# Implementation and Evaluation of the Sensornet Protocol for Contiki

Zhitao He
Swedish Institute of Computer Science, Box 1263, SE-164 29
Kista, Sweden
Email: zhitao@sics.se

SICS Technical Report T2007:14
ISSN 1100-3154, ISRN:SICS-T–2007/14-SE

2007-12-18

**Abstract**

Sensornet Protocol (SP) is a link abstraction layer between the network layer and the link layer for sensor networks. SP was proposed as the core of a future-oriented sensor node architecture that allows flexible and optimized combination between multiple coexisting protocols.

This thesis implements the SP sensornet protocol on the Contiki operating system in order to: evaluate the effectiveness of the original SP services; explore further requirements and implementation trade-offs uncovered by the original proposal.

We analyze the original SP design and the TinyOS implementation of SP to design the Contiki port. We implement the data sending and receiving part of SP using Contiki processes, and the neighbor management part as a group of global routines. The evaluation consists of a single-hop traffic throughput test and a multihop convergecast test. Both tests are conducted using both simulation and experimentation.

We conclude from the evaluation results that SP's link-level abstraction effectively improves modularity in protocol construction without sacrificing performance, and our SP implementation on Contiki lays a good foundation for future protocol innovations in wireless sensor networks.

# Contents


**1 Introduction** **1**
- 1.1 Sensornets Overview . . . 1
- 1.2 The Modularity Issue with Existing Sensornet Protocols . . . 2
- 1.3 The Sensornet Protocol . . . 2
- 1.4 The Contiki Operating System . . . 3
- 1.5 Thesis Overview . . . 3
  - 1.5.1 Results . . . 4
- 1.6 Document Structure . . . 4

**2 Background** **5**
- 2.1 Communication Protocols for Sensornets . . . 5
  - 2.1.1 Link Layer Functionality Overview . . . 5
  - 2.1.2 Network-Layer Protocols for Sensornets . . . 6
  - 2.1.3 The Sensornet Protocol . . . 7
- 2.2 The Contiki Operating System . . . 7
  - 2.2.1 Contiki Processes . . . 7
  - 2.2.2 Protothread . . . 8
  - 2.2.3 Timers and Interrupts . . . 8

**3 Analysis of SP Services** **9**
- 3.1 Data Transfer Service . . . 9
  - 3.1.1 Sending and Receiving Packets Using SP . . . 10
  - 3.1.2 Packet Buffering and Scheduling . . . 10
  - 3.1.3 Protocol Multiplexing . . . 11
- 3.2 Neighbor Management Service . . . 12
  - 3.2.1 SP Neighbor Table . . . 12
  - 3.2.2 Link State Maintenance . . . 13
- 3.3 Applicability of SP . . . 13

**4 Methods** **14**
- 4.1 Programming Patterns . . . 14
- 4.2 Tools . . . 14
  - 4.2.1 NETSIM . . . 15
  - 4.2.2 Telos . . . 15

**5 Implementation** **16**
5.1 Data Transfer Service . . . . . 16
5.1.1 SP Message Pool . . . . . 16
5.1.2 Control and Feedback . . . . . 20
5.2 Neighbor Management Service . . . . . 20
5.2.1 Neighbor Table Operations . . . . . 20
5.2.2 Neighbor Management Policy . . . . . 21

**6 Evaluation** **23**
6.1 Single Hop Evaluation . . . . . 23
6.2 Multihop Networking Using SP . . . . . 23
6.2.1 The Treeroute Protocol . . . . . 24
6.2.2 Porting Treeroute to A Modular Network Layer Architecture . 26
6.2.3 Verification of SP-based Treeroute . . . . . 30
6.2.4 Issues With the NLA Architecture . . . . . 31
6.2.5 An Alternative Network Layer Architecture . . . . . 31

**7 Conclusion and Future Work** **33**
7.1 Conclusion . . . . . 33
7.2 Future Work . . . . . 33

# Chapter 1

# Introduction

## 1.1 Sensornets Overview

Sensornets have emerged as a highly competitive technology for a wide range of existing and future applications, which encompass diverse distributed sensing and controlling tasks in different environments. The main reasons for adoption of sensornets in such applications are low cost, good scalability, and ability of collaborative operation. In a typical sensornet application, tens, hundreds or thousands of sensor nodes are deployed, collaboratively forming a sensornet to perform a common task. A sensornet provides services of data retrieval, event detection, and in-network information processing in an autonomous manner.

A typical sensor node comprises the following components: sensors, microprocessor, radio transceiver, and battery. The sensors monitor real-time physical phenomena in the environment. The microprocessor stores and processes the sensed data. The radio transceiver exchanges data and networking information with neighboring nodes.

A sensornet may consist of a single type of nodes, or heterogeneous nodes with different capacities and functions, depending on requirements and constraints. Moreover, the nodes may form different types of network topologies, such as tree, cluster, mesh or hybrid. Typically, there is a base station node that acts as a data sink as well as a gateway to the user.

Communication and collaboration between sensor nodes require that they follow a set of protocols. Each protocol provides a service for its user and/or regulates communication between its peers. Like other communication protocols, sensornet protocols are often organized as a stack of layers, in which a lower layer provides service to its next higher layer. The protocol stack as a whole forms an abstraction of communication mechanisms that the application is executed upon. Figure 1.1 illustrates a layered protocol stack commonly adopted by network designers.

A sensornet designer is confronted with a number of well known limitations and difficulties. Firstly, sensor node hardware is highly constrained in bandwidth, processing power, memory, and most critically, energy capacity. These precious resources need to be used in an efficient manner to insure normal network operation over a long

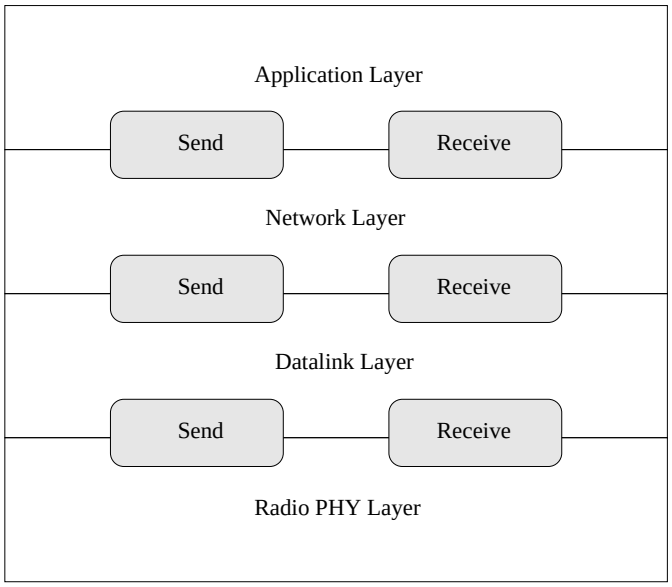


Figure 1.1: Layered Protocol Stack Architecture

lifetime. Secondly, channel fluctuation and network dynamics often bring challenges because a sensornet typically has a large number of nodes spread across a vast geographical area. These uncertainties tend to undermine a sensornet's stability and performance, and put a potential limit on network scalability.

## 1.2 The Modularity Issue with Existing Sensornet Protocols

The application specific requirements and resource constraints forced many earlier sensornet designs to resort to customized protocol stacks that aggressively integrate functionality of the network and datalink layers into a complex monolithic entity, or make arbitrary decisions on layer boundaries. This holistic approach usually aims at achieving optimal results for particular design goals, at the cost of however compromised modularity. Consequently, the vertically integrated system often suffers from blurred component boundaries and tightly coupled operations, which hampers code reuse and interoperability, making future improvement and innovation of sensornet protocols a difficult task.

## 1.3 The Sensornet Protocol

To promote the ease of protocol composition and design reuse, Culler et al. have proposed a sensornet architecture [1]. Drawing analogy to the IP-based Internet archi-

tecture, the authors argues that a similar "narrow waist", i.e., a common service that separate applications above and technologies below as IP does for Internet, would bring greatly improved modularity to sensornets. Such an architecture would be adaptable to future developments at both the application level and device level.

Unlike IP, the proposed "narrow waist" layer, named Sensornet Protocol (SP), is a single hop service that resides above the datalink layer and below the network layer. The lowered waist allows various multihop protocols to take advantage of the common link-level abstraction provided by SP while optimizing their own core functionality. In addition, the architecture includes cross-layer services that are shared among different layers, such as power management and time synchronization.

## 1.4 The Contiki Operating System

Contiki is an operating system for memory-constrained embedded systems developed by SICS [2]. It is an open-source OS written in the C programming language. Its concurrency model provides an event-based inter-process communication mechanism naturally suited for typical sensornet applications. Each Contiki process consists of a *Protothread*, a program control abstraction that hides the yielding points within a multi-state task behind straightforward *wait* statements [3], whereby implementation of communication protocols can be simplified.

## 1.5 Thesis Overview

This thesis's main objective is implementing SP on the Contiki operating system, in order to further evaluate SP as a link abstraction service and explore possible improvement on the existing SP design. We also want to show that sensornet protocols such as SP can be more easily developed using Contiki's programming abstractions.

We perform an analysis of the SP services to capture the salient features of the protocol. We find similarity between SP's data transfer service and IEEE 802.2 [4]. On the other hand, SP's neighbor management service provides additional support for link state maintenance required by low power MACs.

We implement the SP components and primitives using a combination of ordinary routines and Contiki processes and events. We experiment with a number of data structures to implement the SP message pool and compare their advantages and disadvantages in terms of functionality and complexity. We implement the SP management service as a set of globally accessible routines to enhance information sharing.

To show how a link-level protocol interacts with SP, we adapt a simple link protocol that sends individual packets using Automatic Repeat Request (ARQ) and switches the radio according to a preconfigured sleep/wake up scheme. We evaluate SP's throughput performance by running a traffic generation program on both a network simulator and a hardware platform.

We also port the Treeroute protocol that combines convergecast and dissemination to two SP-based network-layer architectures. We examine the feasibility of SP

to support higher-layer protocols, and explore the implications the SP interface has on composition of higher-layer protocols.

### 1.5.1 Results

Our evaluation of the experimental results reveals that: SP's communication mechanism provides good link-level abstraction without sacrificing link performance, and our implementation of SP on Contiki supports multiple coexisting network layer protocols with better modularity than the integrated approach, and may bring potential saving in code size. Finally, our work may serve as a solid basis for future sensornet protocol development, in particular on the Contiki operating system.

## 1.6 Document Structure

The rest of this thesis is structured as following: Chapter 2 provides background information about sensor network protocols and Contiki; Chapter 3 gives an analysis of SP services and interfaces; Chapter 4 outlines the methods used to carry out the work; Chapter 5 provides details of the implementation of SP services on Contiki; Chapter 6 presents evaluation results from single hop communication tests, and discuss the porting of the Treeroute protocol to SP; Chapter 7 draws our conclusion and outlines future work.

# Chapter 2

# Background

In the chapter We go over first the general communication design problem in sensor networks, followed by a description of Contiki's supporting features for implementation of communication protocols.

## 2.1 Communication Protocols for Sensornets

We focus our considerations on the network layer and the datalink layer of the OSI Reference Model [5], which are present in the majority of sensornet applications, and have attracted a large amount of research efforts in the academia.

### 2.1.1 Link Layer Functionality Overview

The datalink layer consists of two sublayers: medium access control (MAC) and logic link control (LLC).

The MAC sublayer directly interacts with the radio. It provides unique device address, performs framing/deframing, and most importantly, regulates the points in time for neighboring nodes to access the shared radio medium. Sensornet MAC protocols often adopt additional power-saving mechanisms in order to preserve battery energy. Such mechanisms include: slotted protocols such as S-MAC [6] and T-MAC [7], in which neighboring nodes are synchronized to a time slot structure whereby random channel contention concentrates in short listening intervals interleaved with long sleeping intervals; preamble listening protocols such as WiseMAC [8] and B-MAC [9], in which the transmitter sends a long preamble to wake up its intended recipient before sending the actual data packet; and IEEE 802.15.4 [10] beaconed mode, in which a *coordinator* node organizes channel access of its associated *device* nodes using a superframe structure that includes contention-based and contention-free periods. All these MAC protocols aim at reducing major components of energy overhead in radio communication: packet collision, packet overhearing, idle listening, and protocol overhead.

The LLC sublayer resides on top of MAC and below the network layer. The main functionality of LLC is link management and protocol multiplexing, and may optionally include error control and flow control. A reference LLC standard is IEEE 802.2 [4]. The protocol requires a common service interface provided by various IEEE 802 MAC protocols via the MAC Service Access Point (MSAP), in turn it provides a uniform service interface to each network layer protocol via a Link Service Access Point (LSAP). The LLC sublayer/MAC sublayer interface defines data transfer primitives for sending and receiving MAC packets in a connectionless datagram style.

Whereas the MAC sublayer is indispensable for its role in radio bandwidth provisioning, wireless protocol architects make different choices about whether an LLC sublayer is used to encapsulate the MAC sublayer. The reason is that different MAC protocols, apart from providing some common single hop packet transfer service, also provide various control functions that can either be used by a dedicated management component at the next higher layer or by a monolithic network layer that handles both data and control operations. As an example of the former approach, WLANs based on IEEE 802.11 [11] use the 802.2 LLC on top of the 802.11 MAC data service, leaving complementary MAC management functions to be exposed to the Station Management Entity (SME) through the MAC Sublayer Management Entity (MLME) SAP. In contrast, the ZigBee protocol [12] takes the latter approach of tight stack architecture, whereby the network layer lies directly atop the IEEE 802.15.4 MAC sublayer, monopolizing both the MAC data service and management services.

### 2.1.2 Network-Layer Protocols for Sensornets

Whenever the diameter of a sensornet exceeds the radio transmission range and nodes outside of their mutual range need to communicate, information must traverse through intermediate nodes, resulting in a multihop network. The various aspects of multihop networking are addressed by different network layer protocols.

Topology control insures that sensor nodes are connected to each other in some way so that there exists one or more communication paths between two distant nodes. Under a specific topology control protocol, a given node may only communicate directly with a subset of its physical neighbors. The resulting reduced connectivity brings about a number of benefits, including lower transmission power, reduced contention, and simplified routing. In exchange, delay and fault tolerance are degraded. A flat topology, such as a mesh, assigns equal roles to all nodes across the network. In contrast, a hierarchical topology, such as a cluster tree, assigns special roles to a subset of nodes who carry out resource allocation and/or packet routing. Topology control in sensornets is often done in a partly or fully distributed manner.

Routing is essentially a path selection problem based on an established topology. The goal of a specific routing protocol is often finding a minimum cost path in some sense. Each node that participate in routing maintains a routing table. A router receives a packet, checks its destination, finds a preferred neighbor by looking up the routing table, and forwards the packet to that neighbor. The forwarding process continues until the packet reaches its destination. A rich variety of routing protocols exist. Internet-style, node-centric unicast routing protocols such as DSDV [13], DSR [14], AODV [15] provide general solutions, but fail to address the application-specific nature of

sensornets, that often imply a certain dominant traffic pattern different from end-to-end unicast, such as convergecast [16], dissemination [17], and diffusion [18]. What's more, some applications require location-centric or data-centric communication rather than node-centric routing.

### 2.1.3 The Sensornet Protocol

The first attempt to define the services that SP should provide and to implement useful interfaces for SP was described in [19] and [20]. The implementation features a message pool that buffers outbound packets for link layer transmission, and a neighbor table shared across layers. It defines a control and feedback mechanism that passes control and status information through the SP layer. To prove SP's wide applicability, the implementation drew in a number of existing link and network protocols in the TinyOS [21] code repository, showing uncompromised overall performance and significantly improved code reuse. This work serves to be the starting point of our own implementation of SP on Contiki.

## 2.2 The Contiki Operating System

An application program written for Contiki typically consists of a number of processes that communicate with each other by passing events. An application may interact with system functions such as the communication stack by sending and receiving events.

### 2.2.1 Contiki Processes

A Contiki process is defined as a C structure that consists of a descriptive text string, a function pointer to a thread function, a *protothread* data structure, and a state variable. The user defines a process structure by specifying the process name and the descriptive text string using the PROCESS(name, string) macro. The user then uses PROCESS_THREAD(name, event, data) to define a thread function for the process, which is essentially a C function that has the prototype "char thread(event, data)", where *event* is an ID for a specific event type and *data* is a pointer used to pass user data. The user may register a defined process to the automatic start processes list so that it is run at system start-up, or call *process_start(&process_name)* to manually start the process from another process. When a process is started, the system initializes its protothread data structure and its state variable. System processes and application processes communicate with each other by posting Contiki events, which consists of a pointer to the destined process, an event ID, and a data pointer, to the system's event queue. The Contiki scheduler dispatches a queued event to the destined process by calling its thread function with the event ID and the data pointer as arguments. When the thread function returns, the Contiki scheduler dispatches the next event from the queue. In additional to asynchronous event passing, processes may post synchronous events to each other, which bypasses the scheduler and is equivalent to nested function calls.

### 2.2.2 Protothread

The user implements process logic in the thread function. A thread function is structured as a protothread, where initialization code is followed by an infinite loop that proceeds execution upon reception of specific events and stops at the next event waiting point. Essentially, such a loop implements a state machine, whose state definitions and state transition rules are however implicitly expressed by the protothread's WAIT_EVENT statements, which saves the current line number in the source file to the protothread data structure and returns. When the next event arrives, the thread branches to the previous WAIT_EVENT statement and continues execution from there. Therefore, a sequential program structure can be retained despite that the process logic consists of multiple states. The following code segment shows a Contiki process that turns the radio on and off at fixed intervals.

```
PROCESS_THREAD(LinkLowPower_process, ev, data)
{
  static struct etimer etimer1;

  PROCESS_BEGIN();

  while(1) {
    etimer_set(&etimer1, CLOCK_SECOND * 2);
    radio_on();
    link_state = LINK_IDLE;
    process_poll(&SPSend_process);

    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&etimer1) &&
     link_state == LINK_IDLE);
    etimer_set(&etimer1, CLOCK_SECOND * 3);
    radio_off();
    link_state = LINK_SLEEPING;
    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&etimer1));
  }

  PROCESS_END();
}
```

### 2.2.3 Timers and Interrupts

The event timer process is a system process started by Contiki. The process is polled by hardware timer interrupt at each system clock tick. The user defines event timers that are associated to specific processes. When an event timer expires, the event timer process posts a specific event to the user process that is associated with the event timer. Similarly, other hardware interrupts, such as an pending packet signalled by the radio chip, are converted to events by low level drivers.

# Chapter 3

# Analysis of SP Services

In the chapter we investigate the main design features of the current TinyOS SP implementation. SP is a bridging layer between the network layer and the MAC sublayer. Multiple network layer protocols may coexist above the SP layer, using the SP data transfer service to exchange packets with their peer protocols in neighboring nodes. The SP neighbor management service allows network layer protocols to participate in the construction of a neighbor table together with available link level mechanisms. At the low level, SP assumes a basic MAC packet transfer service. SP interacts with this packet service and other protocol-specific link-level control services via a customized adaptation sublayer that provides the gluing between generic SP primitives and specific link mechanisms. We will refer to the combined SP adaptation sublayer and the MAC sublayer as the link layer. Figure 3.1 illustrates the SP layer architecture.

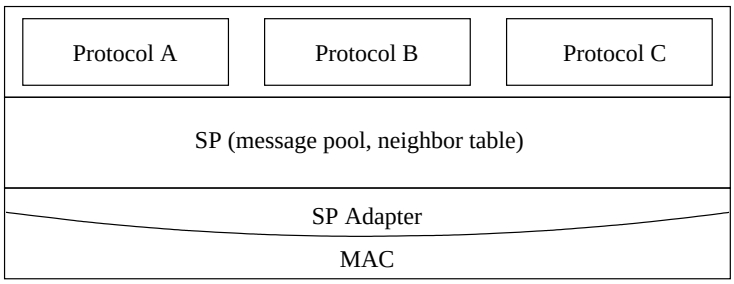


Figure 3.1: The SP layer architecture

## 3.1 Data Transfer Service

We disclose the similarity between SP's data transfer service and IEEE 802.2 in terms of interface and meta-data exchange. We then examine SP's packet buffering and scheduling policy, as well as its support for next higher layer protocol multiplexing.

Table 3.1: Network layer/Datalink layer data transfer interfaces of SP and IEEE 802.2 acknowledged connectionless-mode

| Primitive | SP | 802.2 |
|---|---|---|
| Send packet, expecting status feedback | x | x |
| Receive packet | x | x |
| Modify submitted packet | x | |
| Cancel submitted packet | x | |
| Fetch next packet to send | x | |
| Retrieve packet from neighbor | | x |
| Prepare packet to be retrieved | | x |

### 3.1.1 Sending and Receiving Packets Using SP

The interface SP provides to network protocols for data transfer is very similar to the *acknowledged connectionless-mode* of the IEEE 802.2 network layer/LLC sublayer interface. Particularly, both SP and the 802.2 allow users to specify per-packet priority and reliability, also both provide status feedback to each packet sending request. In addition to single packet sending, SP also supports *message futures*, allowing the user to set the number of packets awaiting to be sent, so that SP may fetch the remaining unsent packets quickly. Table 3.1 shows a comparison between the primitives provided by the two interfaces.

### 3.1.2 Packet Buffering and Scheduling

Each packet submitted to SP for transmission is bound with an *SP message*, a meta-data tag containing control information that is to be used by the link protocol, and feedback information resulted from the subsequent transmission. SP messages are stored in a fixed-size message pool. Based on the availability of the link to each message's destined neighbor, message priority, and submission time, SP selects the next message to transmit. This message scheduling policy is illustrated by Figure 3.2.

If the reliability flag is set in an SP message, SP instructs the link protocol to use any supported reliable transmission mechanism to send the packet, e.g., requiring an acknowledgement from the recipient and retrying an unacknowledged packet up to a specified number of times. The transmission status is stored back into the SP message, so that the network layer user gets notified of whether the packet was transmitted successfully and the whether the underlying channel is congested. The following list summarizes the SP message fields used to pass cross-layer control and feedback information between the network layer and the link layer:

**Control** (submitted together with a bounded packet)

- urgent
- reliability
- retries

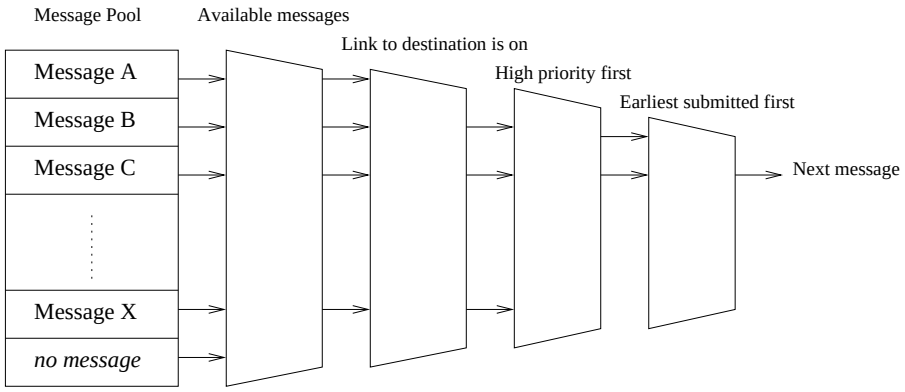


Figure 3.2: SP next message selection policy

**Feedback** (indicates the transmission result of the packet)

- success/failure
- congestion

### 3.1.3 Protocol Multiplexing

When the application requires that multiple network protocols share the datalink service, outbound packets are multiplexed at the link layer. There needs to be a way for the data link protocol to correctly reverse the procedure at the receiving end. SP does not prepend any protocol identifier to submitted packets, therefore it is the network layer's responsibility to handle demultiplexing of received packets. One SP-based network layer design that performs packet demultiplexing using a *dispatcher* component based on packet protocol IDs can be found in [22]. In a special case, however, that there are multiple packet sources above the datalink level but all packets are directed toward a single entity at the receiving end, no demultiplexing is needed. On the other hand, the SP message's *service* field is used by the SP sending process to direct feedback information to the appropriate sending protocol.

In comparison to SP, IEEE 802.2 supports protocol multiplexing by explicit inclusion of two address fields into the 802.2 LLC header: Destination Service Access Point (DSAP) and Source Service Access Point (SSAP). Often an SNAP (Subnetwork Access Protocol) [23] header is also appended to the LLC header in order to identify a specific network layer protocol, such as IP. The 802.2 approach is biased toward a symmetric data link layer architecture that provides fine-grained service multiplexing, but to use these additional headers in every packet would incur significant overhead for many typical sensornet communications.

## 3.2 Neighbor Management Service

SP provides its neighbor management service via a set of operations centered around the neighbor table, which is a knowledge base that stores information about neighboring nodes. Special fields in the table are devoted to maintenance of the link state of each neighbor in order to assist packet scheduling.

### 3.2.1 SP Neighbor Table

SP contains a neighbor table whose interface is open to both the network layer and the link layer. Protocols can use the shared information stored in the table to perform link management, routing, and other functions. Each record in the SP neighbor table contains the following fields:

- neighbor ID
- time-on (local time when neighbor will wake up)
- time-off (local time when neighbor will go to sleep)
- listen (listen to neighbor during its next wakeup period)
- messages pending (packets destined to neighbor awaiting)
- quality (link quality metric)
- extension (user defined parameters)

The SP neighbor interface defines several groups of primitives to operate against the neighbor table:

- construct an entry (add, remove, update)
- query a neighbor by address or entry, query table size
- expire an obsolete neighbor, evict a neighbor from the table
- adjust link quality
- listen to neighbor during its next wakeup period
- populate neighbor table using active link scanning

The primitives listed above comprise data record manipulations and network operations. The former are synchronous, return-on-completion operations, whereas the latter are of asynchronous nature. We will see later in Chapter 5 the programming choices we made in order to simplify the implementation of neighbor management functions.

### 3.2.2 Link State Maintenance

Maintaining the state of each neighbor is necessary to achieve reliability and efficiency in data transfer. Low power MAC protocols using any forms of sleeping/wakeup scheme for sensornets pose an additional challenge for link state maintenance, because the local node needs to keep track of each neighbor's schedule in order to know when the link to a specific neighbor will become available.

SP constantly updates neighbor schedules using the *time-on* and *time-off* fields in a neighbor table entry. The link protocol uses this information to schedule packet sending. A network layer protocol can set a neighbor's *listen* flag so that the local node would wake up and listen to that specific neighbor on its next wakeup period.

Unlike IEEE 802.2, SP does not provide per-link packet sequencing. The user may extend the SP neighbor table with extra columns for storage of packet sequence numbers, or just let the network layer handle redundant packets.

The open interface to the SP neighbor table gives programmers the freedom to construct customized semantics that optimize information sharing among coexisting protocols. However, the freedom to access the neighbor table from anywhere at any time also means it is the user's responsibility to protect against conflicting updates.

## 3.3 Applicability of SP

Based on the previous discussion, we have already revealed some notable features of SP, which are summarized below:

- a unified link service interface
- customizable gluing of MAC and LLC functions
- time scheduling support for low power MACs

Single-hop applications may use SP as a decoupling layer to loosen the often stringent temporal constraints imposed by the underlying MAC protocol. Multi-hop networks of various kinds, on the other hand, benefit from both the temporal decoupling and cross-layer information sharing enabled by SP.

We will explore SP further through our implementation and evaluation in the following chapters.

# Chapter 4

# Methods

## 4.1 Programming Patterns

Implementation of communication protocol primitives, especially those involving inter-layer data transfer such as *request - confirm* primitives shown in Figure 4.1, often involves programming multi-phase operations whose execution flow is split into several states. State transitions are triggered by occurrence of asynchronous events such as arrival of a packet or expiry of a timer. The Contiki scheduler is non-preemptive, so we cannot do block waiting for events. Instead we use *Protothreads*, a program control abstraction in a Contiki process to yield the CPU at points where an event is waited on, and resume execution after the event has arrived.

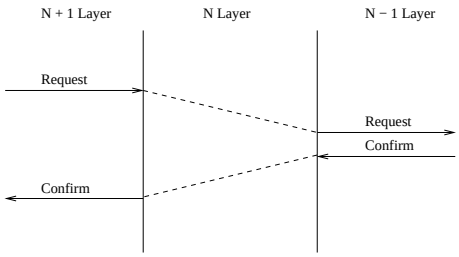


Figure 4.1: Request - Confirm primitive pairs commonly used for inter-layer data transfer

## 4.2 Tools

We use a combination of simulation and experimentation in the course of our protocol development. For function verification, the Contiki NETSIM simulator is used. For timing-sensitive measurements, we use Moteiv Telos nodes to run our protocols.

### 4.2.1 NETSIM

NETSIM is a network simulator for Contiki nodes that simulates a collection of homogeneous nodes deployed in a two-dimensional area. Each node is run as a Linux process that interacts with each other by writing packets to a radio medium emulation process and reading from it. The radio propagation model in NETSIM is a simple disc graph model, in which interference is determined by the presence of packets with overlapping signal range. Due to the nondeterministic nature of the Linux scheduler, NETSIM simulations are not real-time.

We usually use the following work flow for each design iteration:

- configure simulation setup, including deployment and radio parameters
- compile and execute program
- observe network activities and examine log messages
- post-simulation analysis

The NETSIM graphical user interface provides an interactive way for the user to create sensing events, as well as an intuitive view of the overall behaviour of the sensor network under simulation.

### 4.2.2 Telos

A Telos sensor node [24] carries a 2.4 GHz radio transceiver, a 16-bit TI MSP microcontroller, a user button and on-board sensors. Each Telos node comes with a unique 64-bit ID that can be used as MAC address. The CC2420 radio transceiver provides a packet-level data interface for the host controller. Therefore our radio driver deals with whole packets rather than individual bytes as was the case with earlier radio modules. We use an open-source GNU tool-chain for the MSP platform to compile programs and load binaries to the nodes. Because the Contiki build system manages different hardware platforms using a hierarchical directory structure, programs written for NETSIM can be recompiled for Telos without any change, except for code that controls special peripherals.

For tests conducted at minimal scale, i.e., by only two nodes, we may conveniently connect the nodes to a PC via USB ports to observe the log messages. As the the scale goes up, we have to regress to using the on-board LEDs to observe the behaviour of individual nodes.

# Chapter 5

# Implementation

The SP services proposed in [1] have been implemented on the TinyOS operating system [19] [20]. We use the TinyOS implementation as a reference, but do not reuse their code. Rather, we develop our own SP services based on the understanding of the design goals embodied in the original SP proposal. There are two main reasons for this: SP has been proposed as a future standard, so its services are only roughly defined, and the existing implementation should be regarded as tentative and experimental. TinyOS source code is written in the special programming language nesC [25], thus is not directly portable to Contiki, which is based on standard C.

The strength of the TinyOS implementation lies in its proved adaptability with a number of well known MAC protocols including IEEE 802.15.4 and B-MAC. When we started this work, there was no low-power MAC protocol readily available for porting from the Contiki repository. Therefore, we take another approach of starting from a draft vanilla MAC protocol and explore the problems of developing specific link mechanisms and adapting those mechanisms to an abstract set of link services in parallel.

## 5.1 Data Transfer Service

The SP data transfer service provides a uniform way for the next-higher layer user to send unicast packets to any one-hop neighbor or broadcast packets to all neighbors, and to receive packets from them.

### 5.1.1 SP Message Pool

When application data such as sensor readings become available, the communication channel might not be immediately ready to deliver the packet generated, e.g., the destined neighbor node may be in sleep mode at the moment, but will wake up sometime in the future depending on the power-saving MAC mechanism adopted. The SP message pool is designed to provide both necessary buffering of untransmitted packets and to maintain the control and status information associated with each packet.

We implement the message pool in several stages, starting from a simple but incomplete design that evolves into the final full featured structure.

### FIFO

We begin with the simple assumption that the MAC protocol is CSMA/CA based, allowing nodes to access the channel at any time in conformance to a synchronized sleep/wake up scheme. The beaconed mode of IEEE 802.15.4 satisfies this assumption. Since neighboring nodes are synchronized with the local node, transmission of submitted packets can be scheduled in a first-come-first-served manner.

We implement this FIFO-style message pool as a cyclic buffer. Two operations can be performed against the message pool: enqueue and dequeue. Figure 5.1 shows the typical procedures involved in packet sending. The application first generates an outbound packet, then issues a *send request* event to the SP sending process, which in turn enqueues the packet to the FIFO along with packet attributes. When the node becomes idle, the link adaptor sending process fetches a previously enqueued packet from the FIFO and transmits it. Upon completion, the link adaptor feeds back the transmission status to SP, which in turn notifies the application with an *send confirm* event. The application then handles the event and releases the allocated packet buffer. A link state variable keeps track of the current state of the radio driver, protecting the link protocol in sleeping or busy state from being affected by events such as a newly queued message. Figure 5.2 shows the link state transition diagram. FIFOs are very efficient: insertion or removal of an element takes a constant number of cycles regardless of the buffer size.

### Priority Queue

Later we duplicate the FIFO into two queues, for low priority and high priority packets respectively. Packets tagged with the *urgent* flag are enqueued in the high priority queue, so that they get transmitted ahead of ordinary packets. We provide the same enqueue/dequeue interface to the user by means of a pair of wrapper functions that direct the actual packet insertion or removal operation to either of the queues. This introduces a small overhead over the case with a simple FIFO, but the performance is still scalable in respect to buffer size. Also traded off is the effective capacity of the message pool: if either of the queues becomes full, it begins to reject packets of that priority, despite that the other queue might still have empty slots. Figure 5.3 illustrates the double queue structure.

### Link State-Aware Message Pool

As long as the previous assumption of synchronized, randomly accessible neighborhood holds, queues suffice to provide fair and efficient scheduling. TDMA-based MAC protocols that distribute each neighboring node's sleep/wake up schedule across the channel period, however, require the local node to be able to schedule packet transmissions according to the knowledge of its neighbors' schedules. In such a case, in order to filter out the not schedulable packets, the SP message pool needs to be aware

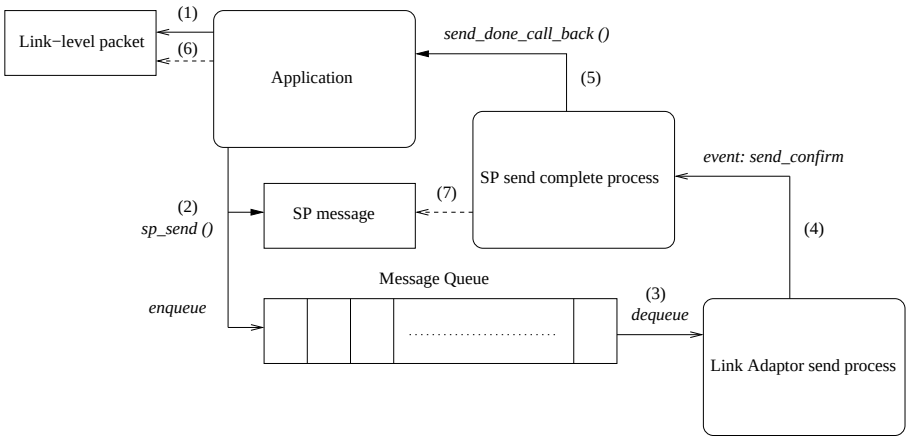


Figure 5.1: Sending a packet via SP using FIFO message pool. (1) Application generates a packet. (2) Application calls *sp_send()* to bind packet and attributes with an SP message, then enqueues the message. (3) Link adaptor wakes up and dequeues the message. (4) The link adaptor notifies the SP send complete process about completion of packet transmission by signaling a send confirm event along with the SP message. (5) The SP send complete process invokes the application's callback routine that handles send completion. (6) The application frees the sent packet. (7) SP frees the SP message.

of the link state of the neighbor that each submitted packet is destined, so that those schedulable are scheduled based on priority and submission time. Such a next packet selection policy has been illustrated in Figure 3.2, which reflects significant added complexity compared with FIFO queues.

We implement the link state-aware message pool as a static array of SP messages. Two operations, post and pend, can be performed against it and are shown in Figure 5.4. The post operation inserts to the array an SP message tagged with submission time. The pend operation carries out link state querying as well as priority and submission time comparison. Link states are stored in the SP neighbor table's time-one/time-off columns. Since the neighbor table is implemented as an unsorted array, the time it takes to determine a neighbor's link state is O(N), where N is the array size. If the message pool has a size M, the time it takes to find the highest priority among the schedulable messages is therefore O(M*N). A second iteration through the message pool is needed to choose the earliest submitted packet from the result of the first iteration, thus the algorithm's complexity is O(2*M*N).

The inter-process communication pattern among the application, SP, and the link adaptor also needs to be changed to properly handle link state-aware packet scheduling. Instead of simply letting the application posting messages to SP and the link adaptor pending messages from SP as in the case of FIFO queues, now the application still does the posting, whereas SP takes the initiative to invoke pending for the next message. Figure 5.5 illustrates the event-triggered operations involved in packet sending. This

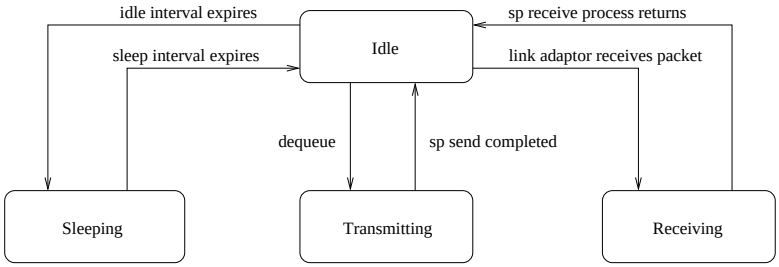


Figure 5.2: State transition diagram of a node's link state

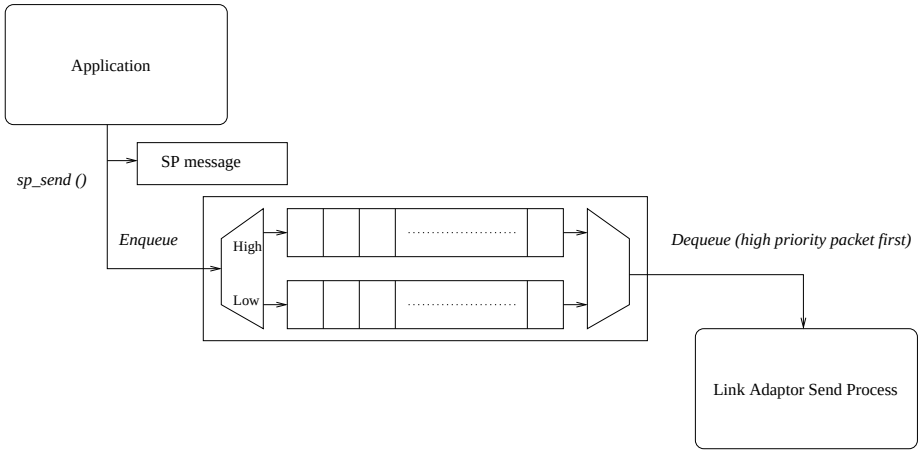


Figure 5.3: The priority queue has the same enqueue/dequeue interface as FIFO.

implementation of the SP message pool resembles our TinyOS counterpart, except that the TinyOS message pool lets SP also fetch packets from the application by taking advantage of user supplied information called *message futures*, a feature we have not found to be of great necessity.

### Link State-Driven Message Pool

The link state-aware message pool is a full feature design that supports various power saving MACs. We use this design in most of the evaluations. The coupling of neighbor state maintenance to data transfer, however, has prompted us to contemplate a message pool data structure whose posting and pending operations are directly driven by changes of link states in the neighbor table. In such a data structure, a message is posted to the entry for the destined node in the neighbor table, and is to be transmitted when the link to the specific neighbor becomes idle. The implementation and evaluation is left to future work. Figure 5.6 illustrates this conceptual link state-tied message pool.

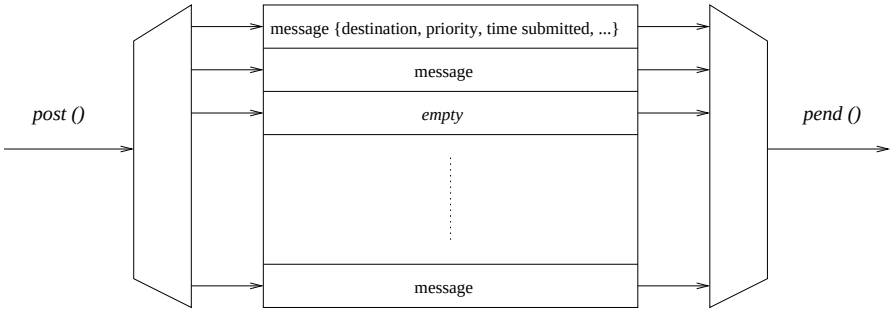


Figure 5.4: Link state-aware message pool

### 5.1.2 Control and Feedback

We implement a vanilla link protocol to work together with SP. The protocol is capable of automatic retransmissions based on link acknowledgement timeout. We define a link-level packet header format that contains a data type field, a packet sequence number, destination address and source address, and payload size. The application may order a packet to be transmitted reliably or not, by specifying in the packet's bounded SP message the *reliability* flag and the *retries* value, which are in turn interpreted by the link adaptor to construct the packet header's data type field, and to invoke the automatic retransmissions mechanism. The reliability flag is reused to notify the transmission result (success/failure) of a packet, so that the application may act accordingly. Our vanilla link protocol does not make use of the *urgent* control flag or the *congestion* status flag.

## 5.2 Neighbor Management Service

The SP neighbor table is another core data structure besides the message pool. It stores shared information about the local node's neighbors on behalf of various different components in the protocol stack.

We implement the table as a fixed size array of records, each of them containing information about a neighbor node. A record contains mandatory fields including neighbor ID, time-on and time-off, listen flag, number of messages pending, link quality, as well as user-defined extension fields.

### 5.2.1 Neighbor Table Operations

We implement the operations against the neighbor table as global C functions. They are listed as follows:

- insert a record
- set a neighbor's properties
- query neighbor properties

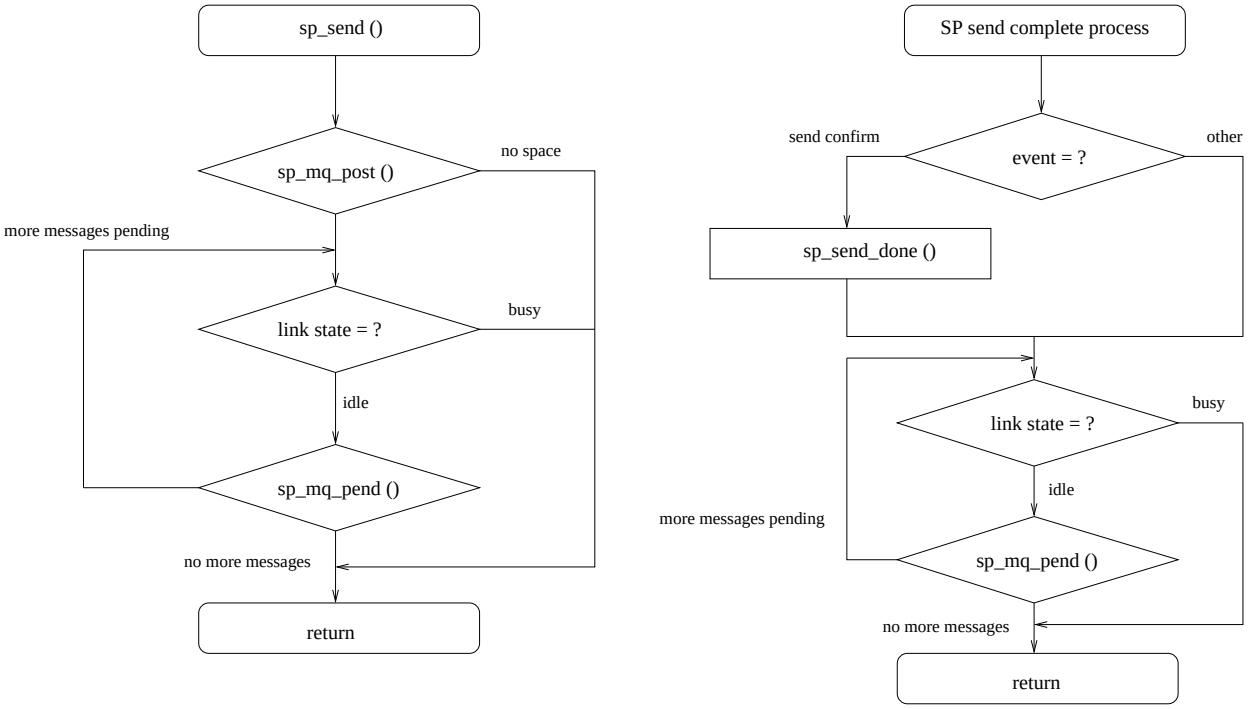


Figure 5.5: Flow charts showing the *sp_send()* routine and SP send complete process working with the link state-aware message pool

- query whether a neighbor is idle
- query table size
- evict a record
- find an empty record
- clear a stored record
- clear the whole table

### 5.2.2 Neighbor Management Policy

Our previous discussion about packet scheduling has shown the usefulness of neighbor management for link state maintenance. Neighbor management is useful in at least two other important aspects: 1. topology control; 2. routing and forwarding. We will show in Chapter 6 how a network protocol ported to SP makes use of the neighbor management utilities.

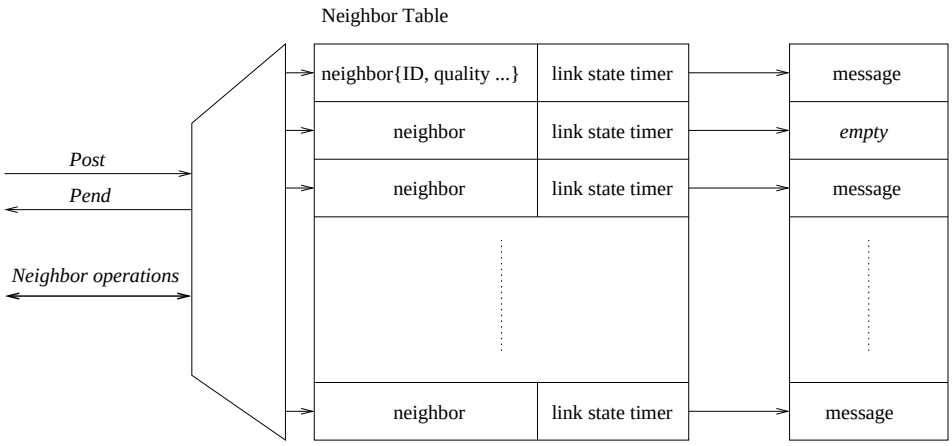


Figure 5.6: Link state-driven message pool. The message slots are matched to neighbor records in the neighbor table one by one.

During the lifetime of a sensor node, neighbor information can be collected in two ways: active probing or passive monitoring. Active probing usually implies broadcasting a scanning packet to the neighborhood, or sending a probing packet to a known neighbor, and expecting responses from the probed neighbors. Passive monitoring, or packet sniffing, instead relies on processing of received and overheard packets captured in data traffic to construct a map of one's neighborhood. Woo et al. [16] have investigated the major design considerations related to passive monitoring, including neighbor insertion and eviction policies.

Our SP implementation does not enforce a neighbor management policy in terms of specific criteria for insertion, removal and updating of neighbor information. Rather, we only provide the necessary interface to operate the neighbor table and leave the choice of neighbor management policy to the user.

# Chapter 6

# Evaluation

In this chapter we present the results from a number of single hop communication tests, and then discuss the findings made by porting of a multihop protocol to SP.

## 6.1 Single Hop Evaluation

We develop an artificial traffic generator program to test the performance of SP. The program may generate a single-hop packet stream based on user-specified burstiness and transmission rates.

We conduct a one-to-one throughput test between two Telos nodes whose radios are configured to communicate using a certain duty cycle. The SP message pool size is set to one. Retransmission is turned off. The sending node generates a new packet for SP to send when a previous packet has been acknowledged by the receiving node. The result shown in Figure 6.1 verifies that SP's performance is consistent as the radio duty-cycle changes.

We configure the traffic generator to generate bursts of packets at 200 ms intervals. We want to put SP under full load by generating traffic at an average packet rate that slightly exceeds the maximum channel bandwidth so that excessive packets are rejected by SP, and then examine how throughput varies with different settings of the message pool size. If the message pool is set to contain only one message at most, the channel becomes fully loaded when the burst size reaches 14 packets, as shown in Figure 6.2. We then set a constant burst size to be 14, but vary the size of the message pool to see if a larger buffer improves throughput. Surprisingly, results shown in Figure 6.3 indicates that the throughput begins to degrade when the buffer size is larger than two. We attribute this performance loss to the inefficient posting and pending operations against the link-state aware message pool.

## 6.2 Multihop Networking Using SP

One salient feature of SP is its support for coexisting network layer protocols. The TinyOS implementation has shown SP working with three such protocols: MintRoute

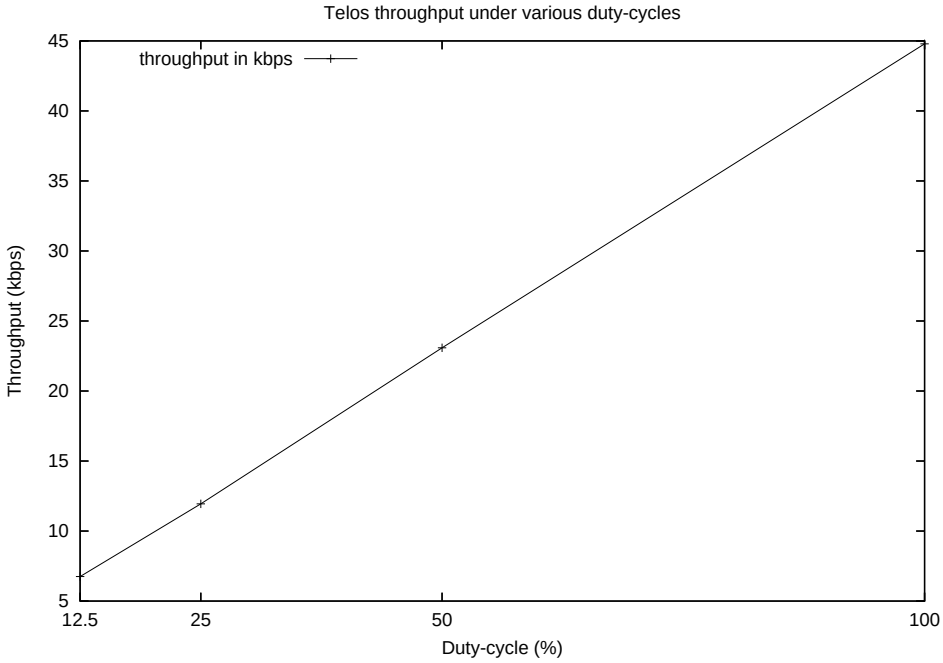


Figure 6.1: Telos throughput under different duty cycles. The transmitting node feeds 100 packets, each 45 bytes in size, to a receiving node as quickly as possible. The radio transceiver's carrier sensing capability tends to extend the wake-up interval by a small amount at each switching point, which introduces a small residual throughput gain observed at lower duty-cycles.

[16], Trickle [17] and Synopsis Diffusion [18], which have been ported from the TinyOS code repository.

We choose to port instead the *Treeroute* protocol from the Contiki repository to SP. Treeroute allows a large number of sensor nodes to cooperatively form a tree topology rooted at a designated basestation and report collected sensor readings to the basestation. Treeroute consists of two protocols: a route construction protocol adapted from the Trickle protocol [17], and a data forwarding protocol based on the routing tree established.

### 6.2.1 The Treeroute Protocol

The original Treeroute protocol has been developed to use the uIP [26] stack in Contiki as the underlying communication service. Route packets and data packets are delivered in two separate UDP broadcast connections.

The route construction protocol establishes a tree topology by having each node to overhear route updates broadcast by neighbors to learn its shortest distance to the root node in terms of number of hops, and broadcasts its own hop count to neighbors. Upon

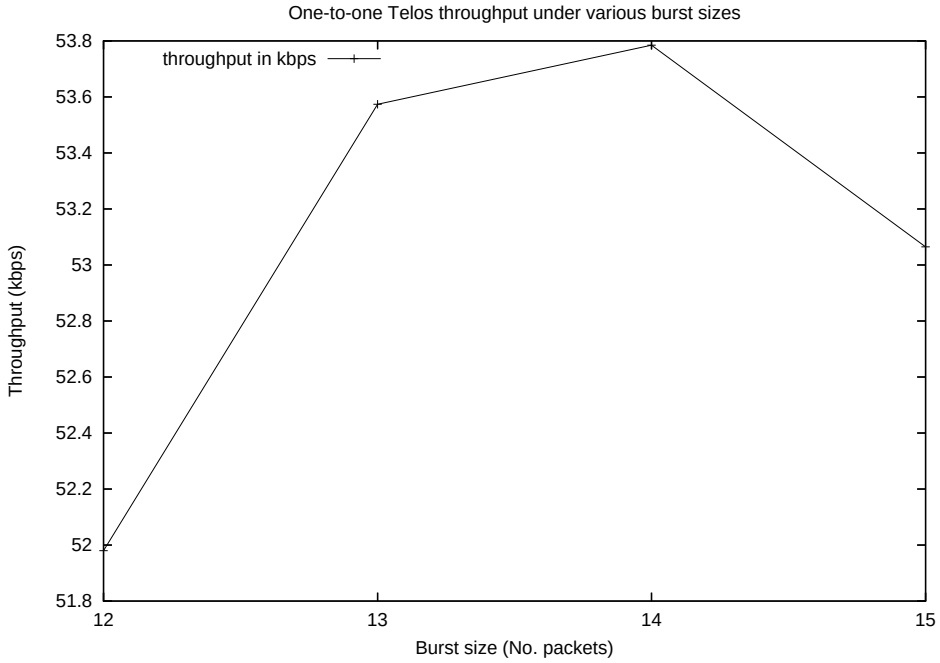


Figure 6.2: Telos burst throughput versus burst size. Message pool size is set to be 1.

receiving a route update, a node compares its hop count with the sending neighbor's hop count: a difference larger than one would imply inconsistency and prompt the node to adjust its hop count. Initially, nodes have maximum hop counts. As the root node starts broadcasting, one-hop neighbors detect inconsistency, update their hop counts to one, and then propagate the information further down to lower level nodes, forming a tree across all nodes in the network. After the tree topology has stabilized, nodes continue broadcasting at diminishing rates. Any future changes in the topology will be detected by neighbors of the spots of the changes and then propagated across the whole network.

A node starts its data forwarding protocol as soon as it has obtained a valid hop count, i.e., a route to the root. Each node periodically carries out sensor readings and reports the data to the root, by sending a unicast data packet to its best parent node, i.e., the neighbor that is one hop closer to the root and has the best link quality, which in turn relays the packet one level up in the tree, and so on, until the packet reaches the root. In order to save bandwidth, the relaying node appends its own sensor reading to the packet to be forwarded, thus only as many packets as the number of branches are originated within each sensor reading interval. To alleviate packet loss, the data forwarding protocol performs per-hop retransmissions based on *implicit acknowledgements*, i.e., overheard packets forwarded by its parent to its grandparent.

Figure 6.4 illustrates the Treeroute route construction and data forwarding processes.

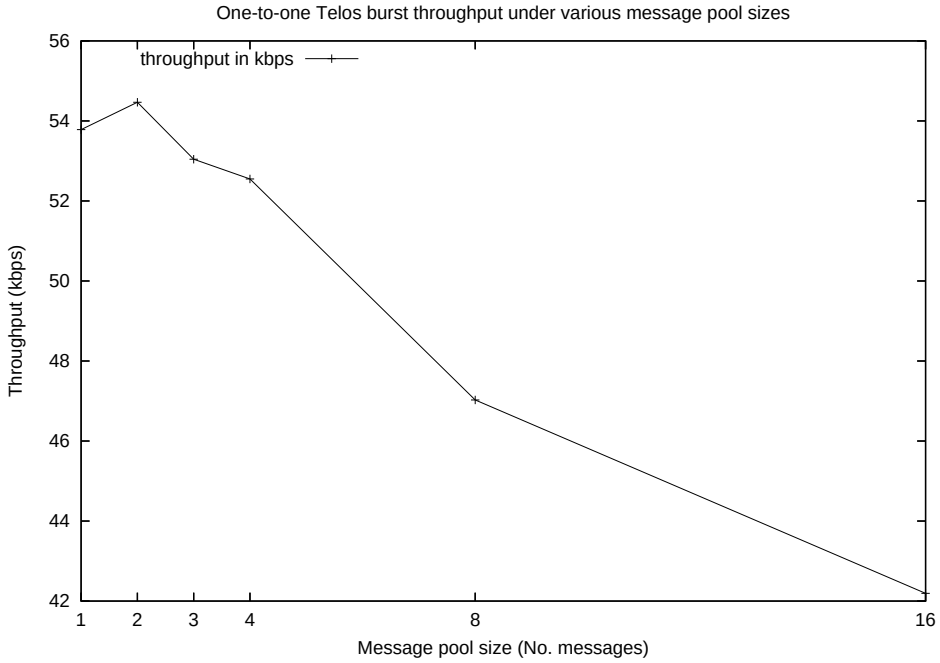


Figure 6.3: Telos burst throughput versus message pool size. Burst size is set to be 14 packets every 200 ms.

### 6.2.2 Porting Treeroute to A Modular Network Layer Architecture

The major considerations for the porting of the original Treeroute include:

- partitioning functionality and defining component interfaces including the interface with SP.
- defining packet header formats
- making use of SP neighbor management

As we consider the porting of Treeroute to SP, we first attempt to adopt the modular Network Layer Architecture (NLA) proposed by Cheng et al. [22]. Similar to IP, NLA provides a best-effort, connectionless multihop communication abstraction to applications. NLA proposes a general, component-based framework to host network layer protocols. Each protocol is functionally partitioned into components interacting with each other via standard interfaces. Coexisting protocols are multiplexed together by an output queue and a dispatcher, which use SP as a single-hop communication service.

We redraw the NLA architecture in Figure 6.5, which resembles a classic IP router architecture consisting of a control plane and a data plane. To enable finer code reuse and run-time sharing, NLA further partitions the control plane and the data plane into smaller, standardized components that implement particular policies or mechanisms.

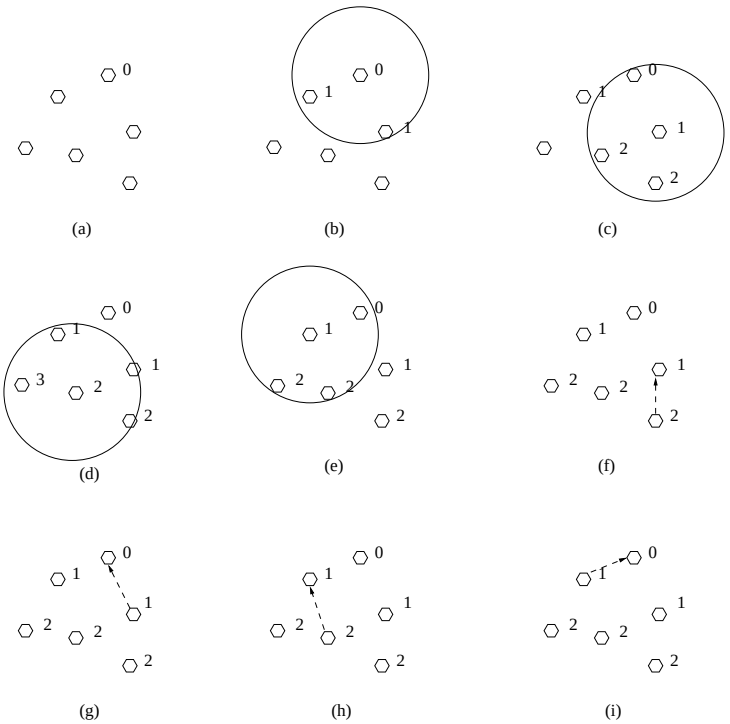


Figure 6.4: Treeroute route construction and data forwarding. A node's hop count is denoted by the number beside it. (a) to (e) show the route construction process. (f) to (i) show the data forwarding process.

The control plane is partitioned into *routing topology* (RT) and *routing engine* (RE), whereby RT is responsible for discovering and maintaining the network topology and RE computes and maintains routes over the topology. The data plane consists of *forwarding engine* (FE) that obtains next hop(s) and *output queue* (OQ) that buffers packets across different protocols at the network layer. Multiple RTs, REs and FEs may coexist in the architecture, allowing flexible combination of protocols. A *protocol service* component provides a wrapping service interface to the application layer on behalf of a specific network layer protocol. Furthermore, the *dispatcher* demultiplexes packets to the FEs for different protocols, whereas the *network service manager* enables the application to intercept packets flowing through FEs.

In addition to defining functional components, NLA defines a generic packet header format. The network layer packet header of NLA consists of four sub-fields. Each subfield is used by a component involved in packet forwarding and is opaque to other components. Figure 6.6 shows the packet header format.

In order to test if the architecture is suitable to use together with our SP implementation, we partition the Treeroute route construction and data forwarding services into NLA components, define an NPDU header compliant with the NLA header format, and

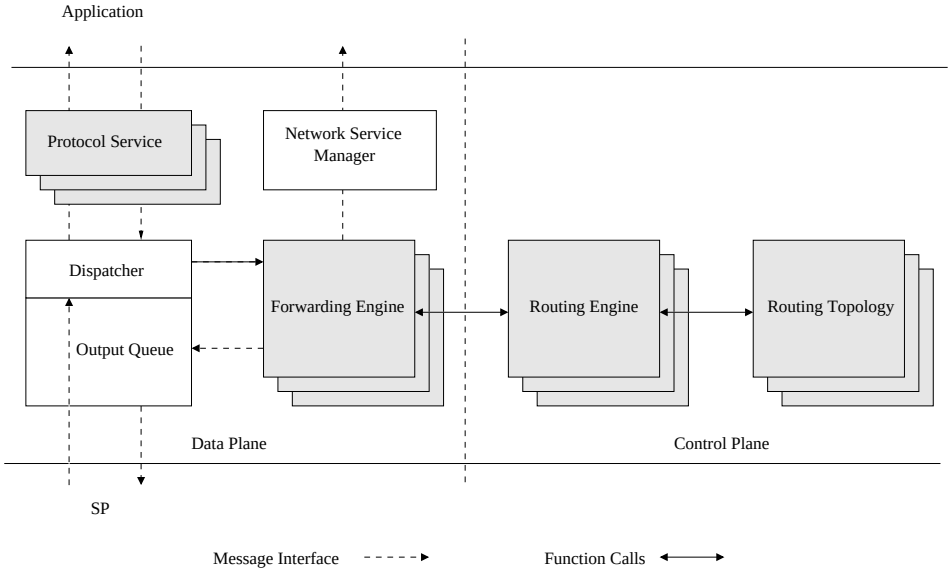


Figure 6.5: NLA architecture overview.

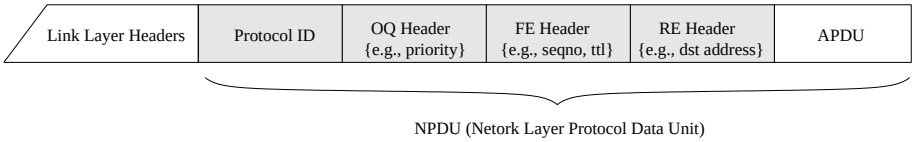


Figure 6.6: NLA packet header format. Each sub-header contains packet attributes to be inspected by a specific component in the forwarding path.

use the SP neighbor table to store topology and routing information.

Figure 6.7 illustrates our NLA-style implementation SP-based Treeroute. We partition each of the two network layer services into: a service process that performs NPDU header construction and destruction for outgoing and incoming packets respectively; a forward process that queries a routing function to obtain the next hop and determines whether a submitted packet should be forwarded or passed to the service process. The forward processes are also responsible for constructing the link-level header and specifying the SP attributes before submitting a packet to the SP message pool. Both outgoing and incoming NPDUs are passed to the dispatcher process, which inspects their protocol IDs and submits them to the appropriate forwarding processes.

Our NPDU header consists of sub-header fields as shown in Figure 6.8. Since we use the SP message pool in place of the NLA *output queue*, the OQ sub-header is used to specify of SP message attributes. The FE sub-header includes a sequence number to suppress redundant packets and a hop limit counter to avoid potential routing loops. The RE sub-header includes the network addresses of the destination node and the source node.

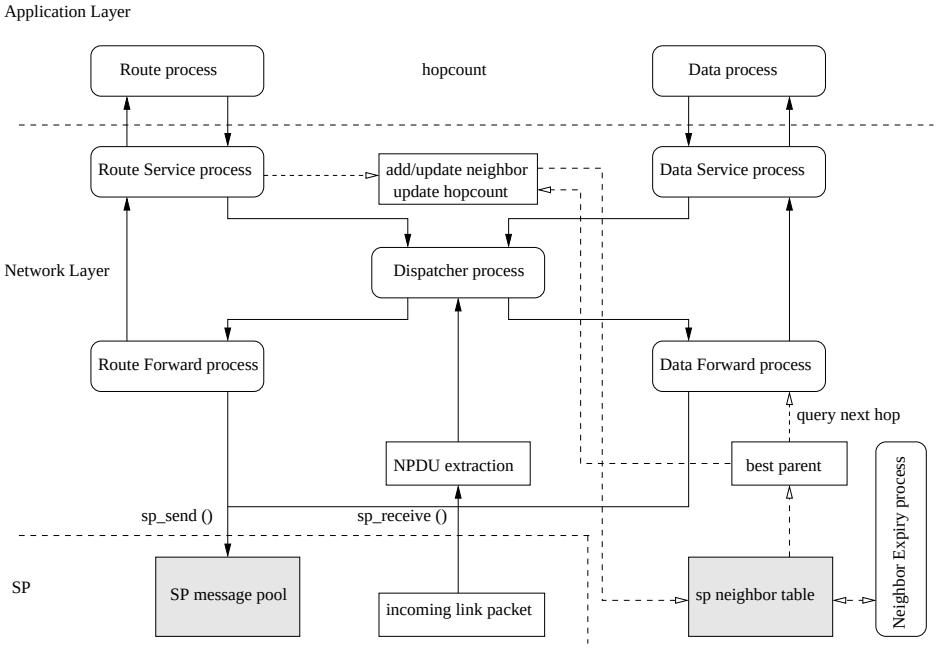


Figure 6.7: Treeroute protocol reimplemented using NLA and SP.

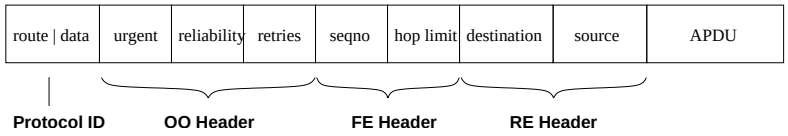


Figure 6.8: NPDU packet format for NLA-adapted Treeroute.

We extend the SP neighbor table to use it as also a routing table for Treeroute. Two columns are added: hop count and expiry timeout. When the route process receives a route update, it either inserts the source node as a new neighbor to the neighbor table, or updates the neighbor's record with the newly received hop count and link quality. The process then adjusts the local hop count to be the best parent's hop count plus one. The operation involves querying the neighbor table's 'hop count' column. When handling originating or forwarding data packets, the data forward process queries the neighbor table to find the best parent and use it as the next hop. To cope with topology changes caused by node mobility, the neighbor expiry process evicts obsolete records from the neighbor table based on expiry timeouts.

Because SP provides per-hop reliability, we are able to remove the retransmission mechanism used for packet forwarding in the original Treeroute, which simplifies the code of the data forward process.

### 6.2.3 Verification of SP-based Treeroute

We verify SP-based Treeroute on NETSIM. We deploy a 4 x 4 grid network where each node can reach its immediate neighbors in the same row or column. After we set a corner node to become the root, a tree begins to take shape, growing towards the opposite corner. When the Trickle interval is set to be 1 second, the time it takes to form a 6-hop tree across the grid is 30 seconds in the worst case. Experiments conducted under different deployment and signal range settings show consistent performance of the protocol.

We notice however a tension between the potentially large number of detectable neighbors and the limited neighbor table size, which further raises the question of how to balance link diversity and link utility in neighbor management policies. In the case of Treeroute, links to parents are of much higher utility than links to children and siblings, because only parents can serve as next hop destination. Therefore we modify the neighbor insertion policy so that only neighbors with lower hop counts are admitted, greatly reducing the size of the neighbor table.

A second issue arises as overhearing of network layer acknowledgements is replaced by automatic link layer retransmission. In Treeroute, because our link adaptor consumes acknowledgements from the parent, the network layer needs to hook a neighbor refreshment routine to SP's send done callback function, otherwise the neighbor expiry process would expire the parent inadvertently if no route updates are heard from the parent within the neighbor expiry timeout interval, which is very likely given the low broadcast rate maintained by the route process's Trickle mechanism. A more subtle side effect caused by reduced overhearing is the ignorance of second best parents who reside on other branches. Lack of activity heard from these next hop candidates would expire them prematurely. We can imagine that in reality, if a node's current parent becomes unavailable temporarily, then the node has no choice to switch to another branch, but can only resets its hop count and try to rejoin the tree.

We deploy a 7-node Treeroute application in a corridor. A PC connecting to the root node monitors the operation of the protocol. The overall stability and performance is consistent with the simulation, in spite of a few problems not previously discovered by simulation. First, because the routing algorithm prefers shortest paths, sometimes a weak link in the trunk or a main branch causes significant service degradation. We modify the best parent algorithm to filter out candidates with low hop counts but weak signal strengths, and add a filtering function to smooth out link quality fluctuations. This results in considerably improved path robustness, albeit at the cost of reduced routing diversity. The problem exposes Treeroute's lack of end-to-end route metric, a shortcoming not addressed by SP either. Secondly, if a branch of nodes has only one parent, then if the crucial link to the parent is broken, the whole branch should return to initial states immediately. In reality, however, the head of the orphaned branch would update its hop count so that it becomes a child of one of its original children, initiating a sequence of Ping Pong route updates between the nodes in the dead branch until the crucial link reappears or the full branch dies out when the maximum hop count is reached.

### 6.2.4 Issues With the NLA Architecture

Our porting of Treeroute to NLA/SP has not been without challenges. We summarize our findings about the problems with the NLA/SP architecture below:

- Single hop protocols such as Trickle have little use of the fine grained components in NLA, except the dispatcher for the purpose of coexistence with other protocols. SP is sufficient for single hop communications. Further more, strict compliance to the NLA header format might introduce unnecessary overhead. For example, we have to duplicate the source link layer address in the RE sub-header to allow neighbor insertion at the recipient node's network layer.

- The NLA output queue has similar function as the SP message pool, thus can be replaced by the latter, unless advanced packet scheduling policies are needed. The OQ sub-header contains basically the same information as SP messages. The header is needed if the information must traverse through a multihop path together with the payload. In that case, however, it would mean SP messages contain redundant information presented in the bound link packet, which certainly does not serve as a good argument to use SP.

- The NLA dispatcher's symmetrical structure might potentially remove useful distinction between inbound and outbound packets. For example, an inbound packet's link layer header are stripped off before entering the dispatcher. The header might contain information such as source link layer address or received signal strength (RSS) that the application wants to intercept.

- NLA assigns the duty of I/O packets diversion to the forwarding engine. This works fine in pure-form forwarding, where the payload is never modified by intermediate routers. In Treeroute's data forward process, however, we need to append a locally produced payload to the payload of the packet being forwarded, breaking the application-agnostic feature intended by the NLA forwarding engine.

### 6.2.5 An Alternative Network Layer Architecture

Having learned precious experience from the NLA/SP Treeroute port, we design an alternative network layer architecture that shares the same external interfaces as NLA but has a simpler internal structure.

Figure 6.9 illustrates the architecture. In this architecture, the dispatcher's responsibility has been changed to passing inbound link packets to their intended service processes, which in turn parses the link layer header to e.g. invoke neighbor update operations, as well as the network layer header to determine whether the packet should be passed up to the application or forwarded. The forward processes now accept outbound NPDUs directly from their respective service processes.

The modified network layer implements the same Treeroute services on SP. The major gains over the previous NLA implementation is reduced code size and easier information sharing between components. The major drawback, on the other hand, is loosened layer boundaries.

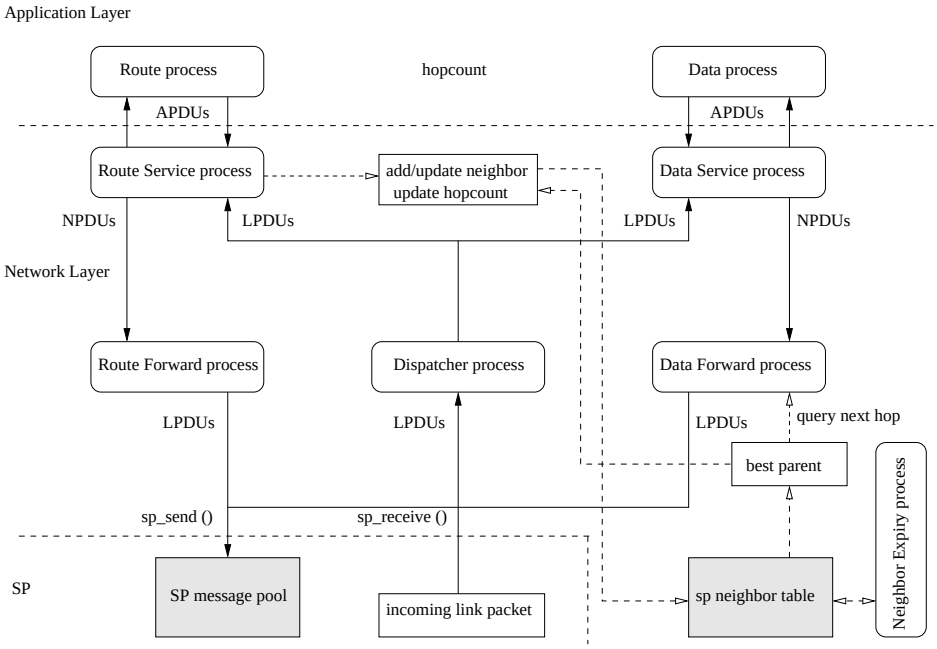


Figure 6.9: Treeroute protocol reimplemented using a simpler network layer architecture and SP.

# Chapter 7

# Conclusion and Future Work

## 7.1 Conclusion

Our implementation and evaluation of the Sensornet Protocol on the Contiki operating system show that SP is a suitable link abstraction service for construction of modular protocol stacks that are growing more and more complex in sensor networks. In addition, we have verified that SP is capable of supporting multiple coexisting network layer protocols when working together with a network layer architecture such as NLA. We have also found a number of interesting trade-offs between functionality and comlexity in the implementation.

We believe there is still much space for improvement, however, in aspects including message pool throughput, support of different neighbor management policies, and coordination with network layer components.

## 7.2 Future Work

We plan to further improve our SP implementation for Contiki in the near future, in the following areas:

- develop link adaptors for representative MAC protocols to explore any unaddressed needs for the SP link abstraction.
- quantify the resource requirements of SP in terms of code size and memory, and compare with IEEE 802.2.
- integrate SP with the-state-of-art features in Contiki, notably the Rime stack and the COOJA cross-layer network simulator [27].
- streamline the searching and sorting operations of the SP message pool and neighbor table to improve both memory and code efficiency.

# Bibliography


[1] D. Culler, P. Dutta, Cheng T., R. Fonseca, J. Hui, P. Levis, and J. Zhao. Towards a sensor network architecture: Lowering the waistline. In *Proceedings of the International Workshop on Hot Topics in Operating Systems (HotOS)*, 2005.

[2] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *IEEE Emnets-I*, 2004.

[3] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the 4th ACM Conference on Embedded Networked Sensor Systems (SenSys 2006)*, Boulder, Colorado, USA, 2006.

[4] IEEE Standard for Information technology – Telecommunication and information exchange between systems – Local and metropolitan area networks – Specific requirements. Part 2: Logical Link Control. IEEE Computer Society, 1998.

[5] H. Zimmermann. OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications*, 28(4):425–432, 1980.

[6] W. Ye, J. Heidemann, and D. Estrin. An Energy-Efficient MAC Protocol for Wireless Sensor Networks. In *Proceedings of the 21st International Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2002)*, New York, NY, USA, June 2002.

[7] T. van Dam and K. Langendoen. An adaptive energy-efficient MAC protocol for wireless sensor networks. In *Proceedings of the first international conference on Embedded networked sensor systems*, pages 171–180. ACM Press, 2003.

[8] A. El-Hoiydi, J.-D. Decotignie, C. C. Enz, and E. Le Roux. wisemac, an ultra low power mac protocol for the wisenet wireless sensor network. In *SenSys*, pages 302–303, 2003.

[9] J. Polastre, J. Hill, and D. Culler. Versatile low power media access for wireless sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 95–107, New York, NY, USA, 2004. ACM Press.

[10] IEEE Standard for Information technology – Telecommunication and information exchange between systems – Local and metropolitan area networks – Specific requirements. Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs). IEEE Computer Society, October 2003.

[11] IEEE Standard for Information technology – Telecommunication and information exchange between systems – Local and metropolitan area networks – Specific requirements. Part 11: Wireless LAN Medium Access Control (MAC) and Phsical Layer (PHY) Specifications. IEEE Computer Society, 1999.

[12] ZigBee Specification 2006. ZigBee Alliance, December 2006.

[13] C. Perkins and P. Bhagwat. Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. In *ACM SIGCOMM'94 Conference on Communications Architectures, Protocols and Applications*, pages 234–244, 1994.

[14] J. Broch, D. B. Johnsona, and D. A. Maltz. The Dynamic Source Routing Protocol for Mobile Ad Hoc Networks (DSR). Internet-draft, IETF MANET Working Group, March 1998.

[15] C. Perkins, E. Belding-Royer, and S. Das. Ad-hoc on-demand distance vector routing. IETF Internet RFC 3561, October 2003.

[16] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 14–27, New York, NY, USA, 2003. ACM Press.

[17] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proceedings of NSDI'04*, March 2004.

[18] S. Nath, P. B. Gibbons, S. Seshan, and Z. R. Anderson. Synopsis diffusion for robust aggregation in sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 250–262, New York, NY, USA, 2004. ACM Press.

[19] J. Polastre, J. Hui, P. Levis, J. Zhao, D. Culler, S. Shenker, and I. Stoica. A unifying link abstraction for wireless sensor networks. In *SenSys*, 2005.

[20] J. Polastre. *A unifying link abstraction for wireless sensor networks*. PhD dissertation, Univerisity of California, Berkerly, 2005.

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

[22] Cheng T., R. Fonseca, S. Kim, D. Moon, A. Tavakoli, D. Culler, S. Shenker, and I. Stoica. A modular network layer for sensornets. In *Proceedings of OSDI*, August 2006.

[23] IEEE Standard for Local and metropolitan area networks : Overview and Architecture. IEEE Computer Society, March 2002.

[24] J. Polastre, R. Szewczyk, and D. Culler. Telos: Enabling ultra-low power wireless research. In *Proc. IPSN/SPOTS'05*, Los Angeles, CA, USA, April 2005.

[25] 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.

[26] A. 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.

[27] F. Österlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt. Cross-level sensor network simulation with COOJA. In *Proceedings of Proceedings of the First IEEE International Workshop on Practical Issues in Building Sensor Network Applications (SenseApp 2006)*, page 8, Tampa, Florida, USA, 2006.