the increased energy spent in interpreting the code—for long running programs the energy saved during the transport of the binary code is instead spent in the overhead of executing the code. Contiki programs use native code and can therefore be used for all types of programs, including low level device drivers without loss of execution efficiency.

SensorWare [8] provides an abstract scripting language for programming sensors, but their target platforms are not as resource constrained as ours. Similarly, the EmStar environment [13] is designed for less resource constrained systems. Reijers and Langendoen [21] use a patch language to modify parts of the binary image of a running system. This works well for networks where all nodes run the exact same binary code but soon gets complicated if sensors run slightly different programs or different versions of the same software.

The Mantis system [3] uses a traditional preemptive multi-threaded model of operation. Mantis enables reprogramming of both the entire operating system and parts of the program memory by downloading a program image onto

EEPROM, from where it can be burned into flash ROM. Due to the multi-threaded semantics, every Mantis program must have stack space allocated from the system heap, and locking mechanisms must be used to achieve mutual exclusion of shared variables. In contrast, Contiki uses an event based scheduler without preemption, thus avoiding allocation of multiple stacks and locking mechanisms. Preemptive multi-threading is provided by a library that can be linked with programs that explicitly require it.

The preemptive multi-threading in Contiki is similar to fibers [4] and the lightweight fibers approach by Welsh and Mainland [23]. Unlike the lightweight fibers, Contiki does not limit the number of concurrent threads to two. Furthermore, unlike fibers, threads in Contiki support preemption.

As Exokernel [11] and Nemesis [16], Contiki tries to reduce the number of abstractions that the kernel provides to a minimum [10]. Abstractions are instead provided by libraries that have nearly full access to the underlying hardware. While Exokernel strived for performance and Nemesis aimed at quality of service, the purpose of the Contiki design is to reduce size and complexity, as well as to preserve flexibility. Unlike Exokernel, Contiki do not support any protection mechanisms since the hardware for which Contiki is designed do not support memory protection.

7.3 System overview

A running Contiki system consists of the kernel, libraries, the program loader, and a set of processes. A process may be either an application program or a *service*. A service implements functionality used by more than one application process. All processes, both application programs and services, can be dynamically replaced at run-time. Communication between processes always goes through the kernel. The kernel does not provide a hardware abstraction layer, but lets device drivers and applications communicate directly with the hardware.

A process is defined by an event handler function and an optional poll handler function. The process state is held in the process' private memory and the kernel only keeps a pointer to the process state. On the ESB platform [5], the process state consists of 23 bytes. All processes share the same address space and do not run in different protection domains. Interprocess communication is done by posting events.

A Contiki system is partitioned into two parts: the *core* and the *loaded programs* as shown in Figure 7.1. The partitioning is made at compile time

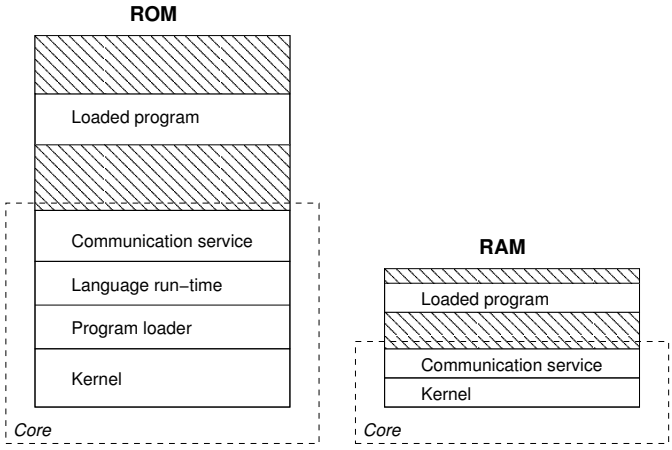

Figure 7.1: Partitioning into core and loaded programs.

and is specific to the deployment in which Contiki is used. Typically, the core consists of the Contiki kernel, the program loader, the most commonly used parts of the language run-time and support libraries, and a communication stack with device drivers for the communication hardware. The core is compiled into a single binary image that is stored in the devices prior to deployment. The core is generally not modified after deployment, even though it should be noted that it is possible to use a special boot loader to overwrite or patch the core.

Programs are loaded into the system by the program loader. The program loader may obtain the program binaries either by using the communication stack or by using directly attached storage such as EEPROM. Typically, programs to be loaded into the system are first stored in EEPROM before they are programmed into the code memory.

7.4 Kernel architecture

The Contiki kernel consists of a lightweight event scheduler that dispatches events to running processes and periodically calls processes' polling handlers. All program execution is triggered either by events dispatched by the kernel or through the polling mechanism. The kernel does not preempt an event handler once it has been scheduled. Therefore, event handlers must run to completion.

As shown in Section 7.8, however, event handlers may use internal mechanisms to achieve preemption.

The kernel supports two kind of events: *asynchronous* and *synchronous* events. Asynchronous events are a form of deferred procedure call: asynchronous events are enqueued by the kernel and are dispatched to the target process some time later. Synchronous events are similar to asynchronous but immediately causes the target process to be scheduled. Control returns to the posting process only after the target has finished processing the event. This can be seen as an inter-process procedure call and is similar to the door abstraction used in the Spring operating system [14].

In addition to the events, the kernel provides a *polling* mechanism. Polling can be seen as high priority events that are scheduled in-between each asynchronous event. Polling is used by processes that operate near the hardware to check for status updates of hardware devices. When a poll is scheduled all processes that implement a poll handler are called, in order of their priority.

The Contiki kernel uses a single shared stack for all process execution. The use of asynchronous events reduce stack space requirements as the stack is rewound between each invocation of event handlers.

7.4.1 Two level scheduling hierarchy

All event scheduling in Contiki is done at a single level and events cannot preempt each other. Events can only be preempted by interrupts. Normally, interrupts are implemented using hardware interrupts but may also be implemented using an underlying real-time executive. The latter technique has previously been used to provide real-time guarantees for the Linux kernel [6].

In order to be able to support an underlying real-time executive, Contiki never disables interrupts. Because of this, Contiki does not allow events to be posted by interrupt handlers as that would lead to race-conditions in the event handler. Instead, the kernel provides a polling flag that it used to request a poll event. The flag provides interrupt handlers with a way to request immediate polling.

7.4.2 Loadable programs

Loadable programs are implemented using a run-time relocation function and a binary format that contains relocation information. When a program is loaded into the system, the loader first tries to allocate sufficient memory space based

on information provided by the binary. If memory allocation fails, program loading is aborted.

After the program is loaded into memory, the loader calls the program's initialization function. The initialization function may start or replace one or more processes.

7.4.3 Power save mode

In sensor networks, being able to power down the node when the network is inactive is an often required way to reduce energy consumption. Power conservation mechanisms depend on both the applications [18] and the network protocols [20]. The Contiki kernel contains no explicit power save abstractions, but lets the the application specific parts of the system implement such mechanisms. To help the application decide when to power down the system, the event scheduler exposes the size of the event queue. This information can be used to power down the processor when there are no events scheduled. When the processor wakes up in response to an interrupt, the poll handlers are run to handle the external event.

7.5 Services

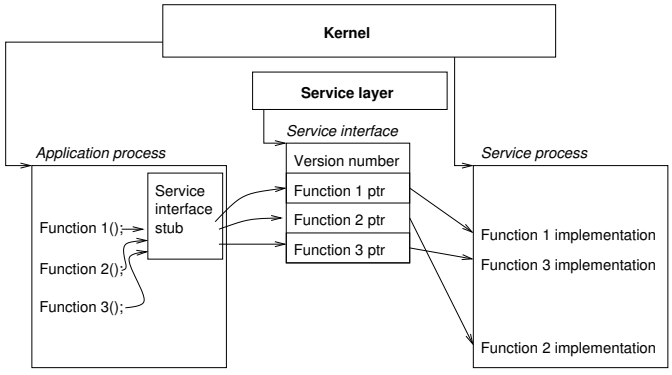

Figure 7.2: An application function calling a service.

In Contiki, a *service* is a process that implements functionality that can

be used by other processes. A service can be seen as a form of a shared library. Services can be dynamically replaced at run-time and must therefore be dynamically linked. Typical examples of services includes communication protocol stacks, sensor device drivers, and higher level functionality such as sensor data handling algorithms.

Services are managed by a *service layer* conceptually situated directly next to the kernel. The service layer keeps track of running services and provides a way to find installed services. A service is identified by a textual string that describes the service. The service layer uses ordinary string matching to querying installed services.

A service consists of a *service interface* and a process that implements the interface. The service interface consists of a version number and a function table with pointers to the functions that implement the interface.

Application programs using the service use a stub library to communicate with the service. The stub library is linked with the application and uses the service layer to find the service process. Once a service has been located, the service stub caches the process ID of the service process and uses this ID for all future requests.

Programs call services through the service interface stub and need not be aware of the fact that a particular function is implemented as a service. The first time the service is called, the service interface stub performs a service lookup in the service layer. If the specified service exists in the system, the lookup returns a pointer to the service interface. The version number in the service interface is checked with the version of the interface stub. In addition to the version number, the service interface contains pointers to the implementation of all service functions. The function implementations are contained in the service process. If the version number of the service stub match the number in the service interface, the interface stub calls the implementation of the requested function.

7.5.1 Service replacement

Like all processes, services may be dynamically loaded and replaced in a running Contiki system. Because the process ID of the service process is used as a service identifier, it is crucial that the process ID is retained if the service process is replaced. For this reason, the kernel provides special mechanism for replacing a process and retaining the process ID.

When a service is to be replaced, the kernel informs the running version of the service by posting a special event to the service process. In response to this

event, the service must remove itself from the system.

Many services have an internal state that may need to be transfered to the new process. The kernel provides a way to pass a pointer to the new service process, and the service can produce a state description that is passed to the new process. The memory for holding the state must be allocated from a shared source, since the process memory is deallocated when the old process is removed.

The service state description is tagged with the version number of the service, so that an incompatible version of the same service will not try to load the service description.

7.6 Libraries

The Contiki kernel only provides the most basic CPU multiplexing and event handling features. The rest of the system is implemented as system libraries that are optionally linked with programs. Programs can be linked with libraries in three different ways. First, programs can be statically linked with libraries that are part of the core. Second, programs can be statically linked with libraries that are part of the loadable program. Third, programs can call services implementing a specific library. Libraries that are implemented as services can be dynamically replaced at run-time.

Typically, run-time libraries such as often-used parts of the language run-time libraries are best placed in the Contiki core. Rarely used or application specific libraries, however, are more appropriately linked with loadable programs. Libraries that are part of the core are always present in the system and do not have to be included in loadable program binaries.

As an example, consider a program that uses the `memcpy()` and `atoi()` functions to copy memory and to convert strings to integers, respectively. The `memcpy()` function is a frequently used C library function, whereas `atoi()` is used less often. Therefore, in this particular example, `memcpy()` has been included in the system core but not `atoi()`. When the program is linked to produce a binary, the `memcpy()` function will be linked against its static address in the core. The object code for the part of the C library that implements the `atoi()` function must, however, be included in the program binary.

7.7 Communication support

Communication is a fundamental concept in sensor networks. In Contiki, communication is implemented as a service in order to enable run-time replacement. Implementing communication as a service also provides for multiple communication stacks to be loaded simultaneously. In experimental research, this can be used to evaluate and compare different communication protocols. Furthermore, the communication stack may be split into different services as shown in Figure 7.3. This enables run-time replacement of individual parts of the communication stack.

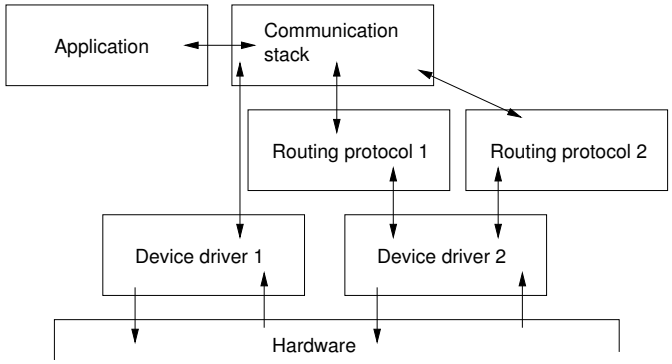

Figure 7.3: Loosely coupled communication stack.

Communication services use the service mechanism to call each other and synchronous events to communicate with application programs. Because synchronous event handlers are required to be run to completion, it is possible to use a single buffer for all communication processing. With this approach, no data copying has to be performed. A device driver reads an incoming packet into the communication buffer and then calls the upper layer communication service using the service mechanisms. The communication stack processes the headers of the packet and posts a synchronous event to the application program for which the packet was destined. The application program acts on the packet contents and optionally puts a reply in the buffer before it returns control to the communication stack. The communication stack prepends its headers to the outgoing packet and returns control to the device driver so that the packet can be transmitted.

7.8 Preemptive multi-threading

In Contiki, preemptive multi-threading is implemented as a library on top of the event-based kernel. The library is optionally linked with applications that explicitly require a multi-threaded model of operation. The library is divided into two parts: a platform independent part that interfaces to the event kernel, and a platform specific part implementing the stack switching and preemption primitives. Usually, the preemption is implemented using a timer interrupt that saves the processor registers onto the stack and switches back to the kernel stack. In practice very little code needs to be rewritten when porting the platform specific part of the library. For reference, the implementation for the MSP430 consists of 25 lines of C code.

Unlike normal Contiki processes each thread requires a separate stack. The library provides the necessary stack management functions. Threads execute on their own stack until they either explicitly yield or are preempted.

`mt_yield();`
Yield from the running thread.

`mt_post(id, event, dataptr);`
Post an event from the running thread.

`mt_wait(event, dataptr);`
Wait for an event to be posted to the running thread.

`mt_exit();`
Exit the running thread.

`mt_start(thread, functionptr, dataptr);`
Start a thread with a specified function call.

`mt_exec(thread);`
Execute the specified thread until it yields or is preempted.

Figure 7.4: The multi-threading library API.

The API of the multi-threading library is shown in Figure 7.4. It consists of four functions that can be called from a running thread (`mt_yield()`, `mt_post()`, `mt_wait()`, and `mt_exit()`) and two functions that are called to setup and run a thread (`mt_start()` and `mt_exec()`). The `mt_exec()` function performs the actual scheduling of a thread and is called from an event handler.

7.9 Discussion

We have used the Contiki operating system to implement a number of sensor network applications such as multi-hop routing, motion detection with distributed sensor data logging and replication, and presence detection and notification.

7.9.1 Over-the-air programming

We have implemented a simple protocol for over-the-air programming of entire networks of sensors. The protocol transmits a single program binary to selected concentrator nodes using point-to-point communication. The binary is stored in EEPROM and when the entire program has been received, it is broadcasted to neighboring nodes. Packet loss is signaled by neighbors using negative acknowledgments. Repairs are made by the concentrator node. We intend to implement better protocols, such as the Trickle algorithm [19], in the future.

During the development of one network application, a 40-node dynamic distributed alarm system, we used both over-the-air reprogramming and manual wired reprogramming of the sensor nodes. At first, the program loading mechanism was not fully functional and we could not use it during our development. The object code size of our application was approximately 6 kilobytes. Together with the Contiki core and the C library, the complete system image was nearly 30 kilobytes. Reprogramming of an individual sensor node took just over 30 seconds. With 40 nodes, reprogramming the entire network required at least 30 minutes of work and was therefore not feasible to do often. In contrast, over-the-air reprogramming of a single component of the application was done in about two minutes—a reduction in an order of magnitude—and could be done with the sensor nodes placed in the actual test environment.

7.9.2 Code size

An operating system for constrained devices must be compact in terms of both code size and RAM usage in order to leave room for applications running on top of the system. Table 7.1 shows the compiled code size and the RAM usage of the Contiki system compiled for two architectures: the Texas Instruments MSP430 and the Atmel AVR. The numbers report the size of both core components and an example application: a sensor data replicator service. The replicator service consists of the service interface stub for the service as well as the implementation of the service itself. The program loader is currently only implemented on the MSP430 platform.

The code size of Contiki is larger than that of TinyOS [15], but smaller than that of the Mantis system [3]. Contiki's event kernel is significantly larger than that of TinyOS because of the different services provided. While the TinyOS event kernel only provides a FIFO event queue scheduler, the Contiki kernel supports both FIFO events and poll handlers with priorities. Furthermore, the flexibility in Contiki requires more run-time code than for a system like TinyOS, where compile time optimization can be done to a larger extent.

Module	**Code size (AVR)**	**Code size (MSP430)**	**RAM usage**
Kernel	1044	810	10 + + $4e$ + $2p$
Service layer	128	110	0
Program loader	-	658	8
Multi-threading	678	582	8 + s
Timer library	90	60	0
Replicator stub	182	98	4
Replicator	1752	1558	200
Total	3874	3876	230 + $4e$ + + $2p$ + s

Table 7.1: Size of the compiled code, in bytes.

The RAM requirement depends on the maximum number of processes that the system is configured to have (p), the maximum size of the asynchronous event queue (e) and, in the case of multi-threaded operation, the size of the thread stacks (s).

7.9.3 Preemption

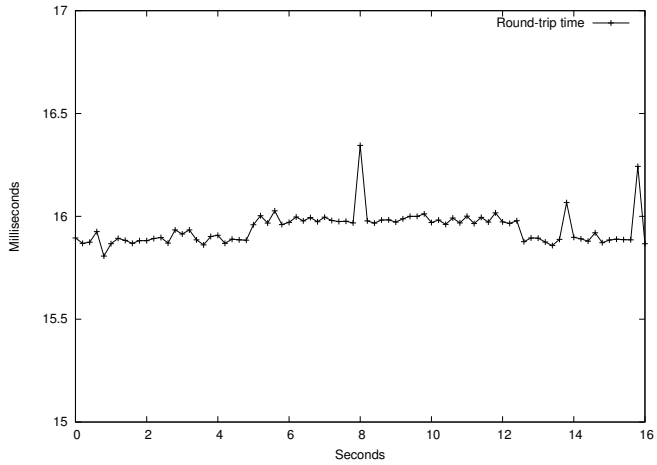

Figure 7.5: A slight increase in response time during a preemptible computation.

The purpose of preemption is to facilitate long running computations while being able to react on incoming events such as sensor input or incoming communication packets. Figure 7.5 shows how Contiki responds to incoming packets during an 8 second computation running in a preemptible thread. The curve is the measured round-trip time of 200 "ping" packets of 40 bytes each. The computation starts after approximately 5 seconds and runs until 13 seconds have passed. During the computation, the round-trip time increases slightly but the system is still able to produce replies to the ping packets.

The packets are sent over a 57600 kbit/s serial line with a spacing of 200 ms from a 1.4 GHz PC to an ESB node running Contiki. The packets are transmitted over a serial line rather than over the wireless link in order to avoid radio effects such as bit errors and MAC collisions. The computation consists of an arbitrarily chosen sequence of multiplications and additions that are repeated for about 8 seconds. The cause for the increase in round-trip time during the computation is the cost of preempting the computation and restoring the kernel context before the incoming packet can be handled. The jitter and the spikes of about 0.3 milliseconds seen in the curve can be contributed to activity in other

poll handlers, mostly the radio packet driver.

7.9.4 Portability

We have ported Contiki to a number of architectures, including the Texas Instruments MSP430 and the Atmel AVR. Others have ported the system to the Hitachi SH3 and the Zilog Z80. The porting process consists of writing the boot up code, device drivers, the architecture specific parts of the program loader, and the stack switching code of the multi-threading library. The kernel and the service layer does not require any changes.

Since the kernel and service layer does not require any changes, an operational port can be tested after the first I/O device driver has been written. The Atmel AVR port was made by ourselves in a couple of hours, with help of publicly available device drivers. The Zilog Z80 port was made by a third party, in a single day.

7.10 Conclusions

We have presented the Contiki operating system, designed for memory constrained systems. In order to reduce the size of the system, Contiki is based on an event-driven kernel. The state-machine driven programming of event-driven systems can be hard to use and has problems with handling long running computations. Contiki provides preemptive multi-threading as an application library that runs on top of the event-driven kernel. The library is *optionally* linked with applications that *explicitly* require a multi-threaded model of computation.

A running Contiki system is divided into two parts: a core and loaded programs. The core consists of the kernel, a set of base services, and parts of the language run-time and support libraries. The loaded programs can be loading and unloading individually, at run-time. Shared functionality is implemented as *services*, a form of shared libraries. Services can be updated or replaced individually, which leads to a very flexible structure.

We have shown that dynamic loading and unloading of programs and services is feasible in a resource constrained system, while keeping the base system lightweight and compact. Even though our kernel is event-based, preemptive multi-threading can be provided at the application layer on a per-process basis.

Because of its dynamic nature, Contiki can be used to multiplex the hardware of a sensor network across multiple applications or even multiple users. This does, however, require ways to control access to the reprogramming facilities. We plan to continue our work in the direction of operating system support for secure code updates.

Bibliography

[1] Berkeley mica motes. Web page. Visited 2004-06-22. URL: http://www.xbow.com/

[2] Eyes prototype sensor node. Web page. Visited 2004-06-22. URL: http://eyes.eu.org/sensnet.htm

[3] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, and R. Han. MANTIS: system support for MultimodAl NeTworks of In-Situ sensors. In *Proc. WSNA'03*, 2003.

[4] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative Task Management Without Manual Stack Management. In *Proc. USENIX*, 2002.

[5] CST Group at FU Berlin. Scatterweb Embedded Sensor Board. Web page. 2003-10-21. URL: http://www.scatterweb.com/

[6] M. Barabanov. A Linux-based RealTime Operating System. Master's thesis, New Mexico Institute of Mining and Technology, 1997.

[7] R. Barr, J. C. Bicket, D. S. Dantas, B. Du, T. W. D. Kim, B. Zhou, and E. Sirer. On the need for system-level support for ad hoc and sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(2), 2002.

[8] A. Boulis, C. Han, and M. B. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. In *Proc. MOBISYS'03*, May 2003.

[9] D. Estrin (editor). *Embedded everywhere: A research agenda for networked systems of embedded computers*. National Academy Press, 2001.

[10] D. R. Engler and M. F. Kaashoek. Exterminate all operating system abstractions. In *Proc. HotOS-V*, May 1995.

[11] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proc SOSP '95*, December 1995.

[12] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proc. SIGPLAN'03*, 2003.

[13] L. Girod, J. Elson, A. Cerpa, T. Stathopoulos, N. Ramanathan, and D. Estrin. EmStar: A Software Environment for Developing and Deploying Wireless Sensor Networks. In *Proc. USENIX*, 2004.

[14] G. Hamilton and P. Kougiouris. The spring nucleus: A microkernel for objects. In *Proc. Usenix Summer Conf.*, 1993.

[15] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proc. ASPLOS-IX*, November 2000.

[16] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. T. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE JSAC*, 14(7):1280–1297, 1996.

[17] P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. In *Proc. ASPLOS-X*, October 2002.

[18] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler. The Emergence of Networking Abstractions and Techniques in TinyOS. In *Proc. NSDI*, 2004.

[19] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proc. NSDI*, 2004.

[20] V. Raghunathan, C. Schurgers, S. Park, and M. Srivastava. Energy aware wireless microsensor networks. *IEEE Signal Processing Magazine*, 19(2):40–50, 2002.

[21] N. Reijers and K. Langendoen. Efficient code distribution in wireless sensor networks. In *Proc. WSNA'03*, 2003.

[22] F. Stajano. *Security for Ubiquitous Computing*. Wiley, 2002.

[23] M. Welsh and G. Mainland. Programming Sensor Networks Using Abstract Regions. In *Proc. NSDI*, 2004.

Chapter 8

Paper C: Connecting Wireless Sensornets with TCP/IP Networks

Adam Dunkels, Thiemo Voigt, Juan Alonso, Hartmut Ritter, and Jochen Schiller. Connecting Wireless Sensornets with TCP/IP Networks. In *Proceedings of the Second International Conference on Wired/Wireless Internet Communications (WWIC2004)*, Frankfurt (Oder), Germany, February 2004.

Abstract

Wireless sensor networks are based on the collaborative efforts of many small wireless sensor nodes, which collectively are able to form networks through which sensor information can be gathered. Such networks usually cannot operate in complete isolation, but must be connected to an external network through which monitoring and controlling entities can reach the sensornet. As TCP/IP, the Internet protocol suite, has become the de-facto standard for large-scale networking, it is interesting to be able to connect sensornets to TCP/IP networks. In this paper, we discuss three different ways to connect sensor networks with TCP/IP networks: proxy architectures, DTN overlays, and TCP/IP for sensor networks. We conclude that the methods are in some senses orthogonal and that combinations are possible, but that TCP/IP for sensor networks currently has a number of issues that require further research before TCP/IP can be a viable protocol family for sensor networking.

8.1 Introduction

Wireless sensor networks is an information gathering paradigm based on the collective efforts of many small wireless sensor nodes. The sensor nodes, which are intended to be physically small and inexpensive, are equipped with one or more sensors, a short-range radio tranciever, a small micro-controller, and a power supply in the form of a battery.

Sensor network deployments are envisioned to be done in large scales, where each network consists of hundreds or even thousands of sensor nodes. In such a deployment, human configuration of each sensor node is usually not feasible and therefore self-configuration of the sensor nodes is important. Energy efficiency is also critical, especially in situations where it is not possible to replace sensor node batteries. Battery replacement maintenance is also important to minimize for deployments where battery replacement is possible.

Most sensor network applications aim at monitoring or detection of phenomena. Examples include office building environment control, wild-life habitat monitoring [17], and forest fire detection [24]. For such applications, the sensor networks cannot operate in complete isolation; there must be a way for a monitoring entity to gain access to the data produced by the sensor network. By connecting the sensor network to an existing network infrastructure such as the global Internet, a local-area network, or a private intranet, remote access to the sensor network can be achieved. Given that the TCP/IP protocol suite has become the de-facto networking standard, not only for the global Internet but also for local-area networks, it is of particular interest to look at methods for interconnecting sensor networks and TCP/IP networks. In this paper, we discuss a number of ways to connect sensor networks to TCP/IP networks.

Sensor networks often are intended to run specialized communication protocols, thereby making it impossible to directly connect the sensor network with a TCP/IP network. The most commonly suggested way to get the sensor network to communicate with a TCP/IP network is to deploy a proxy between the sensor network and the TCP/IP network. The proxy is able to communicate both with the sensors in the sensor network and hosts on the TCP/IP network, and is thereby able to either relay the information gathered by the sensors, or to act as a front-end for the sensor network.

Delay Tolerant Networking (DTN) [9] is a recently proposed communication model for environments where the communication is characterized by long or unpredictable delays and potentially high bit-error rates. Examples include mobile networks for inaccessible environments, satellite communication, and certain forms of sensor networks. DTN creates an overlay network on top of

the Internet and uses late address binding in order to achieve independence of the underlying bearer protocols and addressing schemes. TCP/IP and sensor network interconnection could be done by using a DTN overlay on top of the two networks.

Finally, by directly running the TCP/IP protocol suite in the sensor network, it would be possible to connect the sensor network and the TCP/IP network without requiring proxies or gateways. In a TCP/IP sensor network, sensor data could be sent using the best-effort transport protocol UDP, and the reliable byte-stream transport protocol TCP would be used for administrative tasks such as sensor configuration and binary code downloads.

Due to the power and memory restrictions of the small 8-bit micro-controllers in the sensor nodes, it is often assumed that TCP/IP is not possible to run in sensor networks. In previous work [8], we have shown that this is not true; even small micro-sensor nodes are able to run a full instance of the TCP/IP protocol stack. We have also successfully implemented our small uIP TCP/IP stack [7] on the small sensor nodes developed at FU Berlin [1]. There are, however, a number of problems that needs to be solved before TCP/IP can be a viable alternative for sensor network communication.

The rest of the paper is structured as follows. We discuss proxy architectures in Section 8.2, followed by a discussion of the DTN architecture in Section 8.3. TCP/IP for sensor networks is presented in Section 8.4, and a comparison of the three methods is given in Section 8.5. Finally, the paper is concluded in Section 8.6.

8.2 Proxy Architectures

Deploying a special proxy server between the sensor network and the TCP/IP network is a very simple and straightforward way to connect the two networks. In its simplest form, the proxy resides as a custom-made program running on a gateway computer which has access to both the sensor network and the TCP/IP network. Since all interaction between clients in the TCP/IP network and the sensor nodes is done through the proxy, the communication protocol used in the sensor network may be chosen freely.

The proxy can operate in either of two ways: as a relay, or as a front-end. In the first case, the proxy will simply relay data coming from the sensor network to clients on the TCP/IP network. The clients must register a particular *data interest* with the proxy, and the proxy will then relay data from the sensor network to the registered clients.

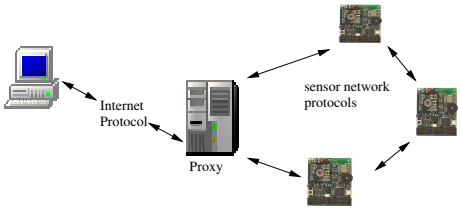

Figure 8.1: Proxy architecture

In the second case, where the proxy acts as a front-end for the sensor network, the proxy pro-actively collects data from the sensors and stores the information in a database. The clients can query the proxy for specific sensor data in a variety of ways, such as through SQL-queries or web-based interfaces.

One advantage of the proxy based approach to interconnect sensor and TCP/IP networks is that the proxy completely decouples the two networks. This naturally allows for specialized communication protocols to be implemented in the sensor network. A front-end proxy can also be used to implement security features such as user and data authentication.

Among the drawbacks of the proxy approach are that it creates a single point of failure. If the proxy fails, all communication to and from the sensor network is effectively made impossible. One possible solution would be to deploy redundancy in the form of a set of back-up proxies. Unfortunately, such a solution reduces the simplicity of the proxy approach. Other drawbacks are that a proxy implementation usually is specialized for a specific task or a particular set of protocols. Such a proxy implementation requires special proxies for each application. Also, no general mechanism for inter-routing between proxies exist.

Proxies have previously been used for connecting devices to TCP/IP networks in order to overcome limitations posed by the devices themselves, or limitations caused by the communication environment in which the devices are located. The Wireless Application Protocol (WAP) stack [15] is intended to be simpler than the TCP/IP protocol stack in order to run on smaller devices, and to be better suited to wireless environments. WAP proxies are used to connect WAP devices with the Internet. Similarly, the Remote Socket Architecture [23] exports the BSD socket interface to a proxy in order to outperform ordinary TCP/IP for wireless links.

8.3 Delay Tolerant Networks

The Delay Tolerant Network architecture [9] is intended for so-called *challenged environments*. Properties of such environments include long and variable delays, frequent network partitioning, potentially high bit-error rates and asymmetrical data rates. DTN is based on the observation that the TCP/IP protocol suite is built around a number of implicit assumptions that do not hold true in challenged communication environments. In particular, the underlying assumptions of TCP/IP are:

- An end-to-end path must exist between source and destination during the whole data exchange.
- The maximum round trip-time for packets must be relatively small and stable.
- The end-to-end packet loss is relatively small.

The DTN architectural design contains several principles to provide service in these environments:

- DTN uses an overlay architecture based on store-and-forward message switching. The messages, called *bundles*, that are transmitted contain both user data and relevant meta-data. A message-switched architecture provides the advantage of a priori knowledge of the size and performance requirements of the data transfer. The bundle layer works as an application layer on top the TCP/IP protocol stack.
- The base transfer between nodes relies on store-and-forward techniques, i.e., a packet is kept until it can be sent to the next hop. This requires that every node has storage available in the network. Furthermore, this allows to advance the point of retransmission towards the destination.

A DTN consists a set of *regions* which share a common layer called the *bundle layer* that resides above the transport layer. The bundle layer stores messages in persistent storage if there is no link available, fragments messages if necessary, and optionally implements end-to-end reliability. The layers below the bundle layer are not specified by the architecture, but are chosen dynamically based on the specific communication characteristics and the available protocols in each region. One or more DTN gateways exist in each DTN region. The DTN gateway forwards bundles between regions, and takes care of delivering messages from other regions to hosts within the local region.

The DTN architecture has been designed with the sensor network paradigm in mind. In sensor networks, the network may be partitioned frequently when nodes go into sleep mode or because of node failure. This will disrupt any end-to-end paths through the network. Also, packet loss rates in sensor networks can be very high [28] and routes may be asymmetric.

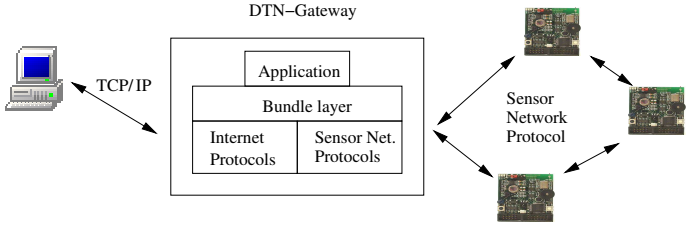

Figure 8.2: Connecting using the DTN architecture

When connecting sensor networks to a TCP/IP network using the DTN architecture, we have at least two regions as depicted in Figure 8.2: one TCP/IP region where the TCP/IP protocol suite is used and one sensor network region where specialized sensor network protocols are implemented. A DTN gateway node is put in between the two networks, similar to where a proxy would have been placed.

The DTN gateway acts much as a relay proxy as discussed in the previous section, and the relay proxy approach can be viewed as a specific instance of the DTN architecture. The DTN architecture is much more general than a simple proxy based approach, however, as the DTN architecture even allows mapping the sensor network into more than one DTN region, with DTN gateways located within the sensor network. For sensor networks where network partitioning is frequent, or where end-to-end communication is impossible, such a network design would be appropriate. A fully DTN enabled sensor network would easily be extended to a TCP/IP network, simply by connecting one or more of the DTN gateways to the TCP/IP network.

8.4 TCP/IP for Sensor Networks

Directly employing the TCP/IP protocol suite as the communication protocol in the sensor network would enable seamless integration of the sensor network and any TCP/IP network. No special intermediary nodes or gateways would be needed for connecting a sensor network with a TCP/IP network. Rather, the

connection would simply be done by connecting one or more sensor nodes to the TCP/IP network. TCP/IP in the sensor network would also provide the possibility to route data to and from the sensor network over standard technologies such as General Packet Radio Service (GPRS) [4]. This leads to an architecture as shown in Figure 8.3.

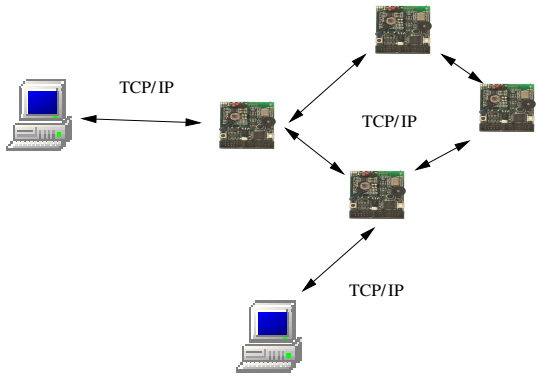

Figure 8.3: Connecting using TCP/IP in the sensor network

Until recently, many believed that tiny sensor nodes would lack the necessary memory and computational resources to be able to run a full instance of the TCP/IP protocol stack. Therefore, the idea of using TCP/IP for sensor networks has not been given much research attention. We have showed that a full TCP/IP stack indeed can be run even on very limited devices [8], and have implemented our small uIP TCP/IP implementation [7] on the sensor nodes developed at FU Berlin [1]. These nodes are equipped with an 8-bit Texas Instruments MSP430 low-power micro-controller with a built-in memory of 2048 bytes. Our TCP/IP implementation requires only a few hundreds bytes of memory to operate, which leaves plenty of memory for the actual sensor node applications.

The fact that we are able to run the TCP/IP stack even on tiny sensor nodes suggest that TCP/IP for sensor networks may be within reach. Sensor networks running the TCP/IP protocol suite would be very easy to connect to existing TCP/IP networks, and would also able to benefit from the wealth of readily available applications such as file transfers using FTP or HTTP and possibly time synchronization with NTP. There are, however, a number of problems with using TCP/IP for wireless sensor networks that need to be addressed before TCP/IP is a viable alternative for sensor networks:

- The addressing and routing schemes of IP are host-centric.
- The header overhead in TCP/IP is very large for small packets.
- TCP does not perform well over links with high bit-error rates, such as wireless links.
- The end-to-end retransmissions used by TCP consumes energy at every hop of the retransmission path.

IP is designed so that every network interface connected to a network has its own IP address. The prefix of the address is the same for all network interfaces in the same physical network and routing is done based on the network prefixes. This does not fit well with the sensor network paradigm, where the main interest is the data generated by the sensors and the individual sensor is of minor importance. Most of the proposed communication protocols for sensor networks use data centric routing and addressing [10, 12] and even though similar mechanisms have been developed as overlay networks on top of IP [21], these usually require too much state to be kept in the participating nodes to be feasible to run on limited sensor nodes.

The size of TCP/IP packet headers is between 28 and 40 bytes, and when sending a few bytes of sensor data in a datagram the headers constitute nearly 90% of each packet. Energy efficiency is of prime importance for sensor networks, and since radio transmission often is the most energy consuming activity in a sensor node [20], a header overhead of 90% is not acceptable. Hence, most protocols developed for sensor networks strive to keep the header overhead as low as possible. For example, the TinyOS [11] message header overhead is only 5%. The header overhead in TCP/IP can be reduced using various forms of header compression [13, 6, 16, 5]. These mechanisms are commonly designed to work only over a single-hop link, but work is currently being done in trying to adopt these mechanisms to the multi-hop case [19].

Furthermore, since TCP was designed for wired networks where bit-errors are uncommon and where packet drops nearly always are due to congestion, TCP always interprets packet drops as a sign of congestion and reduces its sending rate in response to a dropped packet. This leads to bad performance over wireless links where packets frequently are dropped because of bit-errors. TCP misinterprets the packet loss as congestion and lowers the sending rate, even though the network is not congested.

Also, TCP uses end-to-end retransmissions, which in a multi-hop sensor network requires a transmission by every sensor node on the path from the

sender to the receiver. Such a retransmission consumes more energy than a retransmission scheme where the point of retransmission is moved closer to the receiver. Protocols using other mechanisms to implement reliability, such as reliable protocols especially developed for sensor networks [22, 27, 26], are typically designed to be energy conserving.

Methods for improving TCP performance in wireless networks have been proposed [2, 3, 14], but these are often targeted towards the case where the wireless link is the last-hop, and not for wireless networks with multiple wireless hops. In addition, traditional methods assume that the routing nodes have significantly larger amounts of resources than what limited sensor nodes have.

8.5 Comparison of the Methods

The three methods for connecting sensor networks to TCP/IP networks presented here are in some respects orthogonal—it is possible to make combinations such as a partially TCP/IP-based sensor network with a DTN overlay connected to the global Internet using an front-end proxy. It is therefore not possible to make a direct comparison of the methods. Instead, we will state the merits and drawbacks of each of the methods and comment on situations in which each method is suited.

A pure proxy method works well when the sensor network is deployed relatively close to a place where a proxy server can be safely placed. Since the proxy server by design must have more processing power and more memory than the sensors, it is likely to require an electrical power supply rather than a battery. Also, the proxy may need to be equipped with a stable storage media such as a hard disk, which may make the proxy physically larger than the sensor nodes. One example of a situation where these criteria are met is an office building environment. Here, a proxy server can be placed close to the sensor network, perhaps even in the same room as the sensors, and have immediate access to electrical power. Another example would be a nautical sensor network where the proxy could be equipped with a large battery pack and placed in the water with a buoy such that the significance of the physical size of the proxy node would be reduced.

Front-end proxies can also be used for a number of other things, besides for achieving interconnectivity, such as sensor network status monitoring, and generation of sensor failure reports to human operators.

The DTN architecture can be viewed as a generalization of the proxy architecture and indeed a DTN gateway shares many properties with a proxy server.

A DTN gateway in the sensor network region will be placed at the same place as a proxy server would have been placed, and also requires more memory and stable storage media than the sensor nodes. There are, however, a number of things that are gained by using the DTN architecture rather than a simple proxy architecture. First, DTN inherently allows for multiple DTN gateways in a DTN region, which removes the single-point-of-failure problem of the simple proxy architecture. Second, while a proxy architecture usually is specialized for the particular sensor network application, DTN provides general mechanisms and an interface that can be used for a large number of occasions. Also, if the sensor network is deployed in a place with a problematic communication environment, the DTN architecture provides a set of features which can be used to overcome the communication problems. Examples of such situations would be deep-sea exploration or places where seismic activity can disrupt communication.

From an interconnectivity perspective, running native TCP/IP in the sensor networks is the most convenient way to connect the sensor network with a TCP/IP network. One or more sensor nodes would simply be attached to the TCP/IP network, and the two networks could exchange information through any of those nodes. The attachment can be done either using a direct physical link, such as an Ethernet cable, or over a wireless technology like GPRS.

While a TCP/IP enabled sensor network may provide the easiest way to interconnect the networks, it is usually not a complete solution, but must be integrated into a larger architecture. The proxy and DTN architectures discussed here are examples of such an architecture. We can e.g. imagine an office building TCP/IP sensor network that is connected to a front-end proxy located in the cellar of the building. The connection between the proxy and the sensor network would be made using the regular TCP/IP local-area network in the building. Another example would be a TCP/IP sensor network for monitoring the in-door environment in a train. A DTN gateway would be placed in the same train, and the sensor network and the gateway would communicate using TCP/IP over the train's local area network. The DTN gateway would be able to transmit the gathered information over the global Internet at places where the train has Internet access.

Finally, from a security perspective, the front-end proxy architecture provides a good place to implement user and data authentication, since all access to the sensor network goes through the proxy. The DTN architecture is inherently designed for security and uses asymmetric cryptography to authenticate both individual messages and routers. TCP/IP as such does not provide any security, so security must be implemented externally either by using a front-end

proxy, DTN, or any of the existing security mechanisms for TCP/IP networks such as Kerberos. It should also be noted that security methods developed especially with wireless sensor networks in mind [18, 25] can be implemented as application layer security in TCP/IP sensor networks.

8.6 Conclusions

We have presented three methods for connecting wireless sensornets with TCP/IP networks: proxy architectures, Delay Tolerant Networking (DTN) overlays, and TCP/IP for sensor networks. The three methods are orthogonal in that it is possible to form combinations, such as a DTN overlay on top of a TCP/IP sensor network behind a front-end proxy.

The proxy architectures are simple and make it possible to use specialized communication protocols in the sensor network, but are application specific and creates a single point of failure. The DTN architecture also allows for specialized protocols, but provides a much more general communication architecture. DTN is also useful if the sensor network itself is deployed in a challenged communication environment.

Finally, by using the TCP/IP protocol suite for the sensor network, connecting the sensor network with another TCP/IP network is simply done by attaching one or more sensor nodes to both networks. However, attaching the sensor nodes to the TCP/IP network may not always be ideal, and a combination of either a proxy architecture and TCP/IP, or DTN and TCP/IP, may be beneficial.

TCP/IP for sensor networks currently has a number of problems, and therefore further research in the area is needed before TCP/IP can be a viable alternative for sensor networking.

Bibliography

[1] CST Group at FU Berlin. Scatterweb Embedded Sensor Board. Web page. 2003-10-21. URL: http://www.scatterweb.com/

[2] H. Balakrishnan, S. Seshan, E. Amir, and R. Katz. Improving TCP/IP performance over wireless networks. In *Proceedings of the first ACM Conference on Mobile Communications and Networking*, Berkeley, California, November 1995.

[3] K. Brown and S. Singh. M-TCP: TCP for mobile cellular networks. *ACM Computer Communications Review*, 27(5):19–43, October 1997.

[4] J. Cai and D. Goodman. General packet radio service in GSM. *IEEE Communications Magazine*, 35:122–131, October 1997.

[5] S. Casner and V. Jacobson. Compressing IP/UDP/RTP headers for low-speed serial links. RFC 2508, Internet Engineering Task Force, February 1999.

[6] M. Degermark, M. Engan, B. Nordgren, and S. Pink. Low-loss TCP/IP header compression for wireless networks. *ACM/Baltzer Journal on Wireless Networks*, 3(5), 1997.

[7] A. Dunkels. uIP - a TCP/IP stack for 8- and 16-bit microcontrollers. Web page. 2003-10-21. URL: http://dunkels.com/adam/uip/

[8] A. Dunkels. Full TCP/IP for 8-bit architectures. In *Proceedings of The First International Conference on Mobile Systems, Applications, and Services (MOBISYS '03)*, San Francisco, California, May 2003.

[9] K. Fall. A delay-tolerant network architecture for challenged internets. In *Proceedings of the SIGCOMM'2003 conference*, 2003.

[10] J. Heidemann, F. Silva, C. Intanagonwiwat, R. Govindan, D. Estrin, and D. Ganesan. Building efficient wireless sensor networks with low-level naming. In *Proceedings of the Symposium on Operating Systems Principles*, pages 146–159, Chateau Lake Louise, Banff, Alberta, Canada, October 2001. ACM.

[11] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[12] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *Mobile Computing and Networking*, pages 56–67, 2000.

[13] V. Jacobson. Compressing TCP/IP headers for low-speed serial links. RFC 1144, Internet Engineering Task Force, February 1990.

[14] J. Liu and S. Singh. ATCP: TCP for mobile ad hoc networks. *IEEE Journal on Selected Areas in Communications*, 19(7):1300–1315, 2001.

[15] Wireless Application Protocol Forum Ltd. *Official Wireless Application Protocol: The Complete Standard*. Wiley Computer Publishing, 2000.

[16] S. Pink M. Degermark, B. Nordgren. IP header compression. RFC 2507, Internet Engineering Task Force, February 1999.

[17] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless sensor networks for habitat monitoring. In *First ACM Workshop on Wireless Sensor Networks and Applications (WSNA 2002)*, Atlanta, GA, USA, September 2002.

[18] A. Perrig, R. Szewczyk, V. Wen, D. E. Culler, and J. D. Tygar. SPINS: security protocols for sensor netowrks. In *Mobile Computing and Networking*, pages 189–199, 2001.

[19] S. Mishra R. Sridharan, R. Sridhar. A robust header compression technique for wireless ad hoc networks. In *MobiHoc 2003*, Annapolis, MD, USA, June 2003.

[20] V. Raghunathan, C. Schurgers, S. Park, and M. Srivastava. Energy aware wireless microsensor networks. *IEEE Signal Processing Magazine*, 19(2):40–50, March 2002.

[21] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *Proceedings of ACM SIGCOMM 2001*, 2001.

[22] Y. Sankarasubramaniam, O. Akan, and I. Akyildiz. ESRT : Event-to-Sink Reliable Transport in Wireless Sensor Networks. In *Proceedings of the 4th ACM international symposium on Mobile ad hoc networking and computing (MobiHOC 2003)*, 2003.

[23] M. Schläger, B. Rathke, A. Wolisz, and S. Bodenstein. Advocating a remote socket architecture for internet access using wireless lans. *Mobile Networks and Applications*, 6(1):23–42, 2001. ISSN: 1383-469X

[24] S. N. Simic and S. Sastry. Distributed environmental monitoring using random sensor networks. In *Proceedings of the 2nd International Workshop on Information Processing in Sensor Networks*, pages 582–592, Palo Alto, California, 2003.

[25] F. Stajano. *Security for Ubiquitous Computing*. John Wiley and Sons, February 2002. ISBN: 0-470-84493-0

[26] F. Stann and J. Heidemann. RMST: Reliable Data Transport in Sensor Networks. In *Proceedings of the First International Workshop on Sensor Net Protocols and Applications*, pages 102–112, Anchorage, Alaska, USA, April 2003. IEEE.

[27] C.Y. Wan, A. T. Campbell, and L. Krishnamurthy. PSFQ: A Reliable Transport Protocol For Wireless Sensor Networks. In *First ACM International Workshop on Wireless Sensor Networks and Applications (WSNA 2002)*, Atlanta, September 2002.

[28] J. Zhao and R. Govindan. Understanding packet delivery performance in dense wireless sensor networks. In *The First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003)*, Los Angeles, California, November 2003.

Chapter 9

Paper D: Making TCP/IP Viable for Wireless Sensor Networks

Adam Dunkels, Thiemo Voigt, and Juan Alonso. Making TCP/IP Viable for Sensor Networks. In *Proceedings of the First European Workshop on Wireless Sensor Networks (EWSN2004), work-in-progress session*, January 2004.

Abstract

The TCP/IP protocol suite, which has proven itself highly successful in wired networks, is often claimed to be unsuited for wireless micro-sensor networks. In this work, we question this conventional wisdom and present a number of mechanisms that are intended to enable the use of TCP/IP for wireless sensor networks: spatial IP address assignment, shared context header compression, application overlay routing, and distributed TCP caching (DTC). Sensor networks based on TCP/IP have the advantage of being able to directly communicate with an infrastructure consisting either of a wired IP network or of IP-based wireless technology such as GPRS. We have implemented parts of our mechanisms both in a simulator environment and on actual sensor nodes. Our preliminary results are promising.

9.1 Introduction

Many wireless sensor networks cannot be operated in isolation; the sensor network must be connected to an external network through which monitoring and controlling entities can reach the sensor network. The ubiquity of TCP/IP has made it the de-facto standard protocol suite for wired networking. By running TCP/IP in the sensor network it is possible to directly connect the sensor network with a wired network infrastructure, without proxies or middle-boxes [4]. It is often argued that the TCP/IP protocol stack is unsuited for sensor networks because of the specific requirements and the extreme communication conditions that sensor networks exhibit. We believe, however, that by using a number of optimization mechanisms, it is possible to achieve similar performance in terms of energy consumption and data throughput with TCP/IP as that obtained by using specialized communication protocols, while at the same time benefiting from the ease of interoperability and generality of TCP/IP.

We envision that data transport in a *TCP/IP sensor network* is done using the two main transport protocols in the TCP/IP stack: the best-effort UDP and the reliable byte-stream TCP. Sensor data and other information that do not require reliable transmission is sent using UDP, whereas TCP is used for administrative tasks that require reliability and compatibility with existing application protocols. Examples of such administrative tasks are configuration and monitoring of individual sensor nodes, and downloads of binary code or data aggregation descriptions to sensor nodes.

The contribution of this paper are our innovative solutions to the following problems with TCP/IP for sensor networks:

IP addressing architecture. In ordinary IP networks, IP addresses are assigned to each network interface that is connected to the network. Address assignment is done either using manual configuration or a dynamic mechanism such as DHCP. In a large scale sensor network, manual configuration is not feasible and dynamic methods are usually expensive in terms of communication. Instead, we propose a *spatial IP address assignment* scheme that provides semi-unique IP addresses to sensor nodes.

Header overhead. The protocols in the TCP/IP suite have a very large header overhead, particularly compared to specialized sensor network communication protocols. We believe that the shared context nature of sensor networks makes *header compression* work well as a way to reduce the TCP/IP header overhead.

Address centric routing. Routing in IP networks is based on the addresses of the hosts and networks. The application specific nature of sensor networks

makes the use of data-centric routing mechanisms [5] preferable over address-centric mechanisms, however. We propose a specific form of an *application overlay network* to implement data-centric routing and data aggregation for TCP/IP sensor networks.

Limited nodes. Sensor nodes are typically limited in terms of memory and processing power. It is often assumed that the TCP/IP stack is too heavy-weight to be feasible for such small systems. In previous work [3], we have shown that this is not the case but that an implementation of the TCP/IP stack in fact can be run on 8-bit micro-controllers with only a few hundred bytes of RAM.

TCP performance and energy inefficiency. The reliable byte-stream protocol TCP has been shown to have serious performance problems in wireless networks [2]. Moreover, the end-to-end acknowledgment and retransmission scheme employed by TCP causes expensive retransmissions along every hop of the path between the sender and the receiver, if a packet is dropped. We have developed a distributed mechanism similar to TCP snoop [2] that we believe can be used to overcome both problems.

While we are not aware of any research on TCP/IP for wireless sensor networks, there is a plethora of work being done on TCP/IP for mobile ad-hoc networks (MANETs). There are, however, a number of differences between sensor networks and MANETs that affect the applicability of TCP/IP. MANET nodes are operated by human users, whereas sensor networks are intended to be autonomous. The user-centricity of MANETs makes throughput the primary performance metric, while the per-node throughput in sensor networks is inherently low because of the limited capabilities of the nodes. Instead, energy consumption is the primary concern in sensor networks. Finally, TCP throughput is reduced by mobility [6], but nodes in sensor networks are usually not as mobile as MANET nodes.

In Sections 9.2 through 9.6 we describe our proposed solutions to the above problems and report on preliminary results. Finally, Section 9.7 concludes the paper and presents the direction of our future work.

9.2 Spatial IP Address Assignment

For most sensor networks, the data generated by the sensor nodes needs to be associated with the spatial location where the data was sensed. It is therefore a reasonable assumption that the nodes in a sensor network have some way of determining their location, and methods for localization in sensor networks

have been developed [14].

For TCP/IP sensor networks, we propose a *spatial IP address assignment* mechanism to solve the problem of address assignment. With spatial IP address assignment, each sensor node uses its spatial location to construct an IP address. Since we assume that the nodes are aware of their own spatial location, the address assignment requires neither a central server nor communication between the sensor nodes.

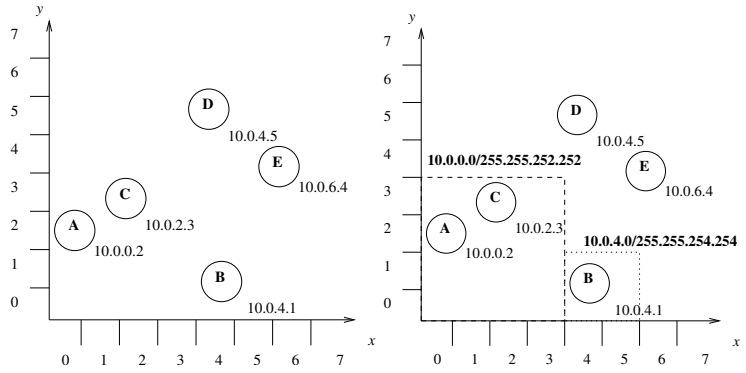

Figure 9.1: Example spatial IP address assignment and two regional subnets.

Figure 9.1 shows an example network with spatially assigned IP addresses. In this particular network, each sensor has constructed its IP address by taking the (x, y) coordinates of the node as the two least significant octets in the IP address. We do not intend to specify the specific way that the addresses are constructed, but assume that it will vary between different kind of sensor networks.

Because location information is encoded in the IP addresses, we can define a *regional subnet* as a set of sensor nodes that share a prefix (Figure 9.1) and implement a straightforward *regional broadcast* mechanism, analogous to ordinary IP subnet broadcasts. This mechanism does not require a special mapping between logical and physical location as needed, e.g., in GeoCast [10].

The spatially assigned IP addresses are not guaranteed to be unique, since two or more adjacent sensor nodes may obtain the same location coordinates and thereby construct the same address. Nodes with duplicate addresses are in the proximity of each other, however, which helps to avoid routing problems; nodes with duplicate addresses are likely to share large parts of routing paths

towards the nodes. Transport layer port number conflicts for sensors that are able to overhear each other's radio communication can be resolved by passive monitoring of the neighbors' communication.

9.3 Header Compression

Energy is often the most scarce resource in wireless sensor networks, and for many applications radio transmission is the most expensive activity [12]. The minimum size of a UDP/IP header is 28 bytes and a 4 bytes sensor data value sent using using UDP/IP has a 87.5% header overhead, which cause large amounts of energy to be spent in transmitting the header.

In sensor networks, all sensor nodes are assumed to cooperate towards a common goal, and therefore the nodes share a common context. For that reason, all nodes can agree on specific UDP/IP header field values for sensor data UDP datagrams. The headers can then be compressed using simple pattern-matching techniques. For example, since all nodes are part of the same IP subnet, there is no need to transmit full IP addresses in the headers of packets that are sourced from or are destined to nodes in the sensor network. Similarly, by utilizing only a small range of UDP ports for the sensor data datagrams, transmitting full 16-bit port numbers is not required for packets containing sensor data.

For TCP connections, standard header compression techniques [8, 9] can be used, but the specific requirements of the sensor network place additional challenges. For instance, while ordinary TCP header compression may be content with the connection end-points detecting and retransmitting incorrectly decompressed headers, a multi-hop wireless sensor network must perform in-network detection and retransmission in a more aggressive manner because of the energy consumption caused by end-to-end retransmissions. It should also be noted that others are working on multi-hop aware header compression techniques [11] that could be beneficial for TCP/IP sensor networks as well.

9.4 Application Overlay Routing

The spatial IP addressing mechanism provides a way to send IP packets to nodes specified by their spatial location, but a pure IP packet routing scheme cannot readily support data aggregation or attribute based routing. Instead, we believe that application overlay networks may be a good way to implement such mechanisms. At first sight, an overlay network might seem too expensive

for a wireless sensor network, because of the mapping required between the physical network and the overlay network. We argue, however, that by choosing an overlay network that fits well with the underlying physical nature of a sensor network, the mapping is not necessarily expensive.

We believe that UDP datagrams sent using link local IP broadcast [13] is a suitable mechanism for implementing an application overlay network on top of the physical sensor network structure. Link local broadcasts provide a direct mapping between the application overlay and the underlying wireless network topology. By tuning the header compression for the special case of link-local broadcasts, the header overhead of such packets does not need to be significantly larger than that of a broadcast packet directly sent using the physical network interface. Furthermore, link-local application layer broadcasts can also be used to implement both low-level mechanisms such as neighbor discovery and high-level protocols such as Directed Diffusion [7].

In addition to the compatibility aspects, an application layer overlay network also has the benefits of generality in that it can be run transparently over both sensor nodes and regular Internet hosts, without requiring proxies or protocol converters.

9.5 Tiny TCP/IP Implementation

It is often assumed that TCP/IP is too heavy-weight to be feasible to implement on a small system such as a sensor node. We have previously shown [3] that even a small system can run the full TCP/IP protocol stack, albeit with lower performance in terms of throughput. Our uIP TCP/IP implementation [3] occupies only a few kilobytes of code space and requires as little as a few hundreds bytes of memory, and we have successfully ported it to the Embedded Sensor Board (ESB) developed at FU Berlin [1]. The ESB is equipped with a number of sensors, an RF transceiver, and an MSP430 low-power 8-bit micro-controller with 2048 bytes of RAM and 60 kilobytes flash ROM.

9.6 Distributed TCP Caching

The reliable byte-stream TCP was designed for wired networks where bit-errors are uncommon and where congestion is the predominant source of packet drops. Therefore, TCP always interprets packet drops as a sign of congestion and reduces its sending rate in response to a dropped packet. Packet drops

in wireless networks are often due to bit-errors, which leads TCP to misinterpret the packet loss as congestion. TCP will then lower the sending rate, even though the network is not congested.

Furthermore, TCP uses end-to-end retransmissions, which in a multi-hop sensor network requires a retransmitted packet to be forwarded by every sensor node on the path from the sender to the receiver. As Wan et al. note, end-to-end recovery is not a good candidate for reliable transport protocols in sensor networks where error rates are in the range of 5% to 10% or even higher [15]. A scheme with local retransmissions is more appropriate since it is able to move the point of retransmission closer towards the final recipient of the packet.

To deal with these issues, we propose a scheme called *distributed TCP caching* (DTC) that uses segment caching and local retransmissions in cooperation with the link layer. Other mechanisms for improving TCP performance over wireless links, such as TCP snoop [2], focus on improving TCP *throughput*. In contrast, DTC is primarily intended to reduce the *energy consumption* required by TCP. DTC does not require any protocol changes neither at the sender nor at the receiver.

We assume that each sensor node is able to cache only a small number of TCP segments; specifically, we assume that nodes only have enough memory to cache a single segment.

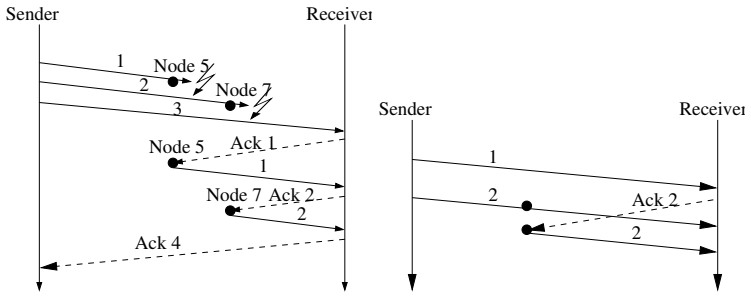

Figure 9.2: Distributed TCP caching (left) and spurious retransmission (right)

The left part of Figure 9.2 shows a simplified example how we intend DTC to work. In this example, a TCP sender transmits three TCP segments. Segment `1` is cached by node 5 right before it is dropped in the network, and segment `2` is cached by node 7 before being dropped. When receiving segment `3`, the TCP receiver sends an acknowledgment (`ACK 1`). When receiving `ACK`

1, node 5, which has a cached copy of segment 1, performs a local retransmission. Node 5 also refrains from forwarding the acknowledgment towards the TCP sender, so that the acknowledgment segment does not have to travel all the way through the network. When receiving the retransmitted segment 1, the TCP receiver acknowledges this segment by transmitting ACK 2. On reception of ACK 2, Node 7 performs a local retransmission of segment 2, which was previously cached. This way, the TCP receiver obtains the two dropped segments by local retransmissions from sensor nodes in the network, without requiring retransmissions from the TCP sender. When the acknowledgment ACK 4 is forwarded towards the TCP sender, sensor nodes on the way can clear their caches and are thus ready to cache new TCP segments.

9.6.1 Segment Caching and Packet Loss Detection

DTC uses segment caching to achieve local retransmissions. Because of the memory limitations of the sensor nodes, it is vital to the performance of the mechanism to find an appropriate way for nodes to select which segments to cache. Initial analysis suggest that a desirable outcome of the selection algorithm is that segments are cached at nodes as close to the receiver as possible, and that nodes closer to the receiver cache segments with lower sequence numbers. To achieve this, each node caches the TCP segment with the highest sequence number seen, and takes extra care to cache segments that are likely to be dropped further along the path towards the receiver. We use feedback from a link layer that supports positive acknowledgments to infer packet drops on the next-hop. A TCP segment that is forwarded but for which no link layer acknowledgment is received may have been lost in transit, and the segment is *locked* in the cache indicating that it should not be overwritten by a TCP segment with a higher sequence number. A locked segment is cleared from the cache only when an acknowledgment that acknowledges the cached segment is received, or when the segment times out.

To avoid retransmissions from the original TCP sender, DTC needs to respond faster to packet drops than regular TCP. DTC uses ordinary TCP mechanisms to detect packet loss: time-outs and duplicate acknowledgments. Every node participating in DTC maintains a soft TCP state for connections that pass through the node. We assume symmetric and relatively stable routes, and therefore the nodes can estimate the delays between the node and the connection end-points. The delays experienced by the nodes are lower than those estimated by the TCP end-points, and the nodes are therefore able to use lower time-out values and perform retransmissions quicker than the connection end-

points.

In TCP, duplicate acknowledgments signal either packet loss or packet reordering. A TCP receiver uses a threshold of three duplicate acknowledgments as a signal of packet loss, which may be too conservative for DTC. Since each DTC node inspects the TCP sequence numbers of forwarded TCP segments, the nodes may be able to compute a heuristic for the amount of packet reordering, and to lower the duplicate acknowledgment threshold if packet reordering is found to be uncommon in the network. Furthermore, care must be taken to avoid spurious retransmissions caused by misinterpreting acknowledgments for new data as acknowledgments that signal packet loss, as shown in the right part of Figure 9.2. The nodes can use estimated round-trip times to distinguish between an acknowledgment that detects a lost packet and one that acknowledges new data.

We are also considering using the TCP SACK option to detect packet loss and also as a signaling mechanism between DTC nodes.

9.6.2 Preliminary Results

We have performed simulations comparing standard TCP with DTC. Our results show vast improvements: For path lengths between 5 and 10 hops and packet loss rates between 5% and 15%, the number of retransmissions that the TCP sender has to perform decreases by a factor of four to eight. For example, with a packet loss rate of 10% for data packets (5% for acknowledgments and 2% for link level acknowledgments), a path length of 10 hops, and with 500 packets to be transmitted the number of required source retransmission decreases from 51 to 6 (averaged over 30 different runs).

In sensor networks, sensor data flows from sources to sinks, whereas control or management data flows from sinks to sources [15]. Therefore, nodes close to the sink usually are the first to run out of energy because sensor data sent towards the sink has to pass them. As shown by our initial simulation results in Figure 9.3, DTC is able to reduce the load at the nodes close to the sink/TCP sender.

We do not yet have any results from the TCP header compression coupled with DTC, but our UDP/IP header compressor is able to reduce UDP/IP headers for sensor data from 28 to three bytes.

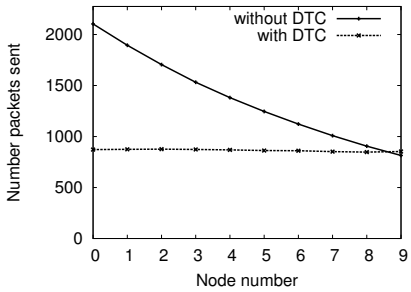

Figure 9.3: DTC load reduction close to sender

9.7 Conclusions and Future Work

In this paper we challenge the assumption that TCP/IP is unsuitable for sensor networks. Our main contributions are a spatial IP address assignment scheme and a mechanism for distributed segment caching called distributed TCP caching.

Future work will be targeted at further development and evaluation of the proposed mechanisms using both simulation and experiments with physical sensor networks. We are currently looking into the interactions between the link layer and header compression mechanisms that work together with DTC. For DTC, we will consider the energy consumption tradeoffs involved with link layers with different levels of reliability. We also intend to compare DTC with transport protocols specifically designed for sensor networks such as PSFQ [15]. Furthermore, we are currently implementing the DTC mechanism on actual sensor nodes in order to measure real-world performance and preliminary results show that the sensor nodes are capable of running both a full TCP/IP stack and the DTC mechanism.

Bibliography

[1] CST Group at FU Berlin. Scatterweb Embedded Sensor Board. Web page. 2003-10-21. URL: http://www.scatterweb.com/

[2] H. Balakrishnan, S. Seshan, E. Amir, and R. Katz. Improving TCP/IP performance over wireless networks. In *MOBICOM'95*, 1995.

[3] A. Dunkels. Full TCP/IP for 8-bit architectures. In *Proceedings of The First International Conference on Mobile Systems, Applications, and Services (MOBISYS '03)*, San Francisco, California, May 2003.

[4] A. Dunkels, T. Voigt, J. Alonso, H. Ritter, and J. Schiller. Connecting Wireless Sensornets with TCP/IP Networks. In *Proceedings of the Second International Conference on Wired/Wireless Internet Communications (WWIC2004)*, Frankfurt (Oder), Germany, February 2004.

[5] D. Estrin, R. Govindan, J. S. Heidemann, and S. Kumar. Next century challenges: Scalable coordination in sensor networks. In *Mobile Computing and Networking*, pages 263–270, 1999.

[6] G. Holland and N. H. Vaidya. Analysis of TCP performance over mobile ad hoc networks. In *MOBICOM '99*, August 1999.

[7] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *Mobile Computing and Networking*, pages 56–67, 2000.

[8] V. Jacobson. Compressing TCP/IP headers for low-speed serial links. RFC 1144, Internet Engineering Task Force, February 1990.

[9] S. Pink M. Degermark, B. Nordgren. IP header compression. RFC 2507, Internet Engineering Task Force, February 1999.

[10] Julio C. Navas and Tomasz Imielinski. GeoCast – geographic addressing and routing. In *Mobile Computing and Networking*, pages 66–76, 1997.

[11] S. Mishra R. Sridharan, R. Sridhar. A robust header compression technique for wireless ad hoc networks. In *MobiHoc 2003*, 2003.

[12] V. Raghunathan, C. Schurgers, S. Park, and M. Srivastava. Energy aware wireless microsensor networks. *IEEE Signal Processing Magazine*, 19(2):40–50, March 2002.

[13] J. Reynolds and J. Postel. Assigned numbers. RFC 1700, Internet Engineering Task Force, October 1994.

[14] Andreas Savvides, Chih-Chieh Han, and Mani B. Strivastava. Dynamic fine-grained localization in ad-hoc networks of sensors. In *Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 166–179. ACM Press, 2001. ISBN: 1-58113-422-3

[15] C.Y. Wan, A. T. Campbell, and L. Krishnamurthy. PSFQ: A Reliable Transport Protocol For Wireless Sensor Networks. In *WSNA'02*, 2002.