PERFORMANCE OPTIMIZATION FOR TCP-BASED WIRELESS SENSOR NETWORKS

Masterarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Ulrich Bürgi
2011

Leiter der Arbeit:
Professor Dr. Torsten Braun
Institut für Informatik und angewandte Mathematik

Contents

List of Figures

List of Tables

Listings

Acknowledgment

I would like to thank my supervising tutor Philipp Hurni for the many hours spent providing great support and invaluable assistance, Prof. Dr. Torsten Braun for giving me the opportunity to write my Master's thesis at the RVS group, and all the other members of the research group who supported me with helpful advice.

Chapter 1

Summary

Nowadays, Wireless Sensor Networks become more and more popular in various domains, such as environmental monitoring, health care, and military operations. They provide a cost-efficient and easy deployable way for data collection and data aggregation in a widespread area of interest. With the availability of lightweight implementations of a TCP/IP stack small enough to run on wireless sensor nodes, it is possible to directly contact the nodes with common, TCP-based applications. Furthermore, wireless sensor networks can be part of the Internet, which, for instance, makes it easier to maintain nodes located far away from the operator.

Using TCP/IP in wireless sensor networks, however, has some considerable disadvantages. TCP has been designed for wired, low error networks, for which packet loss is mainly caused by network congestion. In wireless multihop networks with high bit-error rates TCP's congestion control mechanisms lead to significant degradation of throughput. Additionally, due to the required end-to-end retransmissions, energy consumption is increased, which leads to a shorter lifetime of the sensor nodes.

For this master's thesis, we introduce and evaluate a modular approach of caching TCP segments on intermediate wireless sensor nodes to perform local retransmissions, which we implemented into the Contiki operating system's µIP stack, the lightweight TCP stack implementation of one of today's most popular operating systems for small embedded devices. Besides the core functionality of the simple retransmission mechanism, we also implemented various extensions to test alternative retransmission schemes, such as multiple retransmissions, hop-distance dependent retransmission timeouts, and activity dependent retransmissions. The proposed design works independently of the MAC layer and does not require any specific features from the MAC protocol used. Furthermore, it is transparent to the application layer and a modification of the TCP protocol is not required. The goal of the caching and retransmission system is to reduce the need for end-to-end retransmissions in case of packet loss, increase throughput of TCP data and, in combination with energy efficient MAC protocols, reduce energy consumption.

To evaluate our approach, we perform real-world experiments in an indoor wireless sensor node testbed. We test the behavior of the retransmission mechanism with various combinations of the proposed extensions by transporting TCP data over five routes of different length, and in two scenarios, which differ in network topology and amount of traffic. We also investigate how the MAC layer influences the effectiveness of our implementations by running the experiments with four different MAC protocols, among them radio duty-cycled and energy unconstrained

protocols. In total, we evaluate 2500 individual experiment configurations.

The experiment results reveal, that in most situations our proposed caching and retransmission mechanisms are able to increase throughput, in certain scenarios by up to 84%. Also the power consumption of wireless sensor nodes running energy-efficient MAC protocols can be decreased, despite the increased TCP throughput.

Chapter 2

Introduction

2.1 Wireless Sensor Networks

A Wireless Sensor Network (WSN) is a collection of autonomous, low-cost, low-power sensor nodes with the purpose to monitor ambient conditions, such as temperature, humidity, luminance, movement, pressure, noise levels, and others, within a given area [1]. Along with the actual sensors, sensor nodes are equipped with a radio for node-to-node communication used to transport sensed information through the WSN to an observation entity. They also feature on-board processing units and memory, with which gathered data can be preprocessed before sending, in order to limit the amount of data to transmit. An example of a wireless sensor node is the TelosB / Tmote Sky platform, which we describe in more detail in Section 3.4.1.

Many application areas for wireless sensor networks exist today. For example, WSNs are deployed for military operations, environmental monitoring and health care. Obviously, each WSN scenario has individual requirements to the sensor nodes' hardware and software. A system which observes seismologic activity of an active volcano [2], for instance, requires a network capable of processing a high amount of continuous measurements and nodes with long battery lifetime. On the other hand, a WSN with the purpose of detecting gunfire and localizing the shooter [3] has to be able to transport irregularly occurring event data with as little delay as possible.

A typical configuration of a Wireless Sensor Network is illustrated in Fig. 2.1. A large number of sensor nodes are distributed in an area of interest and interconnected to build a wireless multi-hop mesh network. The WSN is connected to either a base-station or through a gateway to another network (e.g. the Internet) to which the sensed data is forwarded and from which the sensor nodes can be maintained and reconfigured.

Connecting a WSN to the Internet has the benefit that the sensor data can be accessed from virtually everywhere. But since wireless sensor nodes usually use special communication protocols, a gateway node is needed, which negotiates between the WSN and the IP-based network. Sensor node operating systems with direct support of TCP/IP also exist, notably Contiki, which is introduced in Section 3.2. This not only makes it possible to directly access the nodes with commonly used tools and applications (e.g. Telnet [4]). The nodes themselves are able to communicate with every other network entity, either directly or via other nodes. They could receive updates from an administration tool, upload measurement data to a Web server, or send

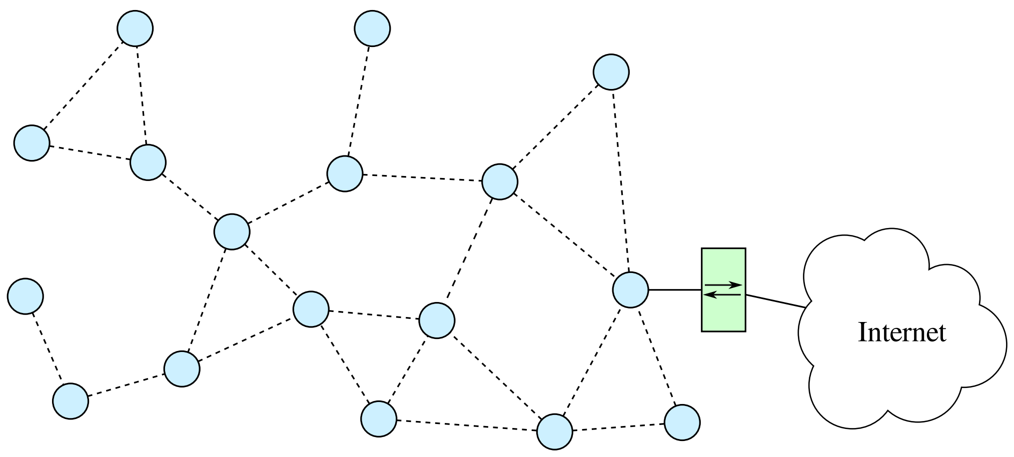


Figure 2.1: Wireless Sensor Network example topology

out e-mails notifying about low battery level, to name just a few examples. Using TCP/IP in a WSN, however, brings the drawback of significant performance decrease, due to reasons we illustrate in the following.

2.2 TCP/IP in Wireless Multi-hop Networks

The TCP/IP protocol suite is the de facto standard for Internet communication. The Transmission Control Protocol (TCP) is implemented on top of the Internet Protocol (IP) and provides reliable delivery of an ordered stream of bytes from one application to another.

After an initial three-way handshake between client and server, the two hosts can independently transmit data to each other. For transmission, the data is split into segments, which are sent to the recipient as individual TCP packets. As it is not guaranteed that each packet takes the same route, the order of arrival may be incorrect. Therefore, the TCP packet header contains a 32 bit sequence number, which increases for each packet by the size of the previous segment. This allows the TCP stack at the receiver side to reorder the received segments before passing them to the application.

The sequence number is also used to provide reliability. Whenever one host, which has already received data, sends a packet to the other, it acknowledges previously received transmissions. This is done by setting the acknowledgment flag in the packet's TCP header and specifying the sequence number of the next packet expected to be received as acknowledgment number. If an incoming packet contains an acknowledgment number for an already sent packet, the host assumes that all previously sent packets with a sequence number greater or equal to this acknowledgment number have been lost. The host will then retransmit these packets. Figure 2.2 shows an example of this behavior. Here, host *A* successfully sends two segments to host *B* that, in return, sends three segments. Unfortunately, *B*'s second segment is lost. *A* acknowledges only the first segment to indicate the need of a retransmission. Finally, when the remaining segment has been received and there is no missing data anymore, *A* sends an acknowledgment without any payload to *B*.

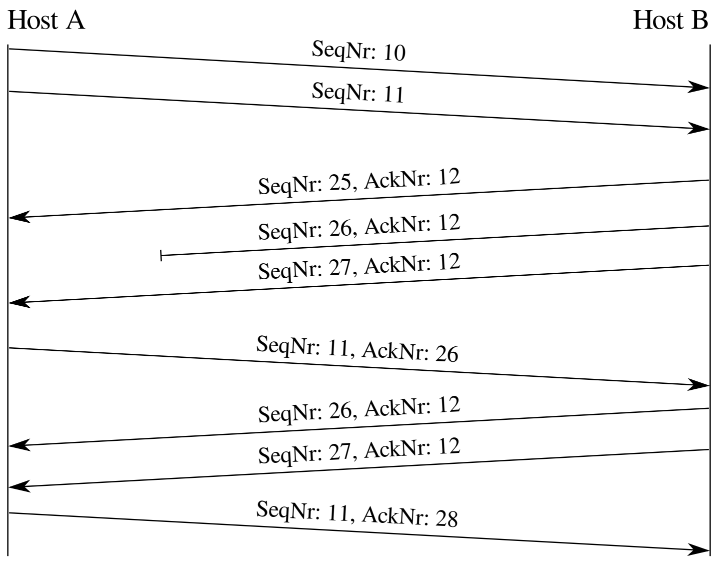


Figure 2.2: TCP packet flow with retransmissions

A first disadvantage of using TCP in wireless sensor networks emerges. A lost segment has to be retransmitted by the original source regardless where the loss actually happened. The nodes on the path between source and destination do not have the possibility to perform a retransmission by themselves. Therefore, the same information has to pass all nodes on a path multiple times, which is a not only a waste of time and consequently throughput, but also energy, a valuable resource for wireless sensor nodes.

In wired networks, for which TCP has originally been designed, this is not of great concern since packet loss caused by transmission errors is a rather rare event. If packet loss does happen, the major reason is network congestion, the situation where there are too many packets on the fly, causing buffer overflows in intermediate nodes, as opposed to lossy links in wireless networks. This is why TCP applies congestion control mechanisms. In each TCP header a host specifies its current *window size*. This is the amount of bytes the host is able to receive at once. When a host starts to transmit data, it will not initially use the full capacity of the receiver's window size. Instead, it will just send one single segment. For each subsequent acknowledgment, the host doubles the amount of simultaneously transmitted segments up to a sender-defined threshold. This behavior, illustrated in Fig. 2.3, is called *slow start* and should ensure that the capacity of the network is not exceeded at the beginning of a connection. This phase is followed by the *congestion avoidance* phase, in which the amount of data sent is only increased by one segment per acknowledgment until the receiver's window size has been reached. When an expected acknowledgment does not arrive in a certain time frame, the sender drops back to transmit only one single segment and restarts the *slow start* phase with a threshold half the previous size. This allows the network to quickly recover from a congestive situation.

In large networks TCP's congestion control mechanisms are essential for flawless communication. However, these mechanisms as well as the end-to-end retransmission scheme have a significant negative impact on the performance of wireless multi-hop networks. Here, the packet

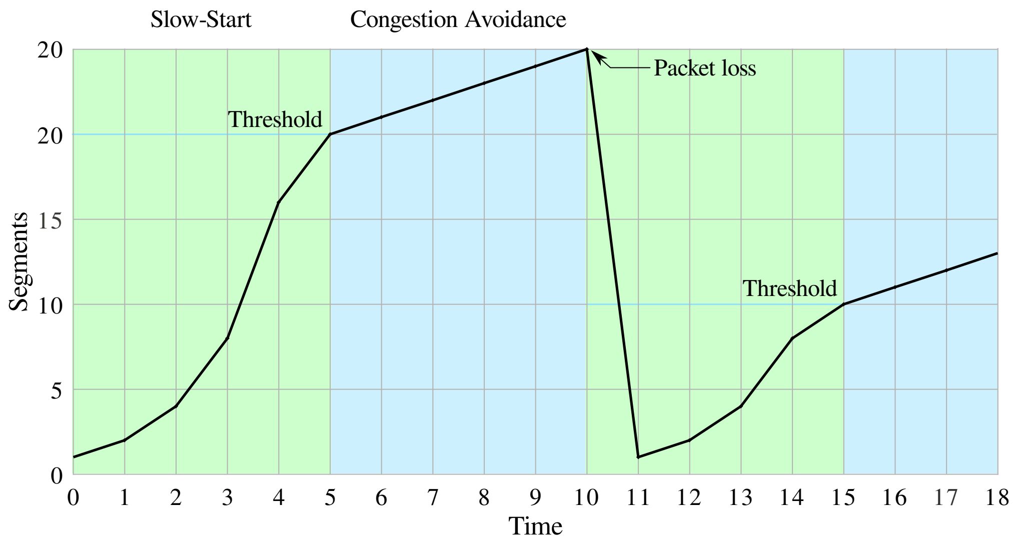


Figure 2.3: TCP congestion control phases

loss rate is much higher than in wired networks, thus causing the congestion control mechanism to frequently reduce the window size. For indoor WSNs packet loss rates of more than 30% on a single hop are not unusual [5]. The major reason for this is not network congestion, however, but the occurrence of transmission errors caused by signal distortion due to interference with other devices, interference due to multipath propagation, and packet collisions. Packet collisions occur because of the half-duplex nature of the wireless media, i.e. the fact that a device cannot transmit and receive simultaneously. Contention-based random MAC protocols lower the probability of such an event, but are not able to completely prevent two devices from initiating a transmission at the same time. Furthermore, it is not possible for a device *A* to detect, whether a neighboring device *B* is already receiving data from a third device *C*, if *A* is outside the transmission range of *C*. This situation is known as the hidden node problem and is also a source of packet collision.

Such kind of phenomena hardly exist in wired networks, where the link layer error rate generally remains very low, e.g. with bit-error-rates of at maximum 10^{-7} for DSL [6].

2.3 Contributions

An overview of the key contributions of this thesis is provided in the following:

- We implemented an approach of caching TCP segments on intermediate wireless sensor nodes to make them able to perform local retransmissions in case of packet loss. The approach is transparent to the application layer, does not require any modification to the TCP protocol, and works independently from the used MAC protocol. This is the first implementation of TCP optimization mechanisms based on DTC [7] and TSS [8] for real sensor nodes.

- Besides the basic retransmission mechanism, we introduced and implemented various extensions, which provide alternative retransmission schemes. The extensions allow the sensor nodes to perform multiple retransmissions of the same TCP segment to cope with subsequent transmission errors, use the hop-distance as an additional value for calculating the retransmission timeout, and make use of MAC layer feedback to provide early retransmissions in case of low external radio activity. Additionally, we investigated in an extension, which provides multiple TCP connections to transport data.

- We extensively evaluated the implementations with experiments run in a real world sensor node testbed under realistic conditions. We tested our approaches in combination with an energy-unconstrained CSMA MAC protocol, as well as with three energy-efficient, radio duty-cycling MAC protocols.

- Our experiment results revealed the feasibility of improving TCP's performance in wireless sensor networks using the proposed local retransmission schemes. Furthermore, we showed that the results are heavily depending on the used MAC protocol, the hop distance between sender and receiver, and the chosen retransmission scheme. In the best case, in combination with the CSMA MAC protocol, an average increase in throughput of 84% has been achieved in experiments across 2, 3, 4, 5 and 6 hops.

2.4 Thesis Outline

This thesis is structured as follows: Section 3 gives an overview of the technologies related to our project, such as the used sensor node operating system, network protocols, and sensor node platform. Other studies, which pursue objectives similar to ours, are also introduced. Section 4 discusses design issues and the actual implementation of our own caching and retransmission mechanisms. The experiment setup and the evaluation of the gained results are introduced in Section 5. Finally, Section 6 concludes with a summary of the presented work and provides an outlook to future work.

Chapter 3

Related Work

In this chapter we give an overview of studies related to our project, and present the used software and hardware components. In Section 3.1, we outline other approaches available that aim to improve the performance or efficiency of TCP/IP in wireless sensor networks. Then, we introduce the Contiki operating system in Section 3.2 focusing on node-to-node communication as well as TCP/IP communication. This is followed by Section 3.3, a presentation of the various MAC protocols we used to evaluate the impact of our implementation. Finally, there is a brief description of the used sensor node platform and testbed management software in Section 3.4.

3.1 TCP/IP in Wireless Networks

As discussed in Section 2.2, the performance of TCP/IP based wireless networks degrades due to TCP's congestion control mechanism and end-to-end retransmission scheme. This is especially true for wireless multi-hop networks. Several approaches concerning this issue have been proposed, such as Snoop [9] on which many others rely. However, we are only aware of two studies explicitly targeting wireless sensor networks: DTC [7] and TSS [8]. The simulation-based studies [7] and [8] have shown that both of them are able to improve throughput while reducing energy consumption. A drawback DTC and TSS have in common is that both protocols are not designed to work in combination with duty-cycling MAC protocols.

3.1.1 Snoop

Snoop [9] aims to improve TPC/IP performance in 802.11-based wireless local area networks, where mobile devices are directly connected to a base station, which itself is connected to a wired network. An example topology is illustrated in Fig. 3.1. The base station monitors ongoing TCP connections and caches all segments sent from the wired network to a mobile device until TCP acknowledgements (ACK) for these segments return. If a packet loss is detected on the wireless link, either due to a timeout or due to duplicate ACKs, the base station will retransmit the affected segments and drop duplicate ACKs. The host in the wired network will therefore not be aware of any transmission errors on the wireless link, which it would interpret as a result of network congestion. This prevents from performance degeneration caused by unnecessarily triggered congestion control mechanisms.

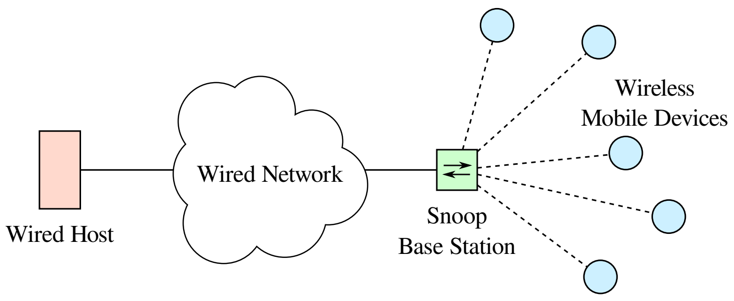


Figure 3.1: Snoop example topology

Snoop also proposes to modify the mobile devices' TCP implementation to understand TCP segments containing a Selective Acknowledgment (SACK) block. The TCP SACK option, proposed in RFC2018 [10] and not commonly supported when Snoop was introduced, provides the possibility for a TCP receiver to inform the sender, which individual segments out of a sequence have actually been received. Traditionally, even when only one segment gets lost, this segment and all subsequent ones have to be retransmitted as the receiver can only respond with an acknowledgment for a whole sequence. Using this additional feature, a base station that detects a packet loss can immediately reply with a SACK, which reduces the amount of retransmissions.

Furthermore, Snoop base stations retransmit selective acknowledgments when they do not receive any packet from a mobile device for a certain amount of time. If the mobile device has sent out a sequence of packets, none of which has been received by the base station, such a spurious SACK reduces the waiting time for the retransmission.

3.1.2 Distributed TCP Caching for Wireless Sensor Networks

Distributed TCP Caching for Wireless Sensor Networks (DTC) [7] is a mechanism which caches TCP segments on forwarding sensor nodes to be able to perform hop-by-hop retransmissions. The ultimate goal of DTC is to reduce energy consumption of the sensor nodes.

In DTC, each sensor node keeps a cache for a single TCP segment. A cache capable of storing multiple segments was assumed to exceed a sensor node's memory. Whenever such a segment is forwarded, it is cached with a probability of 50% if the cache is not already in use. This allows a node to react to TCP acknowledgments (ACK) accordingly to Fig. 3.2: If the acknowledgment number of an incoming segment is greater than the sequence number of the cached segment, the cached segment is removed and the TCP ACK is forwarded as is, since this represents a normally behaving TCP packet flow. However, an acknowledgment number smaller than or equal to the cached segment's sequence number is an indicator for packet loss.

If the acknowledgment number is equal to the cached sequence number, the node simply retransmits the cached segment. Otherwise, DTC additionally makes use of TCP's Selective Acknowledgment Option (SACK). If the SACK block of an incoming TCP ACK packet already contains the sequence number of the cached segment, the node knows that another node, further towards the sender of the ACK, has cached the same segment as well. In that case, the node clears the cache and forwards the ACK. Should the sequence number not be part of the SACK block, however, the node retransmits its cached segment and adds the corresponding sequence

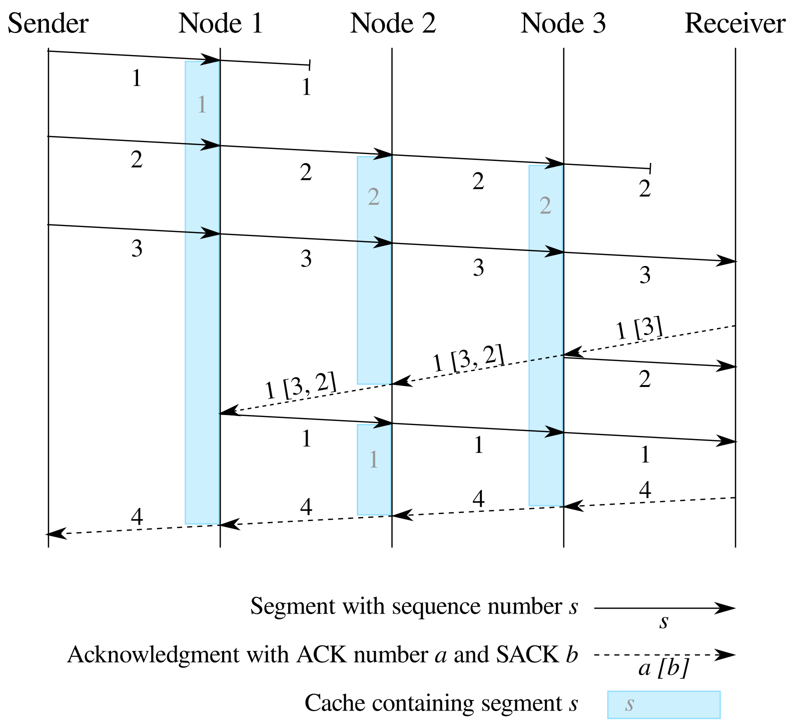

Sender
Node 1
Node 2
Node 3
Receiver
1
2
3
1 [3]
1 [3, 2]
4
Segment with sequence number *s*
Acknowledgment with ACK number *a* and SACK *b*
a [b]
Cache containing segment *s*
s

Figure 3.2: Segment caching and retransmission in DTC

number to the SACK block. If all gaps of the SACK block are filled the node drops the ACK since then all unacknowledged segments are already cached by other nodes. Otherwise, the ACK is forwarded as usual.

In addition to observing the TCP ACK packets, DTC nodes relay on link level acknowledgments. Should a node not receive a link level ACK for a forwarded TCP segment, a retransmission timer is scheduled. If no acknowledging TCP packet is received before the retransmission timer expires the node retransmits the cached segment as well.

OMNet++ simulations, replicating a 6 hop topology with a 10% packet loss rate, have shown, that using DTC the amount of TCP packets required to transport a certain amount of data is indeed reduced. Furthermore, a 450% increase in throughput has been measured.

3.1.3 TCP Support for Sensor Networks

Similarly to DTC, in TCP Support for Sensor Networks (TSS) [8] the sensor nodes cache forwarded TCP segments to be able to perform local retransmissions. TSS nodes maintain a cache for two segments. The first segment has already been forwarded to the next node. It remains cached until the node gains knowledge about the successful reception of the segment at the subsequent node. This can be either explicitly, when the node receives a TCP ACK for this segment, or implicitly, when it overhears the forwarding of the segment by the subsequent node. When an acknowledgment for a cached segment does not arrive after 1.5 times the average round trip time, the node assumes that the segment has been lost and performs a local retransmission.

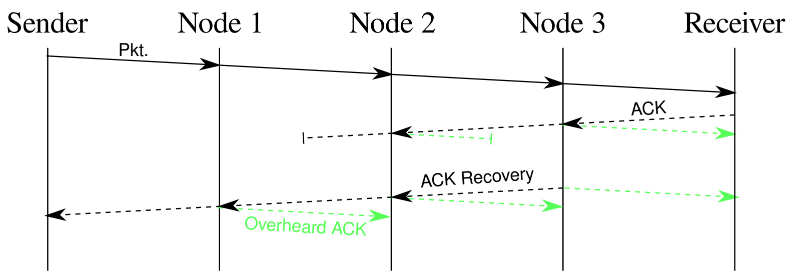


Figure 3.3: TCP acknowledgment recovery in TSS

The second segment has yet to be transmitted. However, this happens only after an (implicit or explicit) acknowledgment has arrived for the first segment. This behavior guarantees, that segments are not transmitted towards a node, for which it is not known whether its cache is already full. Consequently, when one node stops forwarding segments, the preceding nodes will stop transmitting as well.

Concerning TCP ACKs, TSS performs local regeneration of acknowledgments. Whenever a TCP data segment is received, for which the node has already received a TCP ACK, the data segment is dropped and an acknowledgment is regenerated using the highest acknowledgment number ever seen. Additionally, to reduce the delay of acknowledgments, TSS uses a recovery mechanism when it is assumed that a TCP ACK has been lost, which is illustrated in Fig. 3.3. Since TSS nodes are assumed to overhear the communication of other nodes, a node can measure how long it takes from the time it sends out an acknowledgment until it is forwarded again by the recipient. If this delay is larger than twice the average value, the acknowledgment is regenerated and transmitted again.

The performance of TSS has been evaluated in OMNet++. In a sensor node topology with a packet loss rate of 15%, TSS reduced the amount of transmissions needed to deliver 500 data packets over 9 hops by roughly 70%, and almost removed the occurrence of end-to-end retransmissions.

3.2 Contiki

Contiki [11, 12], first released in 2004, is an open source operating system designed for sensor nodes and embedded systems with small amounts of memory. Typically, a configuration requires 2 kB of RAM and 40 kB of ROM. On top of an event driven kernel, Contiki applications can be loaded and unloaded dynamically at runtime. This makes it possible to distribute program code to running sensor nodes to add new functionalities or provide bug fixes without the need of collecting the nodes for offline reprogramming.

To be able to communicate with other devices, Contiki uses a layered network stack as illustrated in Fig. 3.4. At the bottom, providing access to the physical medium, is a platform dependent radio driver. On top of this lies the MAC layer. Several energy efficient, and performance orientated MAC protocols are already implemented, some of which are further discussed in Section 3.3. Finally, there is the Rime layer, Contiki's major protocol stack, providing different methods of node-to-node communication, followed by the optional µIP layer.

3.2.1 Rime Protocol Stack

Rime [13] is a collection of core functionalities for node-to-node communication, so called Rime protocols. The Rime protocols are implemented in different layers. Lower layer protocols provide more basic features used by higher layer, more complex protocols in a non-transparent way. Usually in protocol stacks, outgoing packets acquire additional headers while traveling through the different layers, which are removed again at the receiver when being processed as an incoming packet in the opposite direction. In Rime, however, incoming packets are not modified. Higher layer protocols are therefore aware of the underlying layers and can benefit from additionally available header information.

Applications and other protocols running on top of Rime may directly use any Rime protocol to transmit data. To do so, they have to create a connection using the desired protocol, copy the data to the *Rime buffer* and initiate the actual transmission. The available protocols, illustrated in Fig. 3.4, and their responsibilities are as follows:

abc The most bottom layer, used by all other Rime protocols, is the Anonymous Best-effort Single-hop Broadcast protocol. It is used to send a data packet to all other listening nodes in the receiving range without providing any information about the initiator of the transmission.

ibc To be able to send packets which identify the sender, the Identified Best-effort Single-hop Broadcast protocol is needed. It simply attaches the address of the sending node to the buffered data packet.

uc For sending a packet to just one particular node, the Best-effort Single-hop Unicast protocol attaches the address of the target receiver to the packet. Nodes receiving a packet addressed to another node, will discard it at this layer.

stuc When data is sent using the Stubborn Single-hop Unicast protocol, the packet is cached in a special buffer and is indefinitely sent and resent until an upper layer protocol explicitly stops the transmission.

ruc The Reliable Single-hop Unicast protocol can perform reliable transmissions. Whenever a packet arrives at this layer, an acknowledgment is generated and sent back to the originator. An incoming acknowledgment therefore indicates the successful delivery of the previous packet, which causes the *ruc* layer to tell the underlying *stuc* layer to stop performing retransmissions.

mh, rmh Best-effort Multi-hop Unicast (*mh*) and Hop-by-hop Reliable Multi-hop Unicast (*rmh*) can both be used to send data to a receiver over a multi-hop path. It should be noted, that the actual path finding is not part of Rime. Both protocols have to be provided by the calling application or protocol with a function for choosing the next-hop node of a path. However, Contiki provides a generic approach to build a mesh network which itself is able to provide this functionality on demand.

polite, ipolite The Polite Single-hop Broadcast protocol is a special purpose, gossip based broadcasting mechanism. The main idea is avoid multiple transmissions of identical packets in a node's vicinity. This can occur, for instance, when using broadcasting schemes with negative acknowledgments (NACK), where multiple nodes transmit identical NACKs at the same time as reaction to a lost broadcast. The protocol is available for anonymous usage (*polite*) or in combination with a source identifier (*ipolite*).

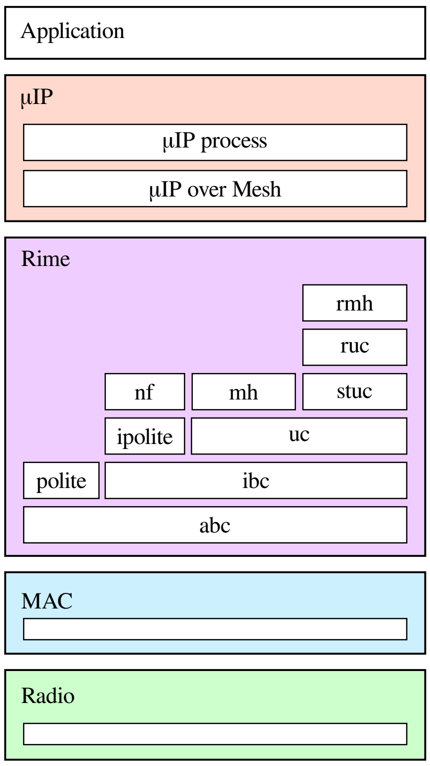


Figure 3.4: Contiki's network stack

nf Best-effort Network Flooding can be used to distribute a single packet to all nodes in a network. It relies on the underlying *ipolite* protocol to reduce the occurrence of redundant transmissions. Contiki itself uses *nf* for path finding when establishing a mesh network.

3.2.2 µIP Stack

Already integrated in the latest versions of Contiki is µIP [14], a minimalistic implementation of the TCP/IP suite, which uses only few amounts of memory and RAM. Besides IPv4, IPv6, and TCP, µIP also supports UDP and ICMP, which is, for instance, required to be able to react to IP related network errors. Other parts of the suite, not vital for TCP/IP or UDP/IP communication (e.g. ARP), were left out in order to reduce code size. But since µIP implements all requirements for Internet hosts, specified in RFC1122 [15], which are related to host-to-host communication, flawless exchange of data should be possible with almost any device providing a full-featured TCP/IP stack. An example for an excluded feature of µIP compared to RFC1122 is that the stack does not allow the transmission of multiple unacknowledged TCP segments. On the one hand, this results in a decrease of the maximal possible throughput. On the other hand, there is no need of a sliding window mechanism, which requires a buffer to store sent, but yet unacknowledged segments, therefore saving valuable memory.

All major functionalities are implemented in µIP's so called *main control loop*. Depending on the implementation, this loop either runs continuously, or each cycle is initiated by a certain event, as it is the case with Contiki as an event-driven operating system. Such an event could either be a packet received from the network or the expiration of a periodic timer. The work flow of the main control loop for handling TCP/IP connections, excluding connection initiation and termination handshakes, is illustrated in Fig. 3.5. UDP packet processing is not illustrated but works analogously.

In order to establish a new TCP connection, an application has to initiate the three-way handshake. This is done by calling the *uip_connect* function. When the connection has been established successfully, the application gets a notification. Until the connection has been closed again, a TCP timer will periodically trigger a cycle of the main control loop to maintain the open connection. The first step of this maintenance process is to check whether the connection has timed out (i.e. no packets have been received for a certain amount of time). If this is not the case, the application is polled for new data. Polling is needed because applications running on µIP are not allowed to send data without explicit permission. If there is a TCP timeout, however, and a previously transmitted packet is still unacknowledged, a retransmission has to be performed. Since outgoing packets are not cached by µIP itself, the application has to do the retransmission.

When the application provides new data, either after being polled or asked for a retransmission, µIP generates TCP and IP headers (e.g. source and destination addresses, port numbers, checksums, etc.) accordingly to the connection state and passes the resulting packet to the underlying layer (i.e. Rime) for transmission.

After the arrival of a new packet from the network, the upcoming main control loop cycle will process the given data. First, the IP headers are validated. Invalid packets are dropped. The next step is to process the packet according to its payload type. This could either be ICMP, UDP or TCP. Packets with an unknown payload type are dropped.

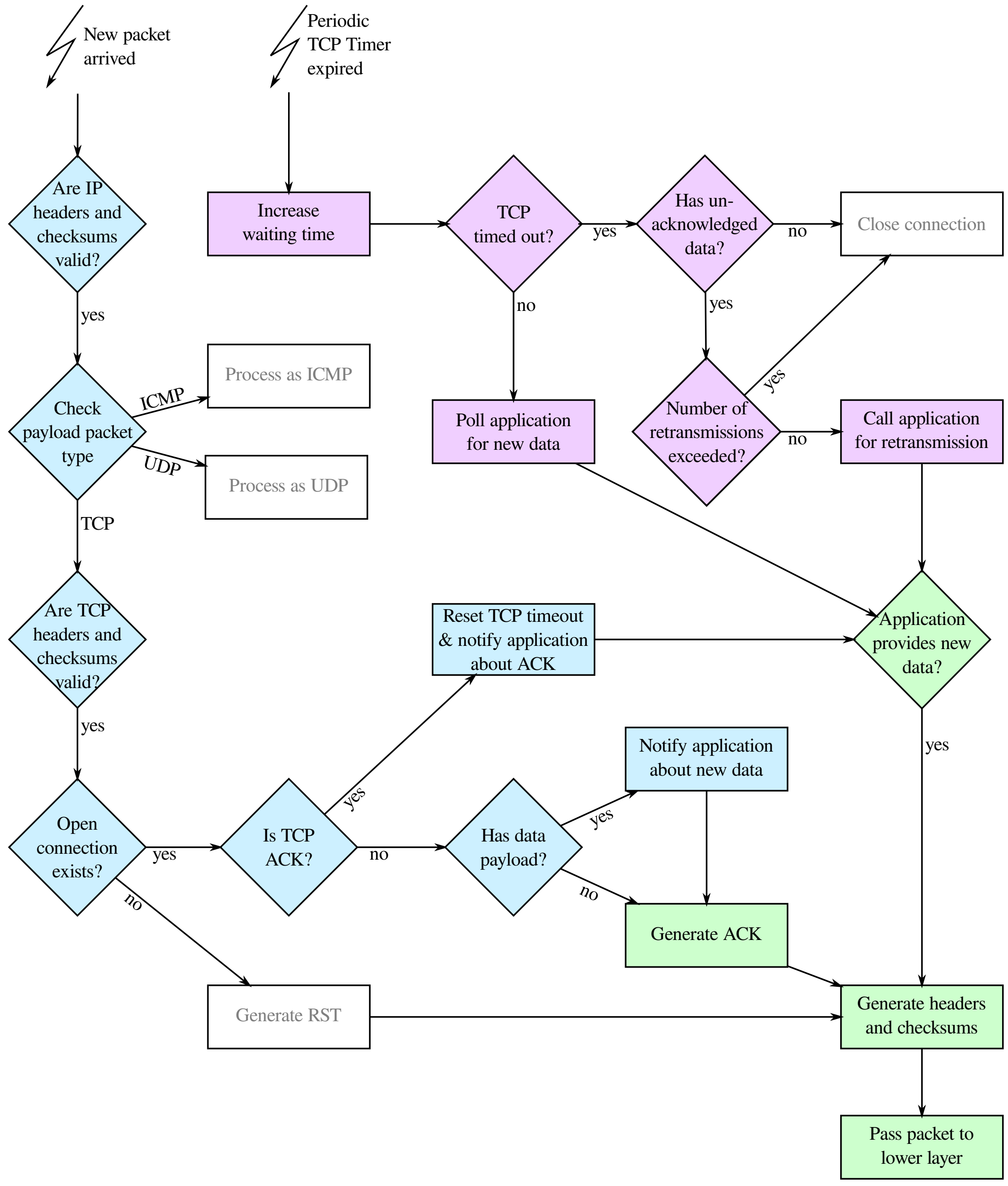

New packet arrived
Periodic TCP Timer expired
Are IP headers and checksums valid?
yes
Increase waiting time
TCP timed out?
yes
no
Has un-acknowledged data?
no
yes
Close connection
Check payload packet type
ICMP
UDP
TCP
Process as ICMP
Process as UDP
Poll application for new data
Number of retransmissions exceeded?
yes
no
Call application for retransmission
Are TCP headers and checksums valid?
yes
Reset TCP timeout & notify application about ACK
Application provides new data?
yes
Open connection exists?
yes
no
Is TCP ACK?
yes
no
Has data payload?
yes
no
Notify application about new data
Generate ACK
Generate RST
Generate headers and checksums
Pass packet to lower layer

Figure 3.5: Simplified µIP control loop, including connection maintenance (*purple*), incoming packet processing (*blue*), and outgoing packet generation (*green*).

In the case of a TCP payload, the TCP header fields are validated and it is checked, whether a matching TCP connection is already open. If this is not the case, a TCP packet with its connection reset flag (RST) set is generated and sent back to the originator of the bogus packet. A proper packet is further processed. If it is an acknowledgment for previously sent data, the application is notified. This permits the application to immediately respond by providing data to µIP, which is then transmitted in the current control loop cycle. The application is also informed, if the incoming TCP packet contains data payload. Additionally in this case, µIP generates a corresponding TCP acknowledgment as a response.

3.2.3 The Rime µIP Interface

The target receiver of an IP packet can either be a node participating in the sensor network, or a device of another network. In the latter case, one Contiki node has to be connected to a host device (e.g. via Serial Line Internet Protocol to a computer), which acts as a router between the sensor network and any other network (e.g. the Internet). This node then becomes a gateway to which all IP packets addressed with an IP address belonging to another subnet are forwarded. Typically, when using IPv4, Contiki nodes live in the private 172.16.0.0/16 subnet. This means, that the most significant sixteen bits are fixed (i.e. 172.16). The least significant bits are defined by the node's 2 byte Rime address. As example, an IP packet addressed to 172.16.8.2 will be delivered to the node with a Rime address of 8.2.

In order to send and receive IP packets within a sensor network, µIP has to rely on Rime. But since Rime and µIP are not aware of each other, Contiki adds an additional bottom layer to the µIP stack, called *µIP over mesh*, through which all ingoing and outgoing IP traffic flows. µIP sees this layer as a network interface, whereas for Rime it is a high layer protocol.

Outgoing IP packets are encapsulated in Rime packets by *µIP over mesh* and sent using Rime's best-effort single-hop unicast protocol (uc). Therefore, the Rime packets require a defined receiver address. In multi-hop networks, the receiver address only belongs to the target receiver on the very last hop. Otherwise, this has to be the address of the next node in the path. To know to which node *µIP over mesh* actually has to send an IP packet, it uses Contiki's built-in routing mechanism.

3.3 Media Access Control Protocols

Media Access Control (MAC) protocols build the link between the network layer and the physical layer, i.e. the radio driver for wireless devices. For wireless sensor networks, we distinguish between duty-cycled, often referred as energy-efficient (E^2) MAC protocols, and non-duty-cycled MAC protocols. Radio transceivers usually are the most power consuming components on a sensor node, not only while transmitting, but also while receiving.

In fact, some transceivers even consume more energy in receive mode than in transmission mode, depending on the transmission power setting of the radio chip. As example, the TelosB sensor node platform's radio consumes up to 19.7 mA in receive mode and only between 8.5 and 17.4 mA in transmission mode using the lowest (-25 dBm) and highest (0 dBm) transmission power, respectively. With the radio turned off, a TelosB node draws 2.4 mA [16].

To overcome this issue, duty-cycled protocols maintain a regular sleep / wake-up cycle in which they have long periods with the radio transceiver turned off (sleep state) to save energy. Examples for such energy saving protocols are B-MAC [17], S-MAC [18], T-MAC [19], WiseMAC [20], X-MAC [21], ContikiMAC [22], and LPP [23]. Recent contributions to this protocol class are the runtime traffic adaptive MAC protocols MaxMAC [24] and BEAM [25]. Since we use X-MAC, ContikiMAC and LPP in our experiments, we describe their basic operation and design principles in the following.

3.3.1 X-MAC

X-MAC [21] is a duty-cycled MAC protocol designed to be used in Wireless Sensor Networks. To be able to transmit data to another sensor node, duty-cycled protocols need to know, when the node, which should receive the data, is awake. Therefore, X-MAC transmits a preamble strobe before the actual data transmission starts. This will alert the receiver in one of its periodic wake-ups.

Typically, preamble-based approaches [17] transmit a known bit sequence for at least the duration of a sleep period, or a shortened preamble just when the receiver is about to wake up [20]. A node receiving such a preamble will stay awake until the data transmission starts. This approach, however, has some crucial disadvantages. Both, sender and receiver have to stay awake until the preamble transmission has completed. In the worst case this takes as long as one sleep period. Furthermore, other nodes hearing the preamble also have to stay awake to receive the first data packet only to recognize that the data is not addressed to them.

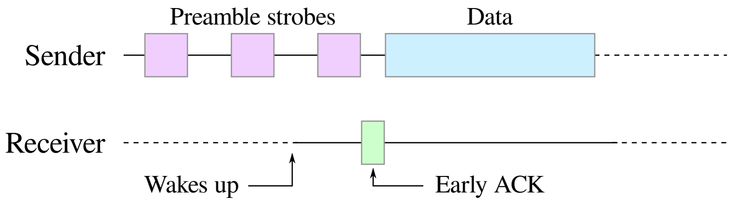


Figure 3.6: Strobed preamble in X-MAC

X-MAC presents a solution for both of these problems by introducing a short, strobed preamble. Instead of just having an arbitrary bit sequence as preamble, X-MAC directly embeds the address of the target receiver into the strobes. Therefore, other nodes receiving such a preamble can immediately go back to sleep. The idea of a strobed preamble is to transmit multiple short preamble packets delimited by short pauses. As illustrated in Fig. 3.6, a receiver can now stop the preamble strobing process by sending an early acknowledgment, which signals reception readiness. This not only reduces the energy consumption by reducing unnecessary waiting times the nodes have to undergo, it also reduces the hop-by-hop latency.

In its original implementation in [21] on top of the MANTIS OS, X-MAC additionally uses a traffic estimation algorithm to be able to adapt the duty-cycle period to varying traffic loads. When operating in sensor networks with alternating phases of high and low traffic load, this brings a better trade-off between throughput and energy saving than having statically configured duty-cycle timings. However, the current X-MAC implementation for Contiki does not support traffic load adaptation. The wake-up interval is statically defined at compile time and kept unchanged.

3.3.2 ContikiMAC

ContikiMAC [22] is the successor of X-MAC in Contiki and combines key concepts used in various other MAC protocols. Borrowed from X-MAC, ContikiMAC relies on short, strobed preambles to notify about an upcoming transmission. But instead of having a rather simple bit sequence as preamble, containing only the target address as actual information, ContikiMAC directly uses data packets as strobes. When a node wakes up and detects an ongoing transmission, it stays awake to receive one complete packet and returns an acknowledgment. This means that at minimum, only three transmissions take place to deliver data small enough to fit in one packet. The idea of using data packets as wakeup signal has also been used in BoX-MAC [26] and BEAM [25].

With the reception of an acknowledgment ContikiMAC nodes learn the other node's wakeup cycle. This highly effective wake-up schedule exchange mechanism was first introduced in WiseMAC [20]. Knowing when the receiver is going to wake up, the sender can minimize the amount of strobe packets needed to deliver a packet. It just has to start the transmission shortly before the receiver wakes up. This behavior, illustrated in Fig. 3.7, is beneficial because requiring less strobes reduces the per data packet overhead and thereby the chance of collisions what increases performance and decreases energy consumption.

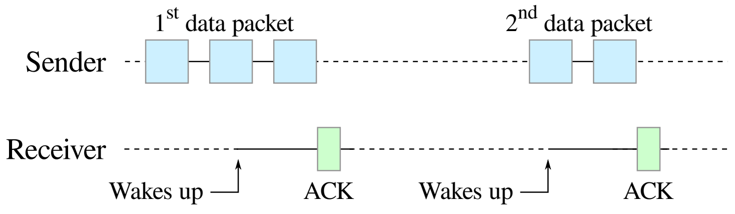


Figure 3.7: Wakeup cycle learning in ContikiMAC

3.3.3 Low Power Probing

As X-MAC and ContikiMAC, Low Power Probing (LPP) [23] is a duty-cycled MAC protocol. In contrast to most duty-cycled protocols, not the initiator of a transmission is sending strobes, but the recipients. All nodes periodically send out probe packets indicating that they are ready to receive data. As shown in Fig. 3.8, a node which wants to send something wakes up and waits for the target receivers to transmit their probes. Each received probe is acknowledged to notify the receiver to stay awake. When all target nodes are known to be listening, the sender transmits the data.

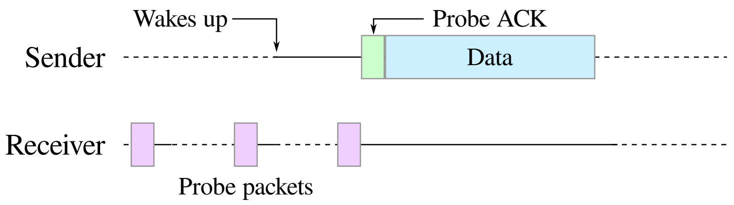


Figure 3.8: Receiver initiated sending in LPP

This behavior is beneficial especially for broadcasting. Using LPP, a sender can minimize its idle listening period, since it actually knows when all receivers are ready. In other duty-cycled protocols, such as the two previously mentioned, a node broadcasting to multiple nodes has to transmit a preamble for the maximum amount of time, i.e. one duty cycle, to ensure successful delivery to all receivers.

A similar receiver initiated approach is used in RI-MAC [27]. In addition to LPP, RI-MAC has a more efficient beaconing mechanism which, for example, is able to recover from collisions. When multiple senders start transmitting at the same time and the receiver detects a collision, it retransmits a beacon. The senders will then wait for a random timespan before retransmitting the data.

3.3.4 NullMAC

Another MAC protocol available for Contiki is NullMAC [28]. It is a minimalistic protocol which just passes data packets from the network layer to the radio driver and vice versa. This implies that NullMAC does not check the radio channel for activity before sending a packet. NullMAC should therefore preferably be used in combination with Contiki's Carrier Sense Multiple Access (CSMA) layer to avoid collisions, which we did for the evaluations presented in this thesis.

Furthermore, NullMAC does not duty-cycle the radio. The resulting high energy consumption is, however, compensated with achieving the maximum throughput possible. This makes NullMAC ideal to compare the impact of modifications of higher network layers to the transmission performance.

3.4 Experiment Resources

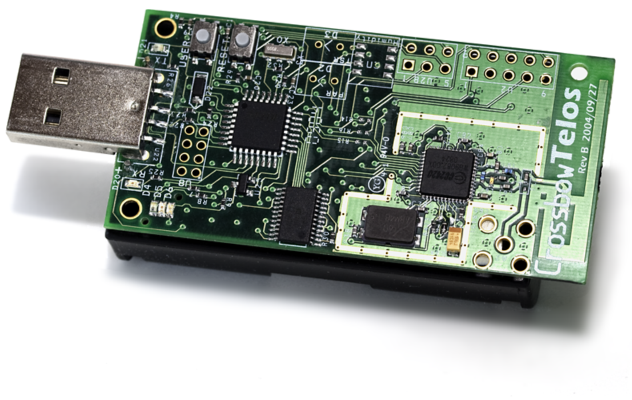
For evaluating our proposed protocol mechanisms throughout the case of this thesis we used TelosB [29] sensor nodes, which are part of a local testbed deployed at the Institute of Computer Science and Applied Mathematics (IAM) at Neubrückstrasse 10 in Bern. The sensor nodes are accessed via the TARWIS [30] testbed management system. Since both, the TelosB nodes and TARWIS, are key components of our experiments, we provide a brief introduction in the following.

3.4.1 TelosB Sensor Node Platform

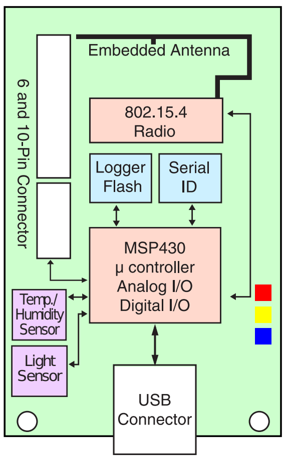
The TelosB [29] sensor nodes, one of is illustrated in Fig 3.9, are produced by Memsic (formerly Crossbow) and follow the same open-source platform design as Sentilla's Tmote Sky. It features an 8 MHz MSP430 microcontroller including 10 kB RAM and 48 kB ROM, a CC2402 radio module, 1 MB flash memory, and two optional sensors for measuring light, temperature and humidity. Additional devices may be attached using the provided 6 and 10 pin connectors.

The radio module, which operates at 2.4 GHz, has an outdoor range of up to 100 m (30 m indoors) and is compliant to IEEE 802.15.4. In heterogeneous sensor networks, this allows the node to communicate with different types of nodes, as long as they support the same standard.

When connected to a computer by USB, the node provides a Universal Asynchronous Receiver Transmitter (UART) interface through which the node can be programmed. This interface can also be used to communicate with the currently running operating system on the sensor node to collect sensor data, for example.



(a) Node with attached AA battery pack



(b) Block diagram [29]

Figure 3.9: TelosB sensor node

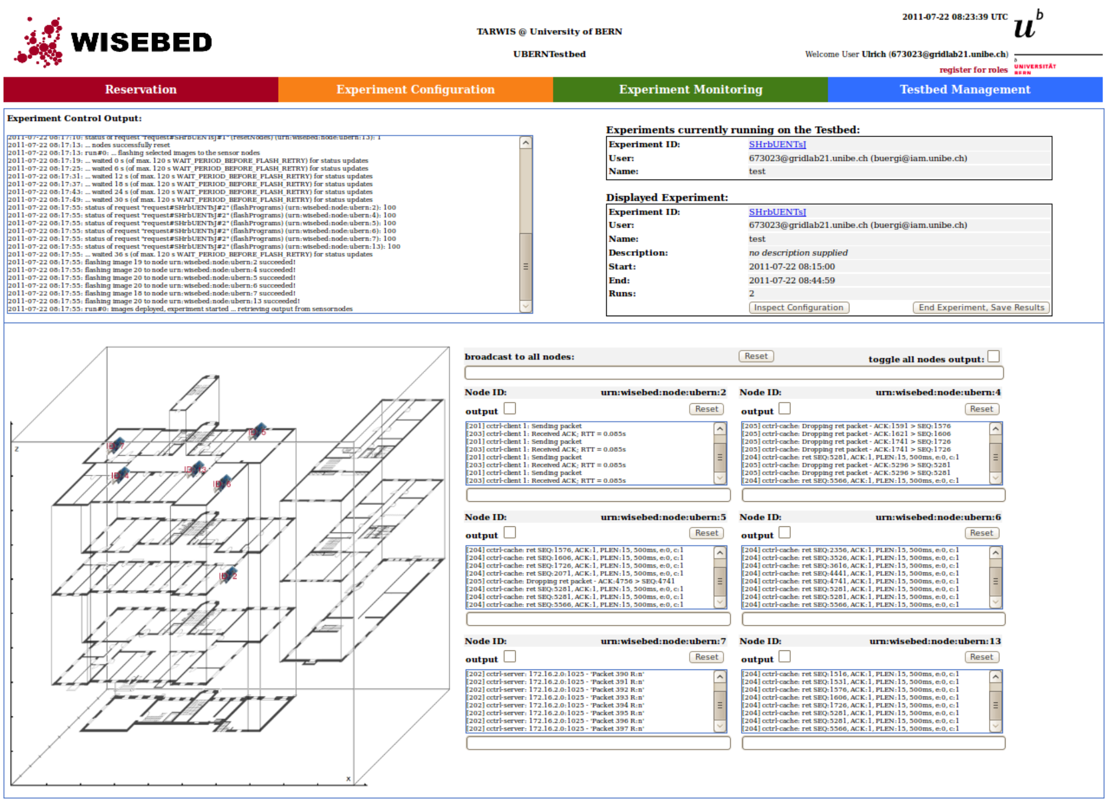
3.4.2 TARWIS

The TelosB sensor nodes, used for the experimental evaluation of our proposed protocol mechanisms, are part of the local WISEBED [31] sensor node testbed, which is deployed at our institute. For scheduling, configuration and maintenance of the experiments, TARWIS, a testbed management architecture for wireless sensor network testbeds [30] was used.

The TARWIS system is designed to work independently from the architecture of the managed testbed, the deployed sensor nodes, and the sensor nodes' operating systems. As long as the nodes can be controlled remotely by the TARWIS server, it does not matter, whether the sensor nodes are directly connected to a single server, are controlled by gateway mesh nodes, or act independently without wired backup channel.

Guided by a web interface, the user can schedule and monitor custom experiments. First, the compiled, binary sensor node images have to be uploaded to the web server. Then, in the reservation section, the user can select the desired sensor nodes to be used for the experiment, and choose the points in time at which the experiment should start and end. Due to the multi-user design of TARWIS, when not all available sensor nodes are reserved for the usage in an experiment, the remaining nodes can be used by other users in their own experiments scheduled to run during the same period of time.

After the reservation, the scheduled experiment has to be configured. For each selected sen-



Experiment Control Output:

```
2011-07-22 08:17:10: status of request "request#SHrbUENTsJ#1" (resetNodes) (urn:wisebed:node:ubern:13): 1
2011-07-22 08:17:13: ... nodes successfully reset
2011-07-22 08:17:13: run#0: ... flashing selected images to the sensor nodes
2011-07-22 08:17:19: ... waited 0 s (of max. 120 s WAIT_PERIOD_BEFORE_FLASH_RETRY) for status updates
2011-07-22 08:17:25: ... waited 6 s (of max. 120 s WAIT_PERIOD_BEFORE_FLASH_RETRY) for status updates
2011-07-22 08:17:31: ... waited 12 s (of max. 120 s WAIT_PERIOD_BEFORE_FLASH_RETRY) for status updates
2011-07-22 08:17:37: ... waited 18 s (of max. 120 s WAIT_PERIOD_BEFORE_FLASH_RETRY) for status updates
2011-07-22 08:17:43: ... waited 24 s (of max. 120 s WAIT_PERIOD_BEFORE_FLASH_RETRY) for status updates
2011-07-22 08:17:49: ... waited 30 s (of max. 120 s WAIT_PERIOD_BEFORE_FLASH_RETRY) for status updates
2011-07-22 08:17:55: status of request "request#SHrbUENTsJ#2" (flashPrograms) (urn:wisebed:node:ubern:2): 100
2011-07-22 08:17:55: status of request "request#SHrbUENTsJ#2" (flashPrograms) (urn:wisebed:node:ubern:4): 100
2011-07-22 08:17:55: status of request "request#SHrbUENTsJ#2" (flashPrograms) (urn:wisebed:node:ubern:5): 100
2011-07-22 08:17:55: status of request "request#SHrbUENTsJ#2" (flashPrograms) (urn:wisebed:node:ubern:6): 100
2011-07-22 08:17:55: status of request "request#SHrbUENTsJ#2" (flashPrograms) (urn:wisebed:node:ubern:7): 100
2011-07-22 08:17:55: status of request "request#SHrbUENTsJ#2" (flashPrograms) (urn:wisebed:node:ubern:13): 100
2011-07-22 08:17:55: ... waited 36 s (of max. 120 s WAIT_PERIOD_BEFORE_FLASH_RETRY) for status updates
2011-07-22 08:17:55: flashing image 19 to node urn:wisebed:node:ubern:2 succeeded!
2011-07-22 08:17:55: flashing image 20 to node urn:wisebed:node:ubern:4 succeeded!
2011-07-22 08:17:55: flashing image 20 to node urn:wisebed:node:ubern:5 succeeded!
2011-07-22 08:17:55: flashing image 20 to node urn:wisebed:node:ubern:6 succeeded!
2011-07-22 08:17:55: flashing image 18 to node urn:wisebed:node:ubern:7 succeeded!
2011-07-22 08:17:55: flashing image 20 to node urn:wisebed:node:ubern:13 succeeded!
2011-07-22 08:17:55: run#0: images deployed, experiment started ... retrieving output from sensornodes
```

Experiments currently running on the Testbed:

Experiment ID:	SHrbUENTsJ
User:	673023@gridlab21.unibe.ch (buergi@iam.unibe.ch)
Name:	test

Displayed Experiment:

Experiment ID:	SHrbUENTsJ
User:	673023@gridlab21.unibe.ch (buergi@iam.unibe.ch)
Name:	test
Description:	*no description supplied*
Start:	2011-07-22 08:15:00
End:	2011-07-22 08:44:59
Runs:	2

Node ID: urn:wisebed:node:ubern:2 — output

```
[201] cctrl-client 1: Sending packet
[203] cctrl-client 1: Received ACK; RTT = 0.085s
[201] cctrl-client 1: Sending packet
[203] cctrl-client 1: Received ACK; RTT = 0.085s
[201] cctrl-client 1: Sending packet
[203] cctrl-client 1: Received ACK; RTT = 0.085s
[201] cctrl-client 1: Sending packet
[203] cctrl-client 1: Received ACK; RTT = 0.085s
```

Node ID: urn:wisebed:node:ubern:4 — output

```
[205] cctrl-cache: Dropping ret packet - ACK:1591 > SEQ:1576
[205] cctrl-cache: Dropping ret packet - ACK:1621 > SEQ:1606
[205] cctrl-cache: Dropping ret packet - ACK:1741 > SEQ:1726
[205] cctrl-cache: Dropping ret packet - ACK:1741 > SEQ:1726
[204] cctrl-cache: ret SEQ:5281, ACK:1, PLEN:15, 500ms, e:0, c:1
[205] cctrl-cache: Dropping ret packet - ACK:5296 > SEQ:5281
[205] cctrl-cache: Dropping ret packet - ACK:5296 > SEQ:5281
[204] cctrl-cache: ret SEQ:5566, ACK:1, PLEN:15, 500ms, e:0, c:1
```

Node ID: urn:wisebed:node:ubern:5 — output

```
[204] cctrl-cache: ret SEQ:1576, ACK:1, PLEN:15, 500ms, e:0, c:1
[204] cctrl-cache: ret SEQ:1606, ACK:1, PLEN:15, 500ms, e:0, c:1
[204] cctrl-cache: ret SEQ:1726, ACK:1, PLEN:15, 500ms, e:0, c:1
[204] cctrl-cache: ret SEQ:2071, ACK:1, PLEN:15, 500ms, e:0, c:1
[205] cctrl-cache: Dropping ret packet - ACK:4756 > SEQ:4741
[204] cctrl-cache: ret SEQ:5281, ACK:1, PLEN:15, 500ms, e:0, c:1
[204] cctrl-cache: ret SEQ:5281, ACK:1, PLEN:15, 500ms, e:0, c:1
[204] cctrl-cache: ret SEQ:5566, ACK:1, PLEN:15, 500ms, e:0, c:1
```

Node ID: urn:wisebed:node:ubern:6 — output

```
[204] cctrl-cache: ret SEQ:2356, ACK:1, PLEN:15, 500ms, e:0, c:1
[204] cctrl-cache: ret SEQ:3526, ACK:1, PLEN:15, 500ms, e:0, c:1
[204] cctrl-cache: ret SEQ:3616, ACK:1, PLEN:15, 500ms, e:0, c:1
[204] cctrl-cache: ret SEQ:4441, ACK:1, PLEN:15, 500ms, e:0, c:1
[204] cctrl-cache: ret SEQ:4741, ACK:1, PLEN:15, 500ms, e:0, c:1
[204] cctrl-cache: ret SEQ:5281, ACK:1, PLEN:15, 500ms, e:0, c:1
[204] cctrl-cache: ret SEQ:5281, ACK:1, PLEN:15, 500ms, e:0, c:1
[204] cctrl-cache: ret SEQ:5566, ACK:1, PLEN:15, 500ms, e:0, c:1
```

Node ID: urn:wisebed:node:ubern:7 — output

```
[202] cctrl-server: 172.16.2.0:1025 - 'Packet 390 R:n'
[202] cctrl-server: 172.16.2.0:1025 - 'Packet 391 R:n'
[202] cctrl-server: 172.16.2.0:1025 - 'Packet 392 R:n'
[202] cctrl-server: 172.16.2.0:1025 - 'Packet 393 R:n'
[202] cctrl-server: 172.16.2.0:1025 - 'Packet 394 R:n'
[202] cctrl-server: 172.16.2.0:1025 - 'Packet 395 R:n'
[202] cctrl-server: 172.16.2.0:1025 - 'Packet 396 R:n'
[202] cctrl-server: 172.16.2.0:1025 - 'Packet 397 R:n'
```

Node ID: urn:wisebed:node:ubern:13 — output

```
[204] cctrl-cache: ret SEQ:1516, ACK:1, PLEN:15, 500ms, e:0, c:1
[204] cctrl-cache: ret SEQ:1531, ACK:1, PLEN:15, 500ms, e:0, c:1
[204] cctrl-cache: ret SEQ:1576, ACK:1, PLEN:15, 500ms, e:0, c:1
[204] cctrl-cache: ret SEQ:1606, ACK:1, PLEN:15, 500ms, e:0, c:1
[204] cctrl-cache: ret SEQ:1726, ACK:1, PLEN:15, 500ms, e:0, c:1
[204] cctrl-cache: ret SEQ:5281, ACK:1, PLEN:15, 500ms, e:0, c:1
[204] cctrl-cache: ret SEQ:5281, ACK:1, PLEN:15, 500ms, e:0, c:1
[204] cctrl-cache: ret SEQ:5566, ACK:1, PLEN:15, 500ms, e:0, c:1
```

Figure 3.10: Experiment monitoring in TARWIS

sor node, the user can assign one of the previously uploaded images. Once the experiment starts, these images will be flashed to the nodes. Additionally, commands can be scheduled, which will be sent to the nodes as serial interface input during the experiment run. A configuration can also be saved as a template and reused in subsequent experiments.

During the runtime of the experiment, TARWIS provides a user interface for live monitoring of the sensor nodes. A screenshot of this interface is shown in Fig 3.10. The area at the top left notifies about the ongoing actions of the TARWIS experiment-dependent processes, such as experiment initiation, sensor node programming, etc. Placed at the bottom, a map of the testbed illustrates the positions of the sensor nodes participating in the current experiment. Adjacent to the map, a window is provided for each sensor node, which displays the latest output captured from the node's serial interface. Below each of these windows the user can enter commands, which are written to the corresponding node's serial interface. This provides the ability of directly interacting with the sensor nodes, what proofed to be quite handy for debugging.

Once the experiment run has completed, the user can download the gathered results, parsed as an XML-based format, for further analysis and evaluation.

Chapter 4

Design and Implementation of Local Retransmission Mechanisms

Throughout the course of this thesis, we iteratively designed, tested, and combined a number of TCP performance optimizations. The starting point of our investigations is formed by the TCP segment caching strategy proposed in DTC [7] and TSS [8] (see Section 3.1) which permits to perform local retransmissions. This idea is the core component of our own design as well. On top of that, we added further extensions, which are independent from each other to a large extent, and which provide new functionalities or modify existing behavior. All of this is unified in our *cache and control* (cctrl) module, implemented into the Contiki OS network stack, with the possibility to enable or disable certain extensions at compile time.

The *cctrl* module is easily integrable into the µIP stack without major modifications and is transparent to the underlying network stack. This implies that it can be used in combination with any available MAC protocol, however, with the drawback of not being able to rely on MAC protocol specific features, such as overhearing of other ongoing transmissions on the radio channel. The only requirement of our approach is to have a sensor network with symmetric routes, where it is guaranteed that data (in particular TCP acknowledgments) traveling from node A to node B will take the same path as data traveling in the opposite direction.

The general design of the *cctrl* module, as well as caching and retransmission functionalities are described in Section 4.1. The additional extensions, which modify the behavior the retransmission mechanism, alter timeout metrics, and gather external information about ongoing traffic, are introduced in Section 4.2, 4.3 and 4.4, respectively. In Section 4.5 we propose an idea, which relies on having multiple connections to transport data.

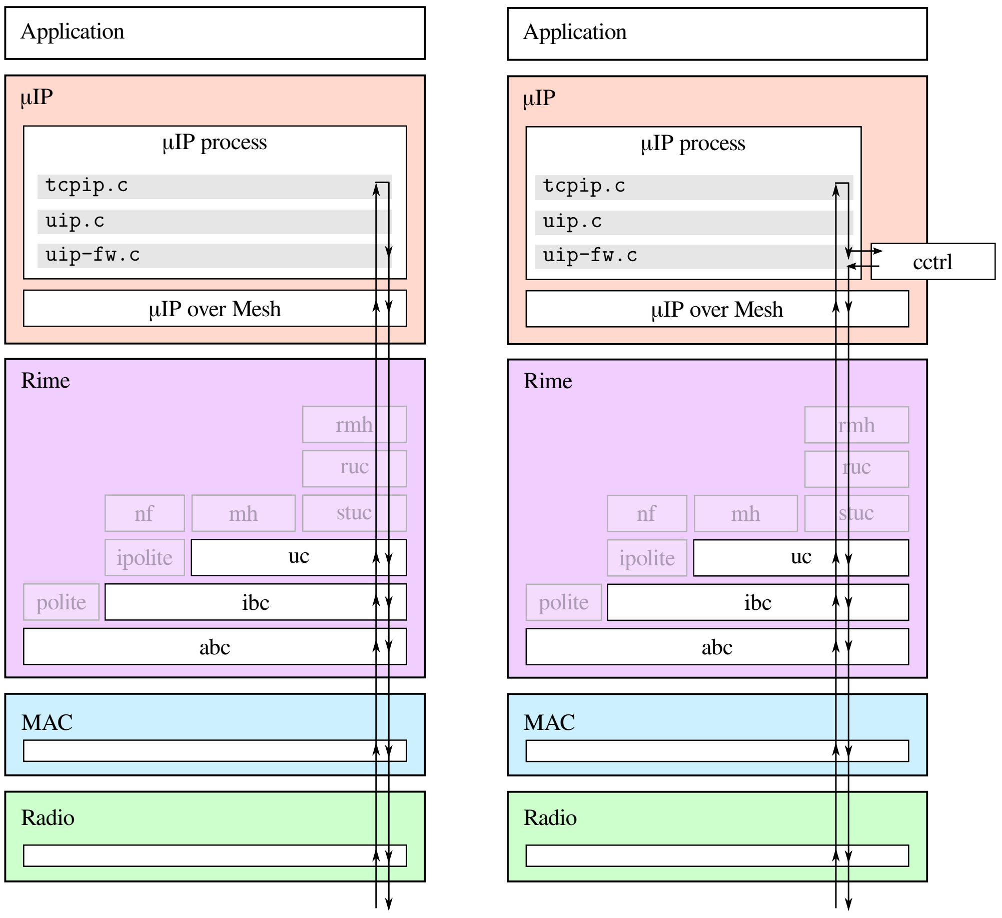
4.1 Segment Caching and Local Retransmissions

4.1.1 Implementation Issues

In order to be able to cache TCP segments, the *cctrl* module has to be aware of all TCP packets the node forwards. Figure 4.1(a) shows the flow of such a packet through Contiki's network stack. A TCP packet, encapsulated in a Rime packet, is received by the radio and passed to the MAC layer, which copies it to Rime's packet buffer. From here, it is processed by the various

Rime modules responsible for sending and receiving unicast traffic until the Contiki module *µIP over mesh* gets notified about the newly arrived packet. The Rime payload is then copied to µIP's packet buffer. When processing the data as a TCP packet, µIP recognizes that the local node is not the target recipient and therefore, instead of notifying the application, sends it back out again.

On its way out, the packet first passes *uip-fw*, whose purpose is to assign outgoing packets to the proper network interface, in case more than one exists. Afterwards it gets passed again to *µIP over mesh* and to all other layers it has traveled through at arrival but in reversed order.



(a) Unmodified network stack (b) Network stack with *cctrl* module

Figure 4.1: TCP packet flow in Contiki's network stack

Since we are only interested in TCP packets forwarded by this node, the easiest way to intercept the packet flow is just before it leaves the actual µIP stack. Caching outbound instead of inbound packets is advantageous because then they are already processed and validated. The actual raw packet data stays the same. Therefore, we modify *uip-fw* in such a way it first calls the *cctrl* module to process outgoing packet still residing in the µIP packet buffer before handing

```
typedef struct {
    char used;
    float rtt;                          /* Average RTT calculated using EWMA          */
    struct uip_tcpip_hdr tcpip_hdr;     /* Header copy of the latest TCP packet       */
    struct ctimer remove_timer;         /* Triggers garbage collection                */
} cctrl_conn_info;

typedef struct {
    char used;
    u16_t pkt_len;                      /* Size of cached packet (headers + payload) */
    u8_t *cctrl_cache_ptr;              /* Points to packet in cctrl_buffer          */
    struct uip_tcpip_hdr *tcpip_hdr;    /* Points to TCP/IP header in cctrl_buffer   */
    struct ctimer retransmission_timer; /* Triggers retransmission                   */
    clock_time_t time_stored;           /* Used to calculate RTT                     */
    cctrl_conn_info *conn_info;
    u8_t retransmission_counter;
} cctrl_cache_entry;

static cctrl_cache_entry cctrl_cache[CCTRL_CACHSIZE * 2];
static cctrl_conn_info cctrl_conn_cache[CCTRL_CACHSIZE * 2];
static u8_t cctrl_buffer[UIP_BUFSIZE * CCTRL_CACHSIZE * 2];
```

Listing 4.1: Cctrl data structures

over to *µIP over mesh*. The adapted packet flow is illustrated in Fig. 4.1(b).

To be able to cache TCP segments, the *cctrl* module needs to allocate memory in which the data is buffered. This buffer has to be large enough to hold at least the content of two TCP/IP packets per observable connection, since data can travel from a client to a server as well as in the opposite direction. In fact, $w_c + w_s$ packets need to be cached, where w_c and w_s represent the TCP window size of client and server, respectively. But because at least one host is running µIP, which, as seen in Section 3.2.2, only allows having one unacknowledged segment per connection in flight, we can safely assume the window sizes not to be greater than one. Therefore, the *cctrl* module allocates a buffer with the size of two times the size of µIP's packet buffer.

Additionally nodes record state information of observed connections and cached segments. For each cached segment a node logs the total packet size, the time of arrival and the number of retransmissions. For each unidirectional connection a node stores the IP addresses and TCP port numbers of sender and receiver needed for identification, the sequence and acknowledgement number of the last packet seen, as well as the average round trip time (RTT) to the receiver. For simplicity, we do not store each connection identifying attribute individually, but copy the entire TCP/IP header, as seen in Listing 4.1, which shows the actual data structures used. As we will show later, this header is also used to regenerate TCP ACKs for retransmissions. Hence, this simplification does not waste a lot of memory, since the only header fields that do not have to be stored are TCP and IP checksums (i.e. $2 * 16$ bits).

4.1.2 Packet Processing

Each IP packet being forwarded is processed by the *cctrl* module (cf. Fig. 4.1(b)). IP packets which do not have a TCP payload (e.g. UDP packets) are ignored and directly passed to the next layer. TCP packets are processed according to the observed state of the connection they belong to and the actual type of the packet. First, it is checked whether the current packet is an acknowledgement of a cached packet, i.e., whether the current packet's acknowledgement number is greater than the cached packet's sequence number. In this case, the packet is removed from the cache. Should in contrary the current packet be a retransmission of a segment for which an acknowledgment has already been received, the packet is dropped and not further processed. Instead, the node regenerates a corresponding TCP ACK, which is explained in more detail in the next section.

The next step is the actual caching: Only packets that contain payload data are cached. The content of the µIP packet buffer is copied to a free slot in the *cctrl* buffer, if available, and a retransmission timer is scheduled. When the retransmission timer expires before an acknowledgment returns, the packet is released from the cache and is sent out again.

Finally, the connection information is updated, which includes a backup of the processed packet's TCP/IP header. This step is omitted for out-of-sequence packets and retransmissions, to ensure that the connection information always holds the highest sequence and acknowledgment number ever seen.

4.1.3 Retransmitting Data and Acknowledgments

We distinguish between two different kinds of packet loss for which a retransmission has to be performed: either a data packet gets lost between sender and recipient, or an acknowledgment traveling in the opposite direction. The loss of a data packet is actually not detectable. This is why for each cached segment a timer is scheduled, which pro-actively initiates a retransmission should it take considerably more time than usual for an acknowledgment to arrive.

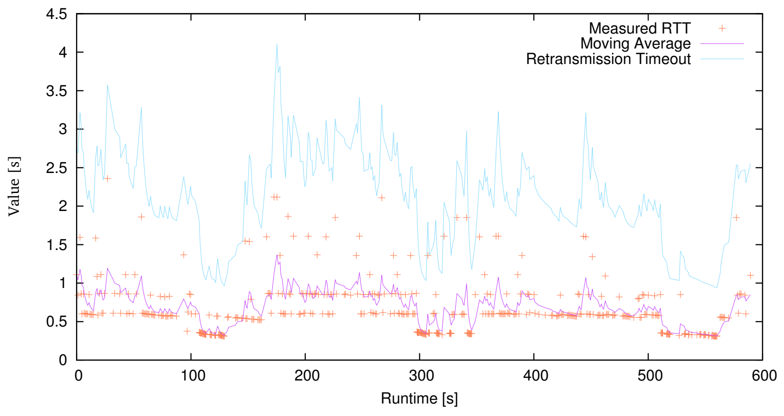
Initially, the retransmission timeout is set to 2 seconds, which is high enough to ensure the first cached segment is not retransmitted early. With the arrival of an acknowledgment for a cached but not yet retransmitted segment, the node can calculate the current round trip time (rtt_c) to the receiver of the cached segment. This is then used to estimate the average RTT of a particular connection using an exponentially weighted moving average (EWMA) function as shown in the following formula, which is used similarly in TSS [8]:

$$rtt = rtt_c * \alpha + rtt * (1 - \alpha) \tag{4.1}$$

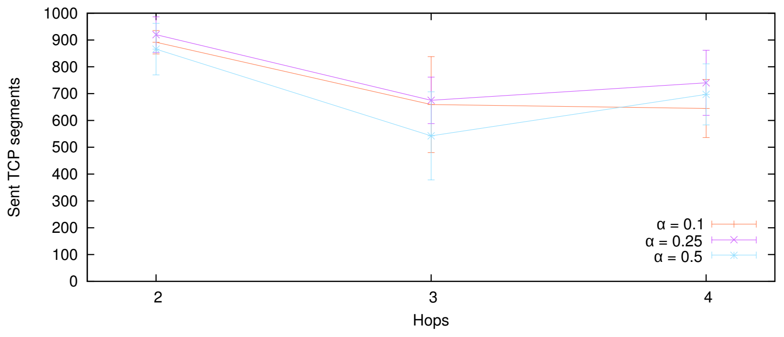
Using an α of 0.25, the actual retransmission timeout is then set to be

$$t_r = n * rtt \tag{4.2}$$

with $n = 3$. Figure 4.2(a) shows a visualization of the EWMA function applied to RTT measurements taken in a real world experiment, in which an X-MAC node sent TCP packets over two hops. The retransmission timeout (t_r), which results out of the average RTT estimate is illustrated as well. It is apparent that using an RTT multiplier of $n = 3$ to derive the retransmission timeout is sufficient to minimize the probability of triggering early (i.e. redundant) retransmissions. TSS, in contrary, uses a value of $n = 1.5$, which, in our case, would be two low and lead to collisions with returning acknowledgments. Further preliminary small scale experiments have supported the choice of these values for n and α as well. In contrary, other combinations



(a) Two-hop round trip time experienced by X-MAC node and inferred retransmission timeout.



(b) Influence on throughput of NullMAC nodes when using different RTT weights in Eq. 4.1.

Figure 4.2: Evaluation of retransmission timeout parameters

of the multiplier $n \in [1.5, 3]$ and $\alpha \in [0.1, 0.25, 0.5]$ could not perform as well. For example, as Fig. 4.2(b) illustrates, using different α values negatively influences throughput.

An acknowledgement is known to be lost when the node receives a retransmission of a data segment which has already been acknowledged by the receiver. As mentioned before, in this situation the node does not further forward the retransmission but instead regenerates an acknowledgement. This is done using the TCP/IP header stored as connection state information.

4.1.4 Maintenance

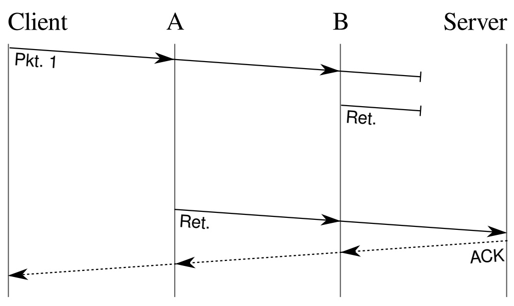
To prevent the *cctrl* cache to be occupied by connections that are no longer in use, a maintenance timer is running for each observed connection. The maintenance timer is initially set to expire after 60 seconds and is reset every time the node receives a packet from this particular connection. Should a maintenance time expire, the corresponding connection information entries are removed together with any remaining cached segments. The same procedure is executed when a TCP FIN or TCP RST is received, since these packets signal that the connection has either been terminated or timed out.

4.2 Multiple Retransmissions and Duplicate Segment Dropping

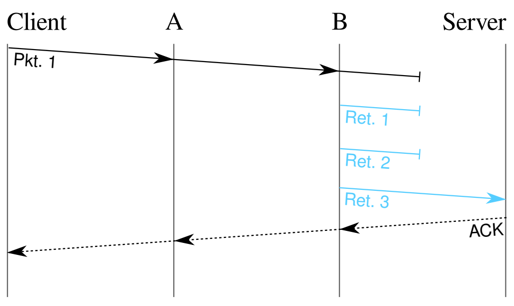
In our initial implementation, a segment is removed from the cache after the retransmission mechanism has started. This is disadvantageous in situations where the first transmission of a packet is lost and, shortly after, the subsequent retransmission as well, e.g., caused by a temporary decrease of channel quality due to timely correlated transmission attempts and collisions. This moves the burden of successful retransmission to nodes further away of the receiver and eventually increases the likelihood of an additional packet loss (cf. Fig. 4.3(a)). Therefore, a first, rather small change in the initial implementation is to allow the nodes to do more than one retransmission of the same segment. After retransmission, the segment remains cached and the retransmission timer gets rescheduled. We limited the total amount of subsequent retransmissions of a specific segment to 3 retries. If these 3 retries fail, the segment is dropped definitively. Shortly after the end-to-end retransmission from the original source is expected to arrive, which will re-initiate the retransmission mechanism for this segment. The resulting change in packet flow is show in Fig. 4.3(b).

Since this strategy causes all nodes to produce more retransmissions, not just the one in front of the lossy link, the overall traffic generated by the network is potentially increased by redundant retransmissions. Therefore, we added the following simple countermeasure (cf. Fig 4.3(c): When a node receives a retransmission of an already cached segment, the node does not forward the segment, but drops it, to reduce the load on the subsequent links.

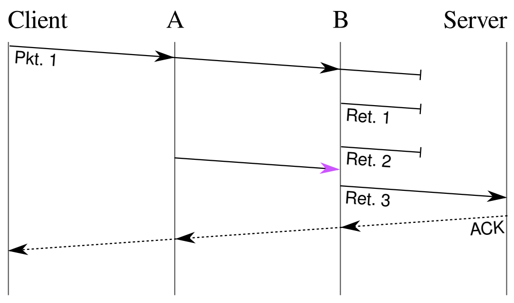
We also considered faking a TCP acknowledgment in such a situation, as illustrated in



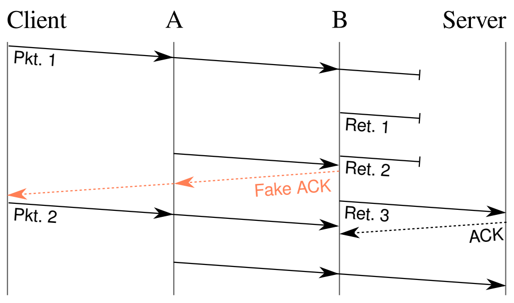
(a) Only one retransmission per node



(b) Up to three retries per node



(c) Dropping of redundant retransmissions



(d) TCP acknowledgment spoofing

Figure 4.3: Cctrl nodes' behavior on packet loss and retransmissions

Fig. 4.3(d). This would prevent the preceding nodes from retransmitting a packet which already has successfully been received by an intermediate node. However, the drawback of this approach is potentially inconsistent TCP traffic, as it violates TCP's end-to-end semantics: A spoofed ACK allows the TCP client to send the next data packet. When this packet arrives at the node which is still waiting for an acknowledgement for the previous segment, the new packet must not be forwarded any further. Because then, should the packet overtake its predecessor and arrive out of sequence, this could violate the receivers propagated receive window. Furthermore, should the retransmitting node exhaust its retransmission limit and ultimately drop the segment, there would be no possibility for recovery, as this segment has - misleadingly - already been acknowledged. For this reason, we rejected to let nodes create fake acknowledgments.

4.3 Hop-distance Dependent Retransmissions

The approach designed and illustrated in this section is based on an observation made in small scale experiments: We experienced that, occasionally, multiple nodes, within a chain of nodes forming the TCP connection, start to retransmit the same segment at roughly the same time. This behavior should be prevented by scheduling retransmissions based on the round trip times between node and receiver. However, in wireless networks round trip times tend to be very inconsistent. This is why we introduce the hop-distance to the receiver as an additional, more reliable parameter for calculating the retransmission timeout.

The hop-distance to a host is implicitly propagated by every TCP packet. The TCP header contains a time to live (TTL) field, whose value is decreased by one every time the packet is forwarded. Hence, subtracting the current TTL from its initial value results in the hop-distance to the original sender of this packet. The initial TTL value used by Contiki is 64, which is also the value recommended by IANA [32].

To use this information to calculate the retransmission timeout, we replace the static $n = 3$ in equation 4.2 by a hop-distance dependent multiplier, where h represents the number of hops to the receiver:

$$t_r = \min\{h, h_{max}\} * \beta * rtt \tag{4.3}$$

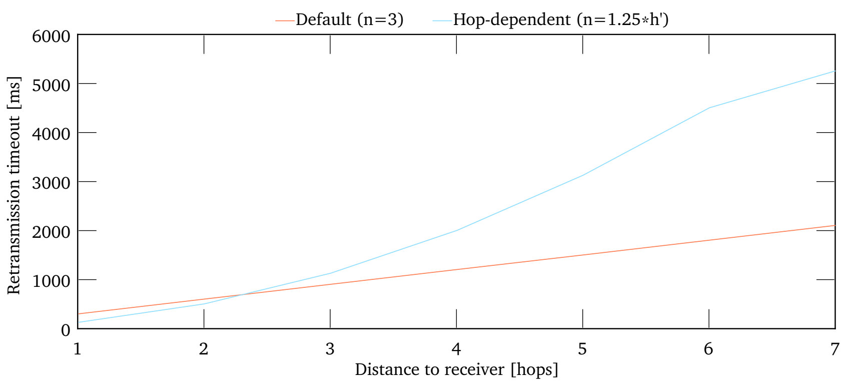


Figure 4.4: Retransmission timeout comparison for example topology with constant per link round trip times of 100 ms

Since the longest route with which we experimented consisted of 7 nodes, we let h_{max} be 6. For β we tested different values in the range between 1 and 2, using $\frac{1}{4}$ steps and $\beta = 1.25$ showed the most satisfying results.

As Fig. 4.4 illustrates, compared to the default retransmission scheme, nodes closer to the recipient are now allowed to retransmit earlier whereas node further away have to wait considerably longer. We expected this to result in a reduced chance of multiple simultaneous retransmissions without major impact on the overall performance.

4.4 Activity Monitoring

Some wireless channel protocols take advantage of the broadcast nature of omnidirectional wireless transmission to gain additional information about ongoing transmissions in the vicinity. DTC and TSS use overhearing of forwarded packets as implicit acknowledgment for a successful reception. Since overhearing is to a large extent radio-type or MAC-layer specific, relying on the general availability of this property would limit the application of our *cctrl* module. For instance, when used in combination with an energy efficient MAC protocol, packets can only be overheard when the radio is currently in a wake-up interval, which is not the case for the majority of time. But also the scarce reception of packets destined for other nodes can bare valuable information. For example, a high amount of overheard packets could indicate a situation in which it would be beneficial for a node to withhold a scheduled transmission to avoid further collisions, since a lot of other nodes are currently trying to send.

4.4.1 MAC Proxy

Our implementation of the *cctrl* module in Contiki hence needs a hook to the used MAC protocol, because this is where overheard packets get rejected. To have access to these packets we need access to the packet buffer of the MAC layer. Since directly manipulating the functionality of MAC protocols would be against the design goal of keeping the MAC protocol unmodified and replaceable, we implemented a MAC protocol proxy. This proxy implements the interface Contiki expects a MAC protocol to have but does not provide any functionality by itself. Instead, it initializes a real MAC protocol and simply forwards every function call. This slightly alters the packet reception workflow, as illustrated in Fig. 4.5. In Contiki, the MAC protocol is notified by the radio driver after a packet reception. The MAC layer proceeds to inform the Rime layer about this event. Rime then calls the MAC protocol's `read` function to start the actual parsing. The detour via Rime is needed to make sure that there is no unprocessed packet left in Rime's packet buffer before the MAC protocol copies the newly received data.

In our activity monitoring approach integrated with our *cctrl* module, Rime calls the `read` function of the newly introduced MAC proxy instead. The proxy forwards this call to the actual MAC protocol, which parses the packet, copies the received data to the Rime buffer and returns the amount of bytes copied. Should there be any error (e.g. a bad CRC checksum) or should the packet have a wrong target address, the return value is zero. However, since also in this case the received Rime packet still resides in the packet buffer, which is accessible by the MAC proxy, the proxy can detect overheard packets and access their content without modification of

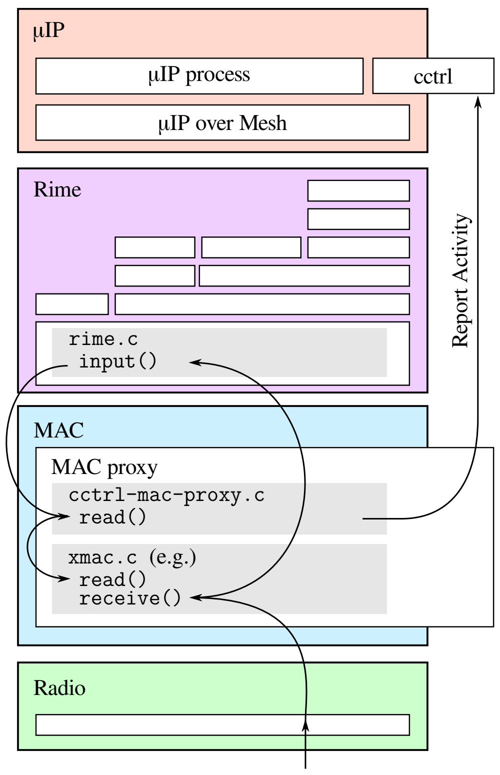


Figure 4.5: Workflow for processing overheard packet using MAC proxy

the underling MAC protocol. In our case, the proxy checks the packet's target address. Should this address not be the node's Rime address (i.e. the packet has been overheard) the *cctrl* module is notified about the channel activity.

4.4.2 Activity Dependent Early Retransmissions

The *cctrl* module stores timestamps of the 20 most recent activity notifications received from the MAC proxy. This history is used to calculate the current activity level, i.e. the amount of overheard packets over a short time window. We chose the time window over which the amount of packets is calculated to always equal the average round trip time of the corresponding connection. The benefit of such an adaptive window size is that it compensates for throughput and packet-flow differences between MAC protocols. A static window size would have to be explicitly configured for each MAC protocol, since in a fixed amount of time the measured activity level would be considerably higher when a high throughput MAC protocol (e.g. NullMAC) is used than it would be in combination with a low throughput protocol (e.g. LPP).

To test whether this approach to generate activity levels has any informative value in combination with duty-cycling MAC protocols, we set up a test experiment using 7 nodes communicating over X-MAC. The experiment follows the same configuration as described in Section 5.1: Five *cctrl* enabled nodes forward TCP traffic between the two other nodes. Each node logged

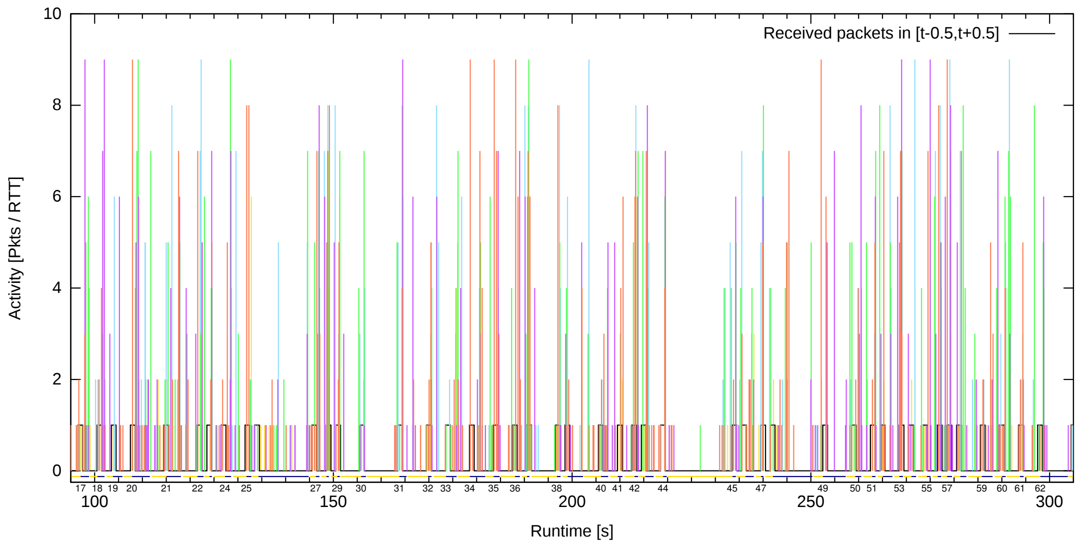


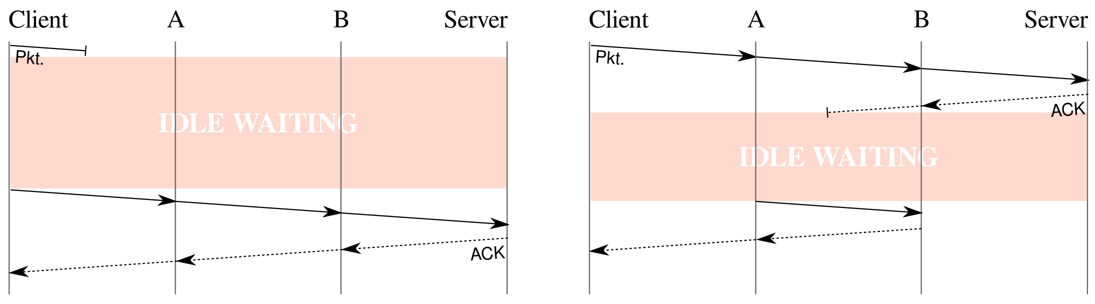
Figure 4.6: Activity levels registered by X-MAC nodes. The dashed line indicates the traveling time for data segments (*yellow*) and acknowledgments (*blue*).

periodically, as well as prior to every local retransmission, their current activity level, i.e., the number of packets overheard over one RTT. These levels are illustrated in Fig. 4.6, where each node is represented by one specific color. Figure 4.6 further depicts the reception of data packets at the end node and the corresponding sequence number, indicated by black bumps and integers along the x-axis, respectively. Additionally, the dashed line shows the traveling time of individual data packets and acknowledgments.

As expected, during normal packet flow all sensor nodes register relatively high activity. However, some packets (e.g. 30, 37, 44, etc.) need a significantly longer time period to be delivered. Furthermore, little or no activity is registered when this happens, indicating that no retransmissions are performed and that there is no conceivable source of packet collision.

Further analysis revealed that there are two common problems which caused this type of delay. Both are visualized in Fig. 4.7. First is the loss of a data packet at the first hop, during the transmission of the original source. Since this host does not use the *cctrl* module, the packet has not yet been cached and cannot be retransmitted early. Hence, all nodes are waiting for the sending host's TCP timeout to occur.

A second problem is the loss of a TCP acknowledgment close to its final destination. The retransmission of the ACK can only be triggered by the reception of a retransmitted data packet. The more nodes the ACK has already passed, the higher is the idle waiting time, as then there are fewer nodes left capable of initiation a retransmission, which, in addition, have larger retransmission timeouts.



(a) Loss of data packet on first (unprotected) hop (b) Loss of ACK packet close to its final destination

Figure 4.7: Two common causes of idle waiting

```
when TCP packet is forwarded do
  schedule retransmission in 2/3 * retransmission timeout

when retransmission timer expires do
  if is first retransmission attempt then
    if activity = 0 then
      retransmit segment
    else
      reschedule retransmission in 1/3 * retransmission timeout
  else
    retransmit segment
```

Listing 4.2: Activity dependent early retansmissions

Implementation

To address this second issue we extended the core implementation of the *cctrl* module to be able to retransmit cached segments earlier if the activity level is lower than a certain threshold. As the pseudo code in Listing 4.2 depicts, when using activity dependent retransmissions, the retransmission timeout is reduced to $\frac{2}{3}$ of the usual value. When the retransmission timer expires, the node only sends the cached packet if the current activity level is zero. Otherwise the retransmission is delayed by the remaining $\frac{1}{3}$ of the usual timeout duration.

4.5 Multiple Connections

We continued our investigations to improve TCP flow performance, while maintaining our main design decisions: The changes should remain independent of any MAC layer participating and should not put in question any basic design issues of the established µIP stack.

One way of further improving the performance of TCP would be to modify Contiki's µIP stack to handle the transmission of multiple unacknowledged segments and to poll the application more frequently for new data. But this would require major changes to µIP's current implementation with the risk of creating unforeseen side-effects and is, therefore, contradictory to our design guideline.

A similar, more versatile approach is to simultaneously establish multiple TCP connections between client and server to transport data. This allows an application to send out a new data packet over a different connection even when a previously transmitted packet has not yet been acknowledged. This also addresses the issues discovered in the previous section: When one connection is idle because all nodes have to wait for a retransmission to be triggered, another connection could still be operating.

The actual implications of this approach were unforeseeable: instead of improving throughput, it could also have led to a network collapse due to the heavier load and more collisions. Therefore, we decided to implement this as a simple proof of concept prototype. We modified the used client application to open multiple TCP connections to the same server instead of only a single one. Whenever µIP polls one of these connections for new data, the application transmits the next outstanding chunk. Hereby, we ignore the fact that data fragments might arrive at the server side in wrong order. However, we present a design idea for this concept which operates beneath the application layer and does not violate TCP specifications in the following section.

4.5.1 Split and Merge

For this concept, the *cctrl* module not only monitors outgoing TCP traffic, but incoming traffic as well. Each node which has a responsibility for a TCP end target, i.e. a node which either runs a TCP client or server application, or which acts as a gateway node to a foreign network, runs a lightweight TCP server. When such a node monitors a TCP SYN coming from its observation target, it redirects this traffic to the locally running TCP server and tries to establish two connections with the server addressed in the original SYN packet. The *cctrl* module of this server will intercept these connection requests and open a single connection to the locally running target application's TCP server. Once the SYN handshake for this last connection has completed successfully, the other connections are acknowledged in reversed order. Now whenever the TCP client sends data, it will be tunneled over either of the two connections within the WSN. An example with a TCP application communicating from a WSN to a foreign host is illustrated in Fig. 4.8.

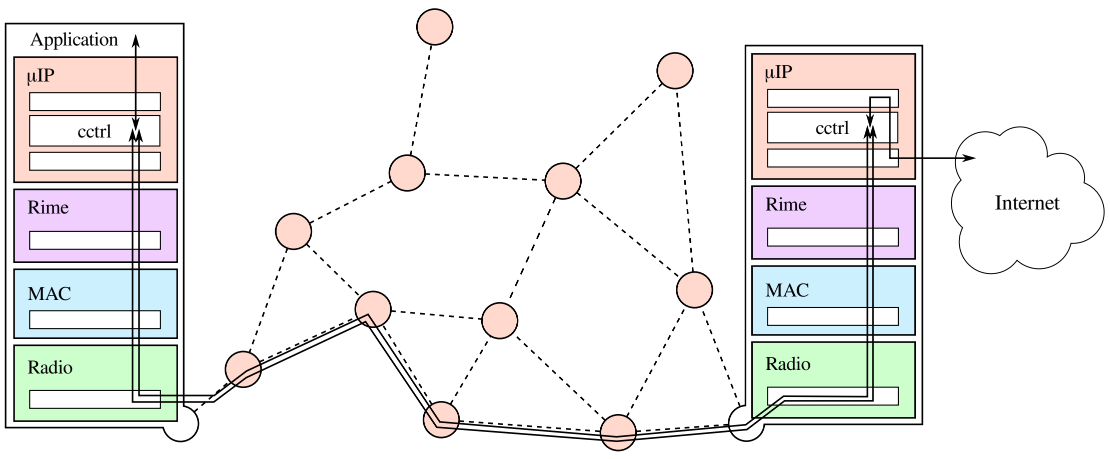


Figure 4.8: Multiple TCP connections between application node and Internet gateway

In order to correctly merge the incoming data split over two connections, the *cctrl* node at the receiver side has to maintain a small buffer to be able to hold back and rearrange segments which may arrive in wrong order. This buffer has to provide space for just a few packets. Should there not be enough memory left for some reason, a receive window of zero can be propagated on one connection to temporary stop its data flow.

4.6 Experiment-related Implementations

Since we wanted to experimentally evaluate the performance of our *cctrl* module, some further implementations and modifications were needed. To generate, transmit and receive TCP traffic within the sensor network, some sort of TCP applications had to be built, which are described in Section 4.6.1. Furthermore, as discussed in Section 4.6.2, the network's topology had to consist of static routes.

4.6.1 TCP Client and Server

To be able to send and receive TCP data, a Contiki TCP client and server application is needed. The implementation of the server software is straightforward. Since we do not need to process the received data at server side only few instructions are needed. Listing 4.3 shows a minimal implementation of such a TCP server, which just opens a socket for incoming connections, and turns the green LED on and off to indicate the successful establishment and closing of a TCP connection, respectively. Our final implementation also produced some log messages at certain TCP events, which were used for statistical analysis.

The TCP client application follows a similar structure. Instead of passively listening to incoming connections, it uses the µIP's `tcp_connect` command to establish one or, when the *multiple connections* extension is used, many TCP connections. Furthermore, it has to react accordingly to raised TCP events, as depicted in Listing 4.4, which shows the client application's main control loop. Whenever µIP polls the application for new data, or requests a retransmission, we use the `cctrl_client_send_packet` function to send out a predefined, fixed size character string, which, amongst other things, contains the current packet number. Should a connection abort or expire, it is immediately re-established and the data transmission continues with as little delay as possible.

4.6.2 Static Routing

As previously mentioned, the only requirement of the *cctrl* module to the network topology is to have symmetric routes, which guarantee that packets traveling from node A to node B will pass the same nodes but in reversed order as packets traveling in the opposite direction. When using Rime's own routing mechanism, however, it is not certain whether a TCP acknowledgment will take the same path as the previous data packet. Rime uses a route discovery protocol and just sends a packet to the first neighboring node, which claims to know a route to the destination. In the worst case, the path between client and server could change with every transmitted packet.

Therefore, we modified this routing scheme to support manually configured, static routes, which are valid for all outgoing Rime packets.

The entries in Rime's routing table now contain an additional flag, to distinguish static from dynamic routes. Originally, entries are removed from the routing table after a certain amount of time, if they do not get refreshed by a recent route discovery. Static routes are ignored during this cleanup. Additionally, static routes can be saved to and loaded from the sensor node's flash memory.

```
#include "contiki.h"
#include "net/uip.h"
#include "dev/leds.h"
#include "../cctrl-route.h"
/*---------------------------------------------------------------------------*/
PROCESS(cctrl_server_process, "CCTRL Server Process");
AUTOSTART_PROCESSES(&cctrl_server_process);
/*---------------------------------------------------------------------------*/
PROCESS_THREAD(cctrl_server_process, ev, data)
{
  PROCESS_BEGIN();

  cctrl_shell_init();               /* Initiate serial shell   */
  cctrl_route_load();               /* Load static routes      */
  tcp_listen(htons(CCTRL_PORT));    /* Open ingoing TCP socket */

  while(1) {
      PROCESS_WAIT_EVENT_UNTIL(ev == tcpip_event);

      if(uip_connected()) {
         leds_green(1);
      }
      if(uip_newdata()) {
         leds_blink();
      }
      if(uip_closed() || uip_aborted() || uip_timedout()) {
        leds_green(0);
      }
  }

  PROCESS_END();
}
```

Listing 4.3: Example of a TCP server

```
while(1) {
  PROCESS_WAIT_EVENT_UNTIL(ev == tcpip_event);

  app = (appstate *)uip_conn->appstate.state;

   if(uip_connected()) {  /* Connection has been established */
      app->connected = 1;
      app->successive_fails = 0;
   }
   if(uip_rexmit()){      /* uIP requests retransmission */
      app->retransmissions++;
      cctrl_client_send_packet(1);
   }
   if(uip_acked()) {     /* Data has been acknowledged */
      app->rtt = (float)(clock_time() - app->last_packet) / (float)CLOCK_SECOND;
   }
   if(uip_poll()){       */ uIP requests new data */
      app->packet_count++;
      cctrl_client_send_packet(0);
   }
   if(uip_closed() || uip_aborted() || uip_timedout()) {
      app->connected = 0;
      if(!uip_closed()) {
         /* Reconnect */
         if (++(app->successive_fails) < MAX_FAILS) {
            conn = tcp_connect(&addr, HTONS(port), (void *)app);
            if(conn == NULL) {
               uip_abort();
            }
         }
      }
   }
}
```

Listing 4.4: TCP client's main loop

Chapter 5

Evaluation

For the evaluation of our *cctrl* module and its various extensions, we run experiments using TelosB sensor nodes deployed in an indoor WSN testbed, as described in the Sections 5.1 and 5.2. We tested the behavior of unmodified Contiki sensor nodes, and sensor nodes running Contiki with *cctrl* module with and without extensions in two different scenarios: First, data was transmitted over two, three, four, five and six hops on one single path. Second, two independent paths with one common node were established, each of which consisting of two to six nodes. The results of these two scenarios are discussed in Section 5.3 and 5.4, respectively.

5.1 Experiment Setup

All experiments were done using NullMAC in combination with the CSMA layer, and X-MAC, both running on Contiki 2.4. With the release of ContikiMAC included in Contiki 2.5's first release candidate, we ported our implementation to the new Contiki version as well. From this moment on we also run the experiments using ContkiMAC and the receiver initiated Low Power Probing protocol (LPP).

Each experiment run lasted 10 minutes during which the TCP clients tried to send as many packets as possible. To keep the size of the individual packets constant, which is a requirement to maintain a constant transmission error probability, all TCP data packets contained a 16 byte character string as payload. Including the various headers attached by the underlying network layers, the radio had to transmit 79 bytes per data packet. The structure of such a packet is illustrated in Table 5.1. A TCP acknowledgment, as transmitted by the servers in response to a data packet, is only 63 bytes in total, since it does not contain any TCP payload.

Besides the *cctrl* experiments, we also evaluated the performance of the default Contiki in combination with the four MAC protocols. For these control experiments, the sensor nodes' network stack was unmodified and did not contain the *cctrl* module. Therefore, the corresponding results are denoted as *unmodified* in the subsequent figures.

To reduce the chance of environmental influence, the experiments were run over night or during weekends. Since the frequency band the nodes use for communication is license-free and many consumer electronics work in the same or a near frequency range, we experienced that more stable results are gained by experiments running at non-working hours. Additionally, each

Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

0	Checksum		Timestamp		Radio
32	Authority Level	Footer			
48	Channel Number (*abc*)		Sender Address (*ibc*)		Rime
80	Receiver Address (*uc*)		Originator Address (*mh*)		
112	Destination Address (*mh*)		Previous Hop Address (*mh*)		
144	Hop Count (*mh*)				
152	Version & Hdr. Length	Type of Service	Lenght		IPv4
184	Identification		Offset		
216	TTL	Protocol	Checksum		
248	Source Address				
280	Destination Address				
312	Source Port		Destination Port		TCP
344	Sequence Number				
376	Acknowledgment Number				
408	Offset	Flags	Window Size		
440	Checksum		Urgent Pointer		
472	Options				
504	Payload Data				Application
536					
568					
600					

Table 5.1: Structure of a data packet

run was repeated 15 times to increase statistical significance. Hence, in the following sections, the figures always show the mean values of 15 runs and the standard deviation. In total, the data presented in this thesis was acquired in 2500 individual experiment runs, which is equivalent to a runtime of 425 hours.

5.2 Testbed Setup

We used up to 13 wireless sensor nodes, which are part of the local WISEBED [31] testbed deployed at our institute. The nodes are configured and monitored via the TARWIS [30] web interface, which is introduced in Section 3.4. As depicted in Table 5.2, depending on the scenario, each of the nodes used has one specific role: A node can either be a client (C), a server (S), or a forwarder (F). The client and server software images are Contiki operating systems running a special purpose TCP client application and TCP server application, respectively, which are introduced in Section 4.6.1. Besides the added support for static routes (cf. Section 4.6.2), the network stack of these two images is unmodified, i.e. without *cctrl* module. The software images of the forwarders do not run any application, just the operating system. The operating system is either an unmodified Contiki (for the control experiments) or has the *cctrl* module integrated.

The experiment configuration that requires the most wireless sensor nodes consists of 13 nodes distributed over two paths, each of which is six hops long. This configuration is illustrated in Fig. 5.1. The paths span over three floors with both TCP clients (Node 11, Node 13) residing at the first floor, and the corresponding TCP servers (Node 1, Node 2) placed in the same room

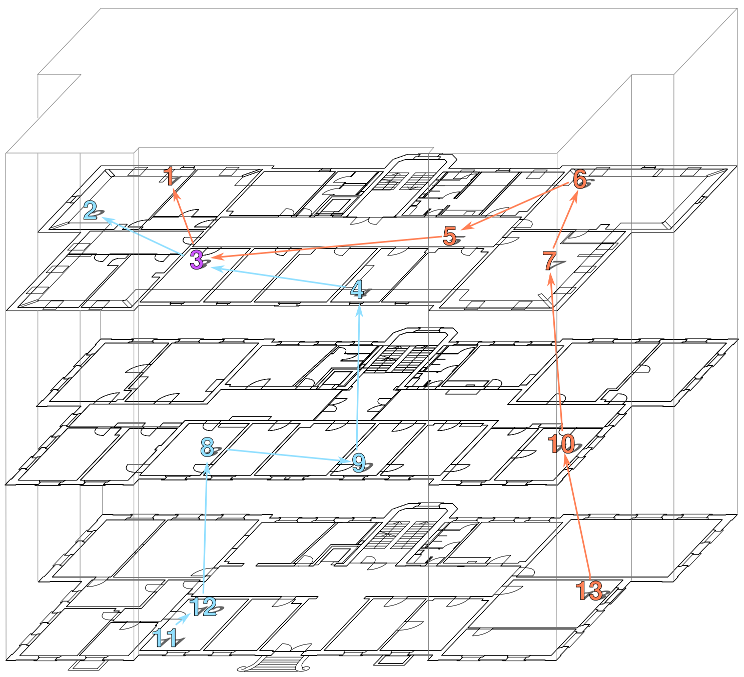


Figure 5.1: Sensor Node Topology

Hops	N_1	N_2	N_3	N_4	N_5	N_6	N_7	N_8	N_9	N_{10}	N_{11}	N_{12}	N_{13}
One route scenario													
2	S		F		C								
3	S		F		F	C							
4	S		F		F	F	C						
5	S		F		F	F	F			C			
6	S		F		F	F	F			F			C
Two route scenario													
2	S_1	S_2	F	C_2	C_1								
3	S_1	S_2	F	F	F	C_1			C_2				
4	S_1	S_2	F	F	F	F	C_1	C_2	F				
5	S_1	S_2	F	F	F	F	F	F	F	C_1		C_2	
6	S_1	S_2	F	F	F	F	F	F	F	F	C_2	F	C_1

Table 5.2: Sensor node roles for the different scenarios and path lengths

at the top floor. Table 5.2 shows the other configurations possible, i.e., with fewer nodes per route or only one route. Basically, the shortening of a route by one hop moves the client node one place closer towards the sender.

5.3 Single Route Scenario

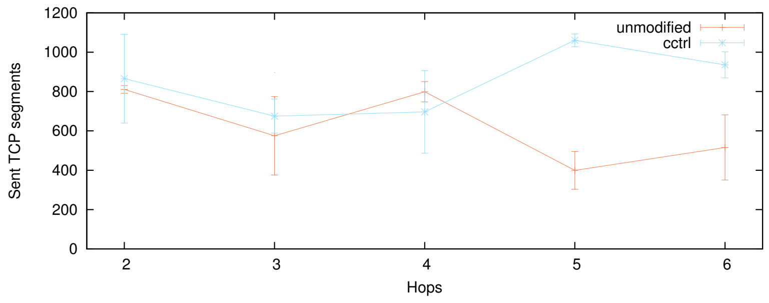
In the single route scenario, we evaluated how the *cctrl* module performs in transmitting data over 2, 3, 4, 5 and 6 hops, in its initial build without extensions, as well as when used in combination with the extensions introduced in Section 4. The goal of this scenario was to find the configuration most capable of increasing throughput, regardless of the MAC protocol used. Furthermore, we were interested in how this would affect the energy consumption when using radio duty cycling MAC protocols.

5.3.1 Segment Caching and Local Retransmissions

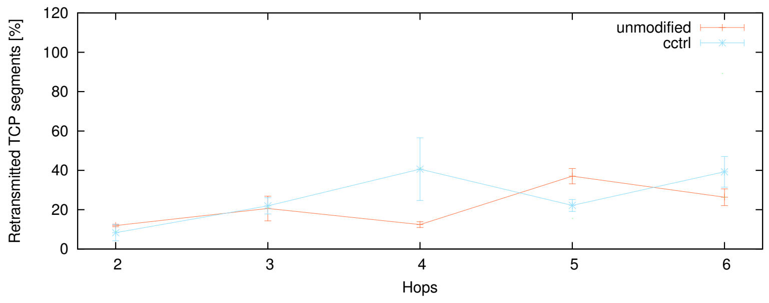
First, we evaluated the behavior of the initial *cctrl* module without any extensions compared to an unmodified network stack. As described in Section 4.1, at this stage, a *cctrl* node is only capable of performing just one single local retransmission.

The total amount of packets the TCP client could transmit on average during the 600 seconds experiments is illustrated in Fig. 5.2(a). As seen, NullMAC seems to benefit from the caching and retransmission mechanism, most notably when data travels long routes consisting

(a) Throughput



(b) Retransmissions



(c) Round trip time

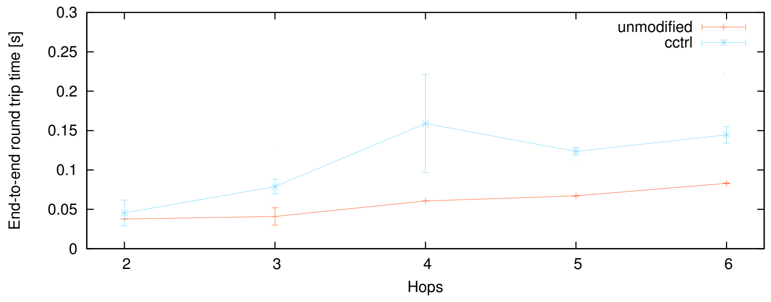


Figure 5.2: NullMAC with *cctrl* module

of five and more hops. For the shorter routes, the result is less distinctive, but performing local retransmissions still tends to be an improvement.

Figure 5.2(b) depicts the amount of retransmissions occurred per successfully sent packet. It should be noted, that in the *cctrl* case, this includes both end-to-end retransmissions as well as local retransmission. For the two and three hop experiments, there is no real difference between the *cctrl* case and the unmodified case, and for the five hop run, the *cctrl* module produces an even lower amount of retransmissions. Since there is no longer only one node performing retransmissions when using *cctrl* nodes, but several intermediate nodes, which can all start local retransmissions, it is interesting to see that the retransmission rate does not increase severely. Assuming only one retransmission is needed to recover a lost packet, this means that only few redundant retransmissions occurred.

For the sake of completeness, we also looked at the change in latency experienced by the TCP client. Figure 5.2(c) illustrates the end-to-end round trip time, which is acquired by measuring the time it takes for a TCP acknowledgment to return to the TCP client after a data packet has been sent out. In both experiments, with *cctrl* nodes and without, the RTT increases approximately linearly with the hop distance. However, the *cctrl* module introduces a slightly faster increase of the end-to-end RTT. This is of course, because the local retransmissions performed by intermediate nodes happen transparently to the TCP client. Consequently, the TCP client's own retransmission timeout is increased as well. This leads to longer waiting times in case a packet loss cannot be recovered by the *cctrl* nodes, and has to be done by the original sender. But this drawback should not be of great concern, since using the *cctrl* module should ultimately lead to faster recovering times, and reduce the need for end-to-end retransmissions.

The results obtained with X-MAC on the MAC layer are rather ambiguous. As with NullMAC, the throughput on five and six hop paths benefits from the *cctrl*'s functionality (cf. Fig. 5.3(a)). In contrary, on paths with less hops between client and server, the nodes performed better without the *cctrl* module.

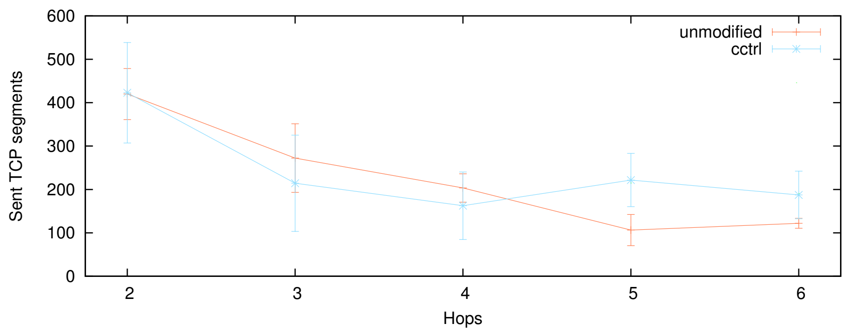
The same trend is observed when looking at the energy consumption in Figure 5.3(b). Because it would not be practically feasible to measure the actual energy consumption of every node in the testbed, Fig. 5.3(b) shows the average radio on time per node and packet, as a substitute, i.e. the sum of all nodes' radio on time divided by the amount of nodes and successfully transmitted TCP packets ($\frac{\sum t^i_{\text{on}}}{|N|*|P|}$). The radio on time t_{on} is the combined duration the radio spends in receive and transmit mode, and is constantly collected by Contiki functions. As discussed in Section 3.3, the radio usually is the most power consuming component on a sensor node. This makes the radio on time ideal to reflect changes in energy consumption.

It is noticeable, that the developing of the radio on time is almost reciprocally proportional to the developing of the throughput in Fig. 5.3(a). Although not achieved with the bare *cctrl* module, this is a first indication that the goal of leveraging throughput does not necessarily result in increased energy consumption, but in greater efficiency.

5.3.2 Multiple Retransmissions and Duplicate Segment Dropping

The first extension for our *cctrl* module, introduced in Section 4.2, adds the capability for a node to produce up to three retransmissions of a cached segment. This gives the forwarding

(a) Throughput



(b) Energy consumption

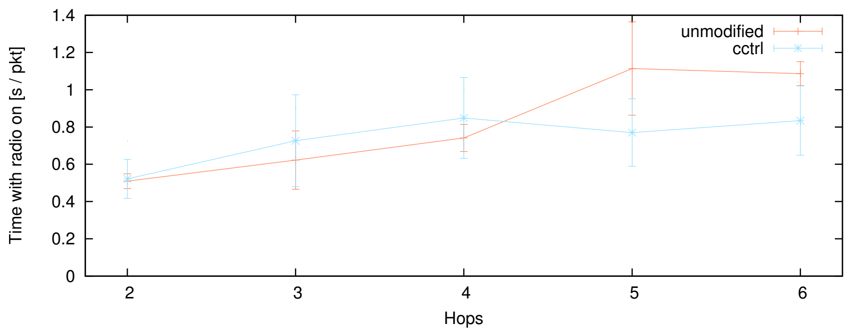


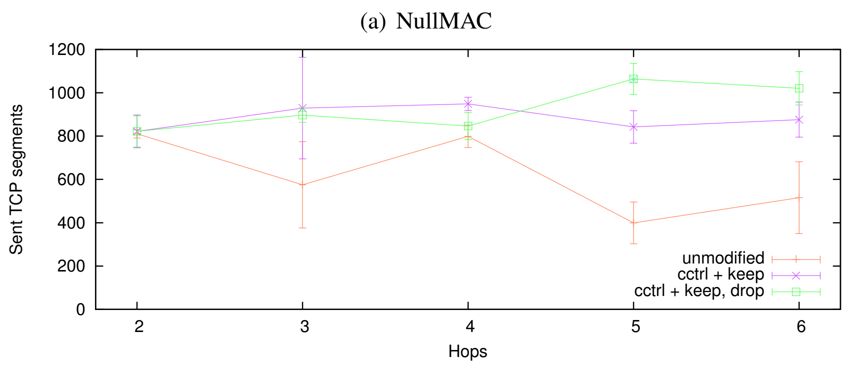
Figure 5.3: X-MAC with *cctrl* module

node the ability to cope with multiple, subsequent packet losses. Additionally, this extension also allows nodes to drop received retransmissions of TCP segments, which they have already cached. However, we tested these two features independently, i.e., we tested having multiple retransmissions with and without the segment dropping feature.

In the following figures, the functionality of keeping the TCP segments cached for multiple retransmissions is denoted as *keep*, whereas segment dropping is denoted as *drop*. Figure 5.4 depicts how these newly introduced changes affect throughput in comparison to having sensor nodes without *cctrl*. For NullMAC (a), the usage of multiple retransmissions clearly increases the total amount of successfully transmitted TCP packets for all path lengths. The dropping of received retransmissions, has only little influence, especially for the shorter routes. The five and six hop routes, however, slightly profit from the resulting reduced radio traffic.

X-MAC (b) also profits from this first extension. However, the difference of using the dropping mechanism has a greater impact on the throughput performance. When having multiple retransmissions in combination with segment dropping, the already poor *cctrl* performance of the two to four hop experiment runs (also cf. Fig. 5.3(a)) is even decreased further. Using the extension without segment dropping, on the other hand, lets the throughput match, more or less, to the one of unmodified nodes.

The same observation can be made when comparing the energy consumption in Fig. 5.5. For the 2, 3 and 4 hop routes, when the *cctrl* module does not drop redundant retransmissions, the average radio on time of a node is almost as low as when no *cctrl* module is used. For the 5 and



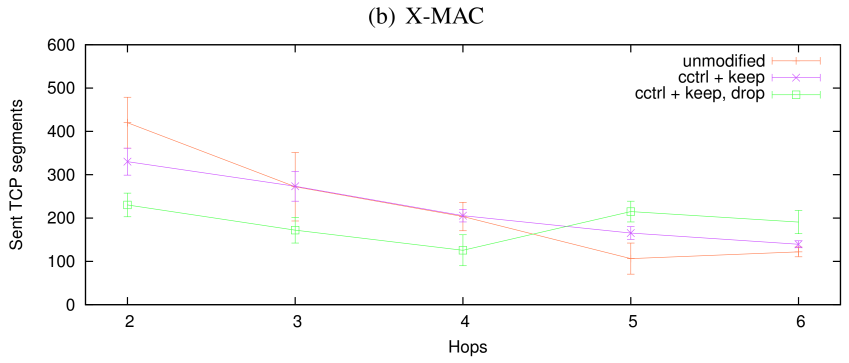


Figure 5.4: Throughput with multiple retransmissions

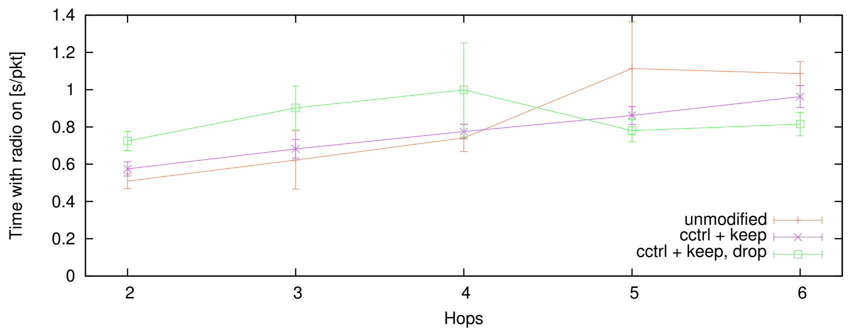


Figure 5.5: X-MAC's energy consumption with multiple retransmissions

6 hop routes, the energy consumption is slightly higher without segment dropping, but is still not as high as without any *cctrl* support.

For these reasons, i.e. good increase in throughput over all path lengths with NullMAC, and with X-MAC remarkably better performance for short paths than when using segment dropping, we decided to disable the segment dropping functionality for the multiple retransmission extension. At the same time, we chose to no longer use the *cctrl* module without this extension, as the experienced performance of both throughput and energy consumption generally increased, or, at least, remained at an equal level.

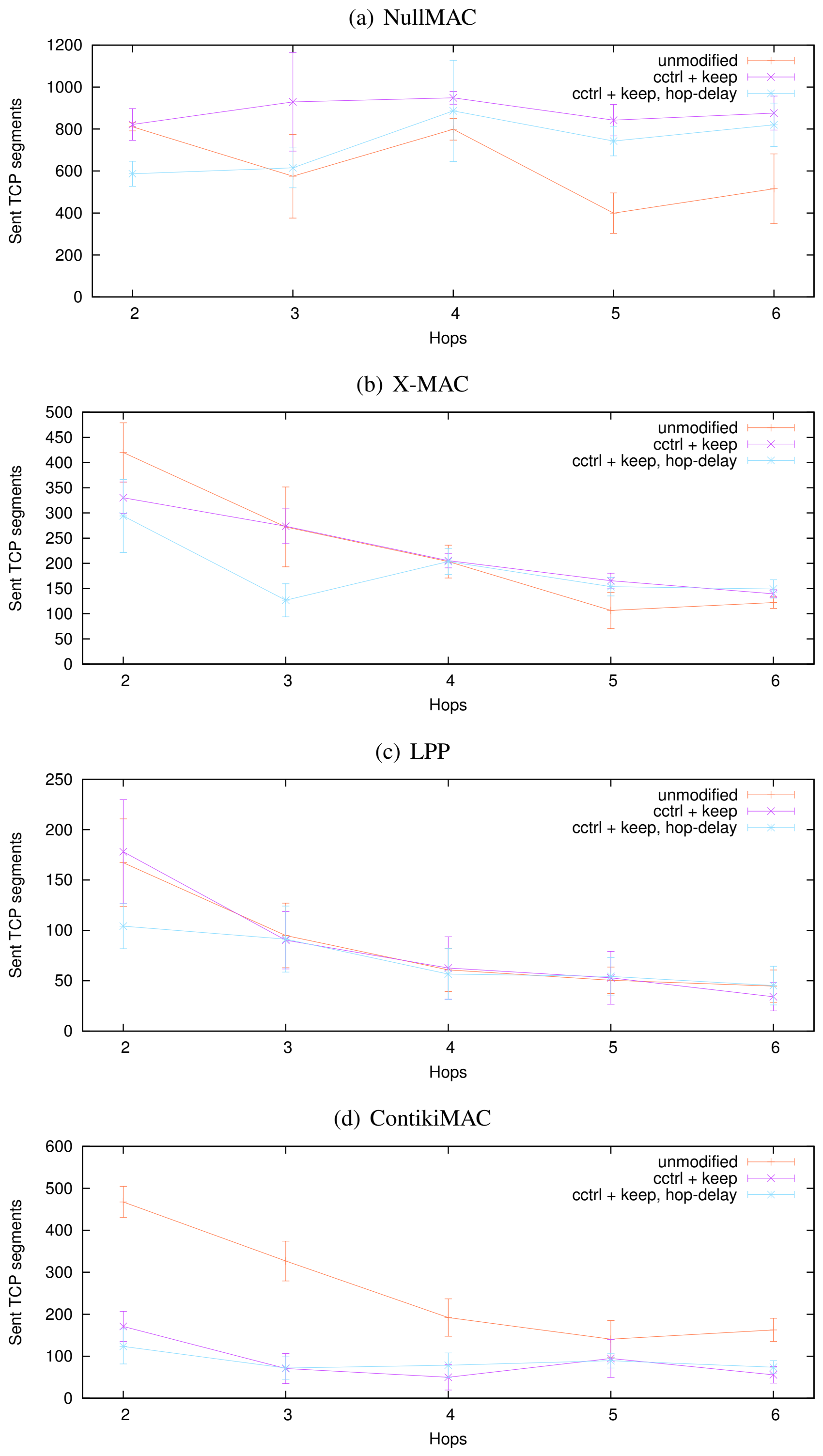


Figure 5.6: Throughput with hop-dependent retransmissions

5.3.3 Hop-distance Dependent Retransmissions

The next extension we implemented, was using hop-distance dependent retransmission timeouts instead of purely RTT dependent timeouts (cf. Section 4.3). The idea was to use a more robust distance metric as a base for calculating the retransmission timeouts and increase the retransmission timeout stronger, the further away a node is from the TCP server. The goal was to reduce the chance of concurrently initiated retransmissions of the same segment by different nodes.

However, as the experiments revealed, this modification could not deliver the expected advantage. As the Figures 5.6(a) and (b) illustrate, the introduction of hop-distance dependent retransmissions clearly reduced the TCP throughput of NullMAC and X-MAC sensor nodes. Thereby, the difference between the hop-distance dependent approach and the regular *cctrl* module is more dominant for the two and three hop experiment runs than it is for the runs using other path lengths. This is rather surprising, since the influence the hop-distance has to the calculated retransmission timeout is not as accentuated for nodes close to the receiver as it is for nodes further away (cf. Fig. 4.4). For example, at 6 hops distance, a node has a 2.5 times longer retransmission timeout when using hop-distance dependent retransmissions, compared to a node at the same distance that does not. Still, this significant difference has almost no influence on the achieved throughput.

A possible explanation for the decreased throughput of the 2 and 3 hop experiments is, that nodes closer than 3 hops use a slightly reduced retransmission timeout, which might lead to early, unnecessary retransmissions, which then interfere with the original TCP transmission.

We tested this *cctrl* extension with the receiver initiated Low Power Probing protocol (cf. Section 3.3.3) and ContikiMAC as well. As seen in Fig. 5.6(c), LPP shows almost the same behavior for all three approaches. Neither is there an improvement, nor an impairment when using the *cctrl* module or the hop-dependent retransmission in particular. When directly comparing LPP with X-MAC and ContikiMAC, it is noticeable that LPP has a rather low throughput. This is because LPP is not designed for high data rates, but for efficient broadcasting.

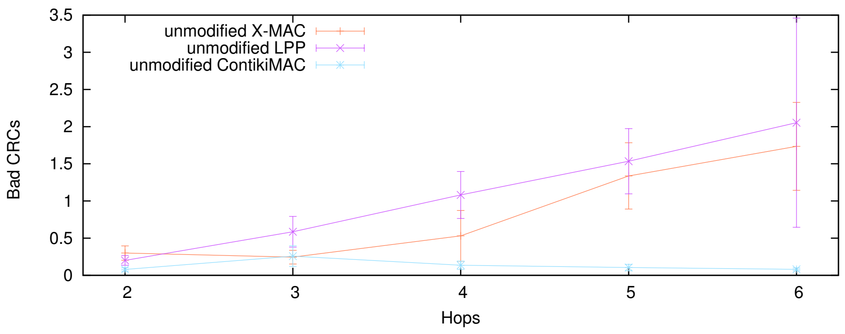
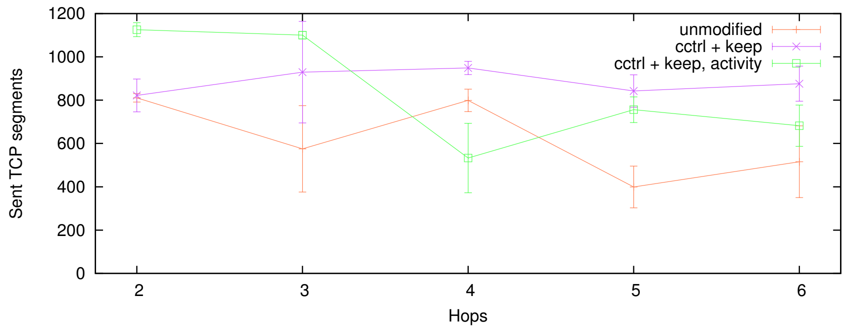


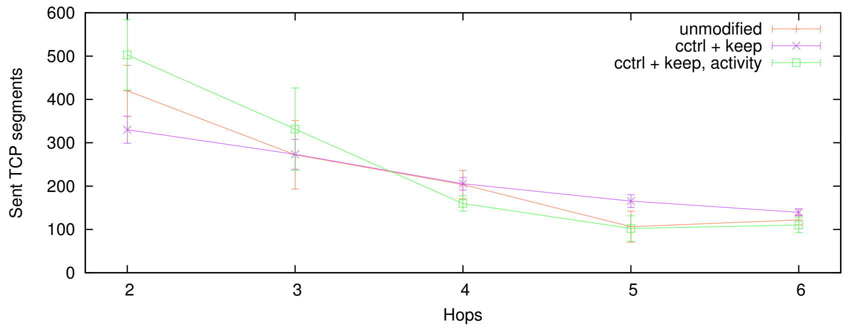
Figure 5.7: Amount of bad Rime packets per end-to-end TCP transmission

Also for ContikiMAC, illustrated in Fig. 5.6(d), there is only little difference in the throughput achieved by *cctrl* nodes using hop-dependent retransmission timeouts compared to the ones who do not. However, in comparison to the experiment with an unmodified network stacks, the *cctrl* module dramatically decreased the amount of transmitted TCP packets. It is apparent,

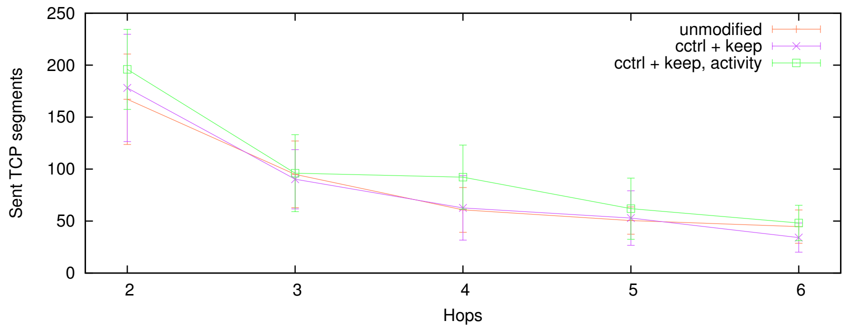
(a) NullMAC



(b) X-MAC



(c) LPP



(d) ContikiMAC

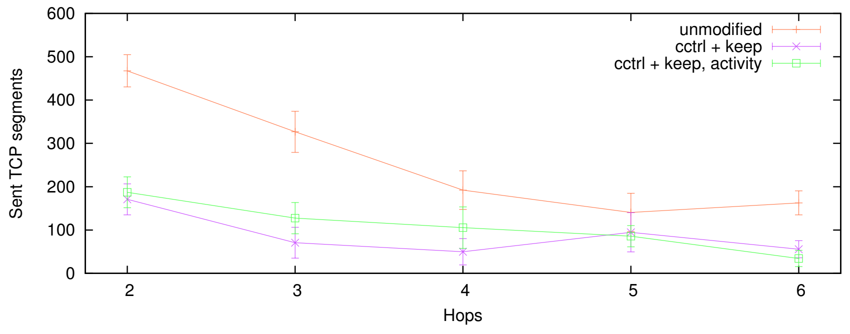


Figure 5.8: Throughput with activity monitoring

that ContikiMAC has already been heavily optimized. Introducing an additional retransmission scheme, which potentially increases the experienced traffic due to redundant retransmissions, is counterproductive.

This assumption is also supported when looking at the amount of Rime packets per TCP packet, which have been received with a bad Cyclic Redundancy Check (CRC) checksum. A packet with a bad checksum has to be dropped by the receiver as this indicates an incorrect reception. As depicted in Fig. 5.7, the arrival of a bad Rime packet is quite a rare event for non-*cctrl* ContikiMAC nodes in comparison to X-MAC and LPP nodes. Therefore, in combination with ContikiMAC, it is more likely for the *cctrl* module to spuriously initiate a retransmission, which then may interfere with the original transmission.

5.3.4 Activity Monitoring

Activity monitoring was introduced in Section 4.4 to have a MAC layer feedback about ongoing external transmissions available in the *cctrl* module. The feedback is used to lower the current retransmission timeout in situations with low activity, i.e., when no transmissions between neighboring nodes are detected.

Figure 5.8 illustrates how this modification affects the achieved throughput. NullMAC and X-MAC show similar outcome, as seen in Fig. 5.8(a) and (b), respectively. For both, when having activity dependent retransmissions, throughput is considerably increased for the two and three hop experiments, but drops for longer paths. With X-MAC the throughput even falls below the control benchmark set by unmodified forwarding nodes, when having paths of four hops and more.

As illustrated in Fig. 5.8(c), LPP profits from activity dependent retransmissions in a more general way. The activity monitoring extension is able to raise the throughput persistently over all tested path lengths, resulting in the highest amount of transmitted TCP packets with this MAC protocol so far. Since LPP is a duty cycling MAC protocol, it is also interesting, whether the improved throughput is accompanied by higher energy efficiency. As Fig. 5.9 reveals, activity monitoring is indeed able to reduce the per packet radio on time of the sensor nodes. However, this is only true for the 4, 5, and 6 hop experiments. If the data travels only 2 or 3 hops, the nodes keep their radios turned on longer.

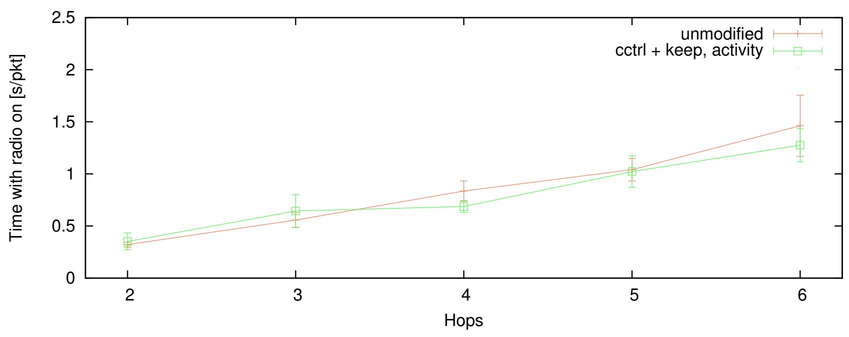
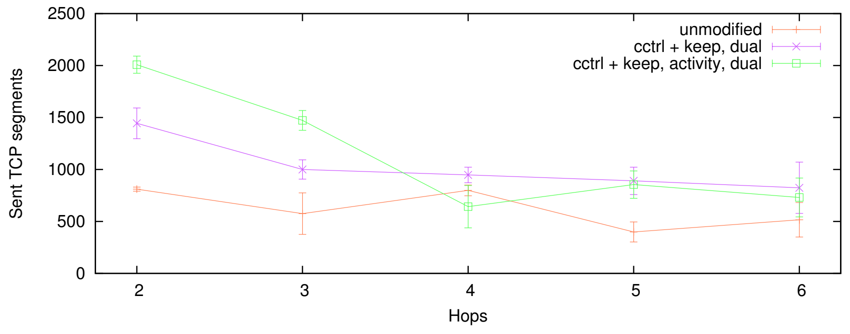
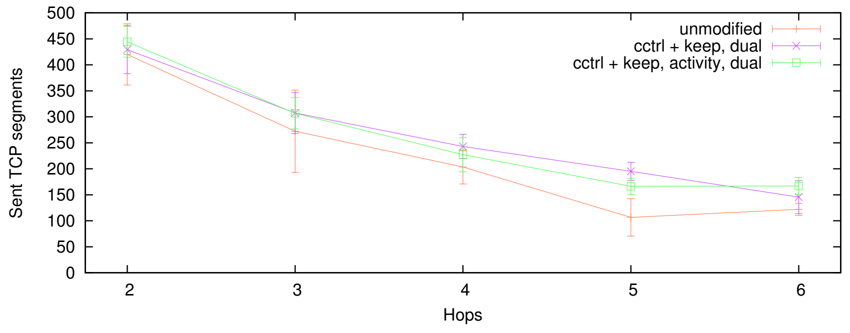


Figure 5.9: LPP's energy consumption with activity monitoring

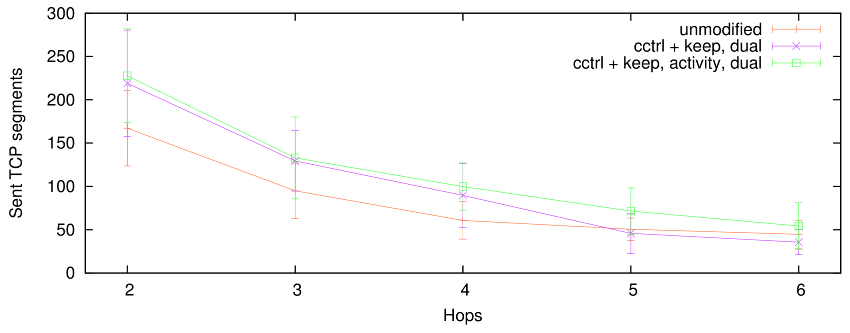
(a) NullMAC



(b) X-MAC



(c) LPP



(d) ContikiMAC

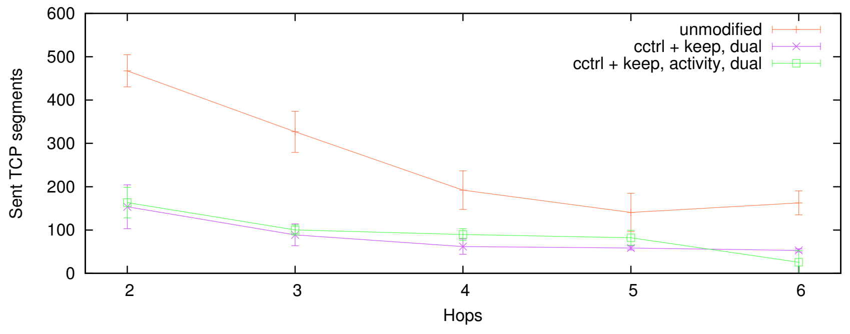


Figure 5.10: Throughput with multiple connections

Cctrl nodes running ContikiMAC, can also benefit from activity monitoring. As seen in Fig. 5.8(d), their achieved throughput is either increased (3 and 4 hop runs), or stays at roughly the same level. However, it is still not sufficient to match the high benchmark set by the control experiment using ContikiMAC nodes without *cctrl* module. As previously seen in Fig. 5.7, this can be explained by the very low amount of transmission errors unmodified ContikiMAC nodes experience, since this increases the chance of spuriously initiated retransmissions, which interfere with other ongoing transmissions.

5.3.5 Multiple Connections

The last modification we have made to Contiki's default behavior is to have a second TCP connection established, through which data is transmitted. Should there be an interruption in the packet flow of any connection, the TCP client node is still able to use the other one to continue the transmission. The actual implementation is described in Section 4.5.

We tested this extension in combination with the *cctrl* module configurations which previously delivered the best results: keeping segments cached to be able to perform multiple retransmissions (*keep*) and *keep* together with activity monitoring. The resulting change in throughput is illustrated in Fig. 5.10.

With NullMAC nodes relying on a second connection (cf. Fig. 5.10(a)) the amount of transmitted TCP packet is almost doubled across all tested hop distances, when compared to the result of unmodified nodes. If, additionally, the *cctrl* module receives activity feedback and adapts its retransmission scheme accordingly, this result is even further increased for the two and three hop routes. With the exception of the 4 hop run, the two alternatives are otherwise at the same level.

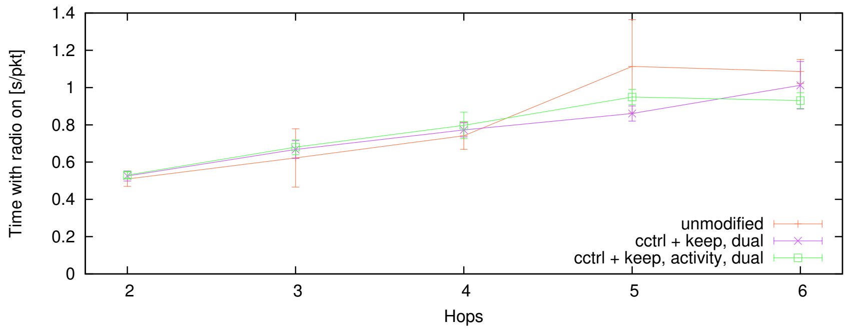
Also X-MAC nodes, illustrated in Fig. 5.10(b), benefit from having a second open connection. The nodes' throughput could be increased consistently for all tested path lengths, in the best case, the five hop experiment, by about 37%. A notable difference between the performance of the nodes using the activity extension and the ones which do not is hardly apparent.

The same is true regarding energy consumption, which is shown in Fig. 5.11(a). The average radio on time per packet, with and without activity monitoring is almost the same. Also when comparing with unmodified nodes, there is only little difference, at least for the 2, 3, and 4 hop experiments. A node has its radio turned on for approximately 500 ms to 700 ms per packet. A real improvement is only achieved in the 5 and 6 hop runs. Here, the average radio on time is lowered by roughly 100 ms. The *cctrl* module only achieves a better energy efficiency when the throughput can be raised by more than a certain amount.

A similar result is revealed by the LPP experiments. As seen in Fig. 5.10(c), the sensor nodes using LPP fundamentally increased the throughput when more than one TCP connection is available. Additionally, the presence of the activity monitoring extension leads to a slightly increased throughput at the 2, 3, and 4 hop experiment runs, compared to the runs without activity monitoring. Furthermore, the achieved throughput stays at a high level even at the 5 and 6 hop runs when the *cctrl* module uses activity dependent retransmissions.

To some extent similar to X-MAC is also the change in LPP's energy consumption, as illustrated in Fig. 5.11(b). For the experiments using short routes ($\leq$ 4 hops), there is almost no difference in the average radio on time. In the five and six hop experiments, however, the *cctrl*

(a) X-MAC



(b) LPP

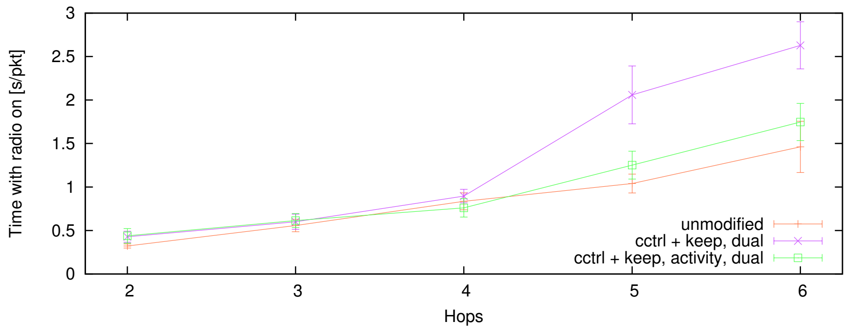


Figure 5.11: Energy consumption with multiple connections

nodes suddenly need to keep their radios turned on for a greater amount of time to successfully send one packet. With LPP, the increase in throughput would have to be even better to maintain energy efficiency.

In contrary, ContikiMAC still does not show any sign of improvement. When used in combination with the *cctrl* module, even with the second connection available, the throughput persistently remains at a very low level (cf. Fig. 5.10(d)). As with the previous experiments, the assumption is still valid that this is caused by falsely triggered retransmissions colliding with the original transmissions.

5.3.6 Overall Comparison

At the beginning of this section, we defined the goal of the single route scenario to find the most suitable *cctrl* configuration, which is capable of increasing transmission performance regardless of the used MAC protocol. While the previous results provide a good insight of how the *cctrl* module variants behave for different path lengths, a more general comparison might be hard to derive. Therefore, for each experiment configuration, we added up the mean values of each run with the different path lengths (i.e., 2 - 6 hops) and calculated its average, to obtain a single value. This not only makes the experiments easier comparable, it also takes into account, that in wireless sensor networks data is usually transported from and to multiple sensor nodes at varying distances.

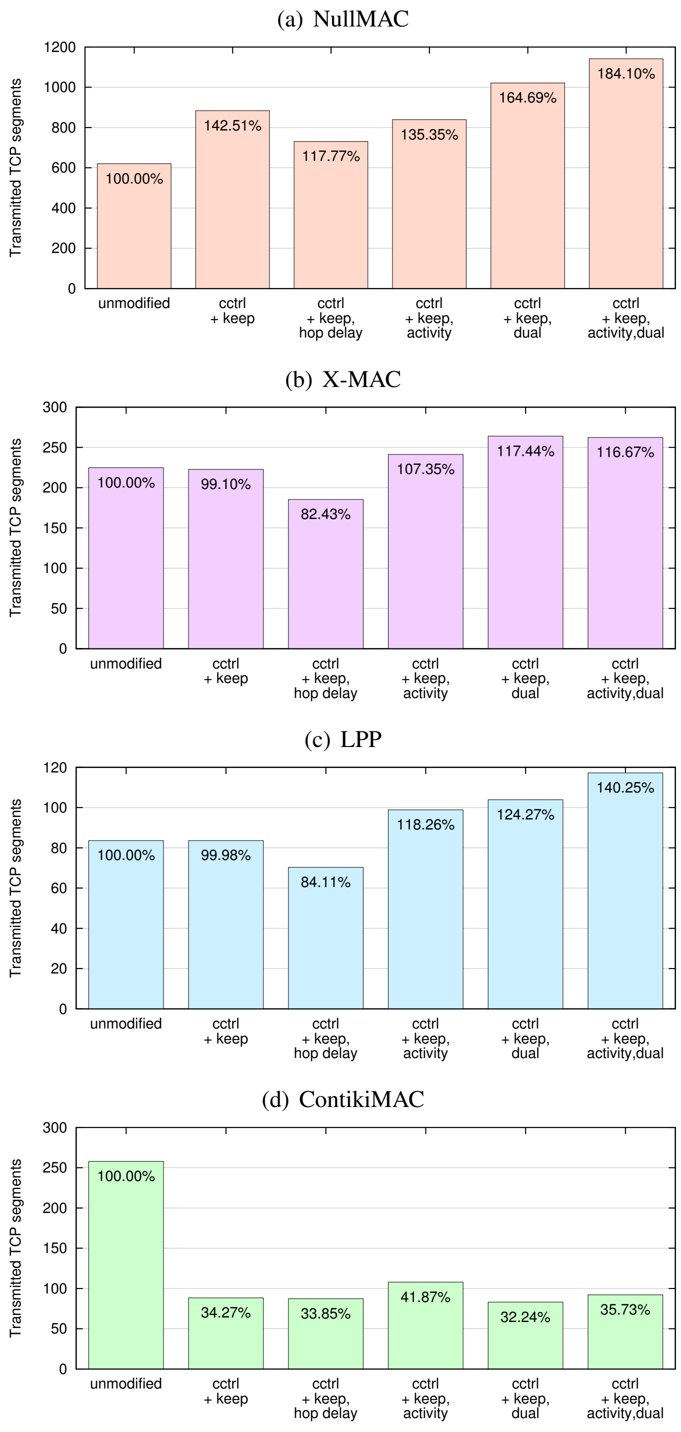


Figure 5.12: Throughput comparison of single route scenario

We use this approach to compare the achieved throughput in Fig. 5.12. It is apparent that in combination with NullMAC (a) the *cctrl* module can generally increase the sensor nodes' performance. The most significant benefit, with respect to throughput, results when the nodes are able to perform multiple local retransmissions with the aid of the *activity monitoring* and *multiple connections* extensions, with an 84% increase of totally transmitted TCP segments.

When using X-MAC, as depicted in Fig. 5.12(b), the outcome does not differ as distinctively as with NullMAC, but an overall increase in throughput could be achieved nevertheless. In average, up to 17% more packets could be delivered, when the *cctrl* nodes can rely on a second connection. The lack or presence of activity monitoring does not make any significant difference. Having hop-distance depending retransmissions, however, causes the throughput to decrease by about the same ratio.

X-MAC nodes' energy efficiency can also be increased slightly (cf. Fig. 5.13(a)). In the best case, the nodes can keep their radios turned off nearly 6% longer per packet.

LPP reveals to be influenced by the various *cctrl* extensions in a similar manner as X-MAC, as Fig. 5.12(c) illustrates. Again, introducing hop-distance dependent retransmission timeouts, results in the lowest experienced throughput. The other modifications achieve equal or better performance when compared to the results obtained by unmodified nodes. The peak, an increase of 40%, is accomplished by the combination of activity monitoring and having multiple connections.

Also when using LPP as MAC protocol the *cctrl* module can result in lower per packet power consumption. Interestingly however, as seen in Fig. 5.13(b), this is not achieved with the configuration that gained the highest throughput (cf. Fig. 5.12(c)). The highest increase in transmission performance also results in about 15% longer radio on time.

Concerning ContikiMAC, Fig. 5.12(d) clearly demonstrates that none of our suggested improvements harmonize with this MAC protocol in this particular scenario. The design of ContikiMAC can already cope with most causes of packet loss. One reason for this is, that ContikiMAC, in contrast to X-MAC and LPP, uses fewer MAC layer transmissions to transport a certain amount of data, as it directly uses data packets as strobes (cf. Section 3.3.2). Introducing a retransmission mechanism, such as the *cctrl* module, is actually counterproductive, because the local retransmissions only introduce additional interference. However, as Section 5.4 will show, this assumption is not valid in all situations.

For these reasons, it is hard to give a final, universally valid recommendation what *cctrl* configuration should be used. Having *cctrl* nodes with the default configuration, which includes the ability to generate multiple retransmissions with support of activity monitoring and multiple TCP connections, achieves the highest experienced throughput with all tested MAC protocols, but ContikiMAC. However, in combination with LPP, this is accompanied by increased energy consumption. Even though a *cctrl* configuration customized for a certain MAC protocol and network topology would be preferable, we think this particular configuration is capable of performing well in most situations. Therefore, we tested its performance again in more demanding situations, as explained in the following section.

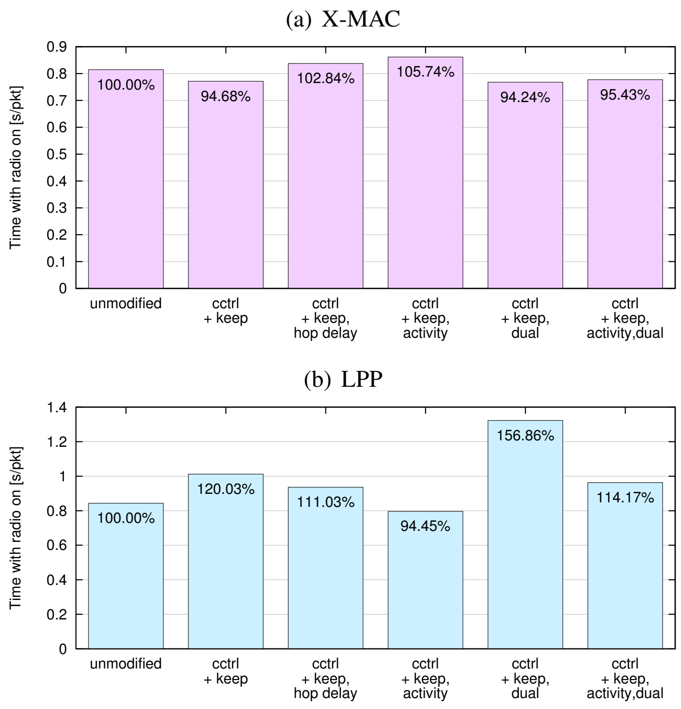


Figure 5.13: Energy consumption comparison of single route scenario

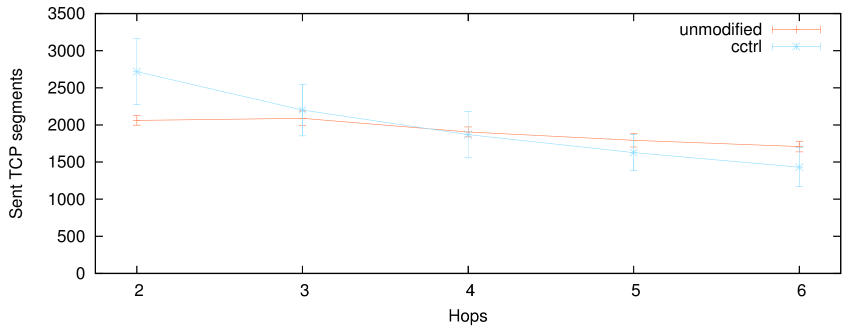
5.4 Double Route Scenario

In the double route scenario, we tested the performance of our *cctrl* module in a situation with increased external traffic. The employed topology has already been illustrated in Fig. 5.1 and Table 5.2. There are almost twice as many nodes participating in each experiment. With the exception of node 3, each node is part in one of two independent routes. Each route contains a TCP client node at one end that tries to send as many TCP data packets to the server node placed at the other end. Node 3 is member of both routes and has to forward traffic of both TCP connections. Therefore, around this particular node radio traffic is increased and packet collisions are more likely to happen.

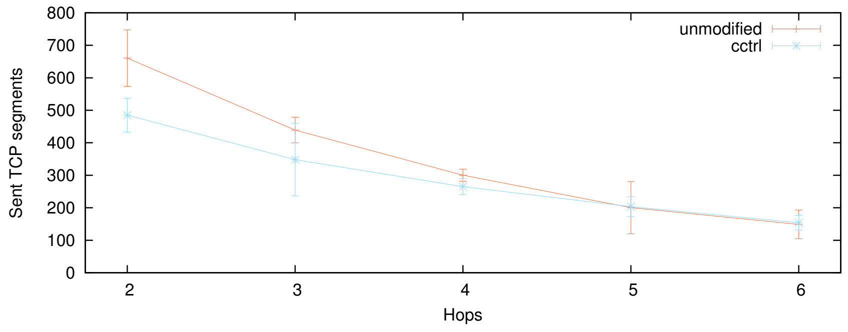
We tested this setup with network topologies consisting only of regular, unmodified Contiki nodes, as well as with topologies consisting only of nodes using the *cctrl* module in the configuration that showed the best overall performance in the previous section, i.e., nodes that perform multiple, activity dependent local retransmissions and rely on a second TCP connection. For simplicity, this configuration is just denoted as *cctrl* in the following figures, instead of *cctrl + keep, activity, dual*.

As before, each experiment was performed with 5 different route lengths, ranging from 2 hops to 6 hops, and each run was repeated 15 times. The following charts show the mean values of all runs and hop distances.

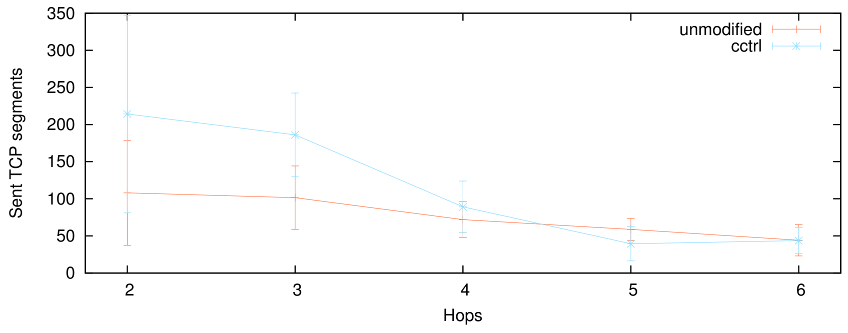
(a) NullMAC



(b) X-MAC



(c) LPP



(d) ContikiMAC

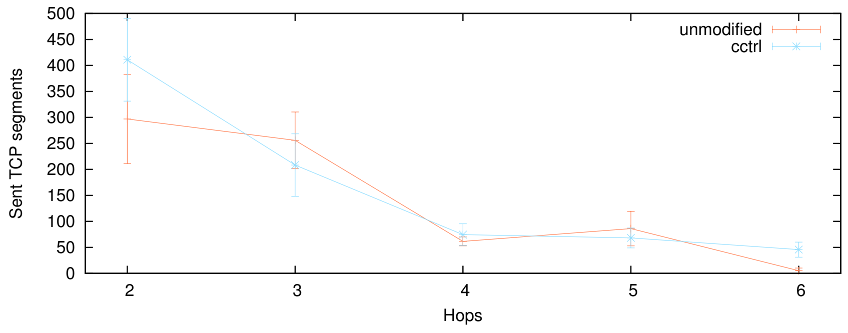


Figure 5.14: Throughput of double route scenario

5.4.1 Transmission Performance

Figure 5.14 compares the total amount of successfully received TCP packets over both established routes. Hence, it represents the transmission performance of the entire topology, not only of a particular subsection (e.g. a single route). Figure 5.15 illustrates the same, but with the results of the various path lengths averaged to a single value.

For all tested protocols, the highest throughput is achieved in the 2 hop experiment runs and decreases almost constantly as the hop distance increases in the subsequent runs, which is a behavior that has been observed many times. The degradation of throughput across n hops approximately follows the $\frac{1}{n}$ rule, where n equals the number of hops the data has to travel, cf. [33].

As expected, NullMAC still achieves the highest throughput of all MAC protocols, with just about 2000 successfully transmitted TCP data packets on average. As seen in Fig 5.14(a), the presence of the *cctrl* module further increases this amount, at least for the 2 and 3 hop runs. Combined with the slightly reduced throughput for the other runs, 3% more packets can be transmitted on average, when the *cctrl* module is used (c.f. Fig. 5.15). Although this is a considerable increase, the improvement is no longer as good as in the single route scenario.

Also the X-MAC nodes do not reflect the improvement experienced in the previous scenario tested. The nodes do not seem to be able to cope well with the additional transmissions performed by the *cctrl* module in an environment with already increased traffic, as the 2 and 3 hop results illustrated in Fig. 5.14(b) reveal. Therefore, the average transmission rate drops by about 16% (cf. Fig. 5.15).

LPP shows the greatest response to the *cctrl* module. As illustrated in Fig. 5.14(c), similar to NullMAC, LPP profits primarily when the data travels only few hops. At 4, 5 and 6 hops, LPP experiences a decrease in throughput, most notably for *cctrl* nodes. A possible explanation is that due to the presence of additional nodes, which periodically transmit probes to indicate reception readiness, more packet collisions occur. On average, the *cctrl* module manages to improve the TCP throughput by almost 49%. This is quite remarkable, as this result is even better by 9 percentage points than what the *cctrl* module could achieve with LPP nodes in the single route scenario.

Most surprising, however, is ContikiMAC's performance. In this scenario, with unmodified Contiki nodes, the average amount of sent TCP segments is less than half the amount of the single route scenario (cf. Fig. 5.12(d) and Fig.5.15). At the six hop run, the transmission almost stopped completely. And instead of a further decrease, as experienced previously, using *cctrl* nodes results in either an equivalent or higher throughput (cf. Fig. 5.14(d)). On average, a gain of 14% is achieved.

A reasonable explanation for this is, that ContikiMAC nodes without *cctrl* module, due to ContikiMAC's robust design (i.e. data packets as strobes, wake-up schedule learning, etc.), experienced considerably few interference in the single route scenario, in comparison with nodes running other MAC protocols. Therefore, when using the *cctrl* module, a major cause for packet loss were unnecessarily initiated local retransmissions, which rendered both, the original transmission, as well as the retransmission, invalid. In the dual route scenario, however, more nodes were participating, and the nodes were responsible for two independent packet flows. This led to additional packet collisions caused by the increased external traffic, which could be recovered by local retransmission.

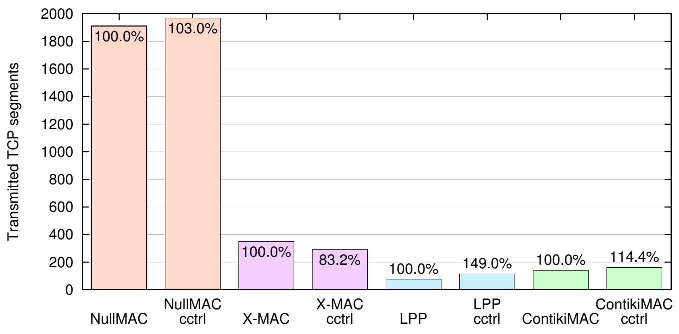


Figure 5.15: Throughput comparison of double route scenario

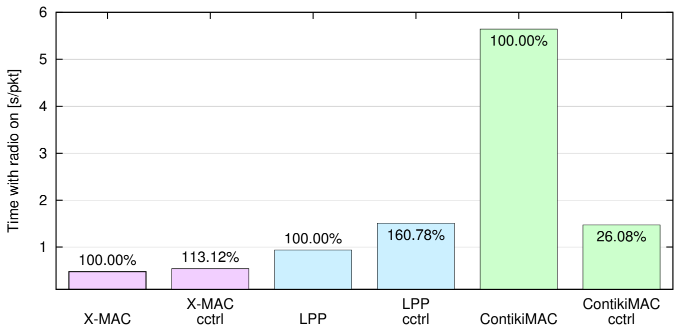


Figure 5.16: Energy efficiency of double route scenario

5.4.2 Energy Efficiency

The change in energy efficiency of the three duty-cycling MAC protocols is illustrated in Fig. 5.16. During this experiment, both X-MAC and LPP could not maintain their respective radio on time, when using the *cctrl* module. In addition to the loss of throughput, X-MAC nodes spend 13% more time with their radios turned on. With LPP nodes this is even exceeded, with a considerable increase of more than 60%.

ContikiMAC, on the other hand, greatly benefits from using the *cctrl* module in regard to energy efficiency. The average radio on time per packet is lowered by almost 73%, resulting in the most significant change of this parameter experienced during our experiments. Further analysis revealed, that without *cctrl* module, ContikiMAC experienced a major amount of subsequent packet losses, which ultimately caused the TCP connection to time out. Re-establishing the connection always means a waste of time and energy, as then no data can be transported.

5.5 Conclusion

In this Section, we presented the results of our sensor node experiments, which evaluated the influence of our *cctrl* module and its extensions in a real world environment. We showed that the four MAC protocols, with which our implementation was tested, led to great variances when comparing the individual results. Some protocols (e.g. NullMAC) generally proved to provide a good response to the local retransmission scheme, whereas others (e.g. ContikiMAC) performed worse.

But also the individual experiments showed inconsistencies when looking at the actual results of different hop distances. The unmodified NullMAC control experiment, for instance, achieved higher throughput on the 3-hop run than it did on the two and four-hop runs. Irregularities like this one are almost certainly caused by temporary changes of environmental conditions and are a typical phenomenon of real world wireless experiments. The Industrial, Scientific, and Medical (ISM) 2.4 GHz band, in which the TelosB radio operates, is shared with a lot of other common technologies, such as WLAN, Bluetooth, wireless audio equipment, cordless phones, and even microwave ovens, to just name a few. Reducing the influence such devices have in our experiments is almost impossible. But even though their influence causes somewhat inconsistent results, and makes it harder to reproduce experiments in equal conditions, they are part of the everyday environment and therefore, it is not bad to have them reflected in experimented results.

Nevertheless, with the proper combination of extensions, our *cctrl* module was capable of increasing throughput in almost any situation and in combination with any MAC protocol. The *cctrl* module can be configured to the sensor nodes' used MAC protocol and network topology with regard to network size and path lengths, to maximize the achieved performance.

Chapter 6

Conclusion & Outlook

6.1 Conclusion

For this master's thesis, we implemented the *cctrl* module, an add-on for Contiki's µIP stack. This module gives wireless sensor nodes, which are forwarding TCP traffic in a chain of nodes, the ability to cache TCP packets and perform local retransmissions. On top of a basic retransmission mechanism, we designed and implemented additional extensions for the *cctrl* module, which alter its initial behavior. These extensions allow a sensor node to perform multiple retransmissions of the same TCP packet in case of subsequent transmission errors, use the hop-distance as an alternative metric to calculate the retransmission timeout, and introduce a feedback mechanism for the MAC layer to reduce the retransmission delay in times of low external radio activity. Furthermore, we investigated in an extension which allows to simultaneously use multiple TCP connections.

We tested our implementations in an indoor wireless sensor node testbed using five routes of different size in two scenarios with low and high radio traffic, respectively. This is the first time, TCP optimization concepts for wireless sensor networks proposed in DTC and TSS have been evaluated using real sensor nodes. The results of both studies are based on simulations only. Furthermore, DTC and TSS have only been evaluated using CSMA MAC protocols. In real world applications, not relying on a radio duty-cycling MAC protocol would result in a considerable waste of energy. In this thesis, we also evaluated these concepts in combination with energy efficient MAC protocols.

Our experiment results revealed that the *cctrl* module can increase the sensor node's achieved throughput of TCP data in many situations. Comparing the results gained with different MAC protocols showed, however, that it is almost impossible to find a configuration of the *cctrl* module, which achieves maximal increase in throughput in combination with every MAC protocol. Due to the significant differences in design and behavior of the MAC protocols, the impact of the single *cctrl* extensions varies heavily.

Furthermore, we saw that the achieved result is affected not only by the choice of the MAC protocol, but also by the network's topology. For example, having X-MAC nodes in a network with 2 or 3-hop paths, using the *cctrl* module with multiple retransmissions and activity monitoring produces the best result. The same configuration, on the other hand, achieves the lowest results in a network with 5 and 6-hop paths. However, for almost any combination of

MAC protocol and network topology an ideal *cctrl* module configuration could be found, with the exception of ContikiMAC in the low traffic, single route scenario. The performance of this protocol could only be improved by the *cctrl* module in situations with heavy external radio traffic.

In general, the resulted increase in throughput ranged from 3% in the dual route scenario with NullMAC, up to 84% in the single route scenario with the same protocol. Out of the radio duty-cycling MAC protocols, X-MAC in combination with the *cctrl* module achieved the highest average throughput in the single route experiment.

Regarding energy-efficiency the results were similar, with some caveats. For certain experiment runs, the configuration which achieved the best throughput also lead to increased power consumption. But in many tested situations the *cctrl* module could slightly reduce the sensor nodes' radio on time, while simultaneously increasing throughput.

6.2 Outlook

In Section 4.5.1 we proposed a design of the *multiple connection* extension, which is completely integrated into the *cctrl* module and operates transparently from the application layer. As it was uncertain, whether or not this extension would be capable of delivering the desired results, we decided to skip the actual implementation and, instead, use a more simple proof of concept approach at the application layer, which does not reorder the segments at arrival. The implementation of this simplified approach is also described in Section 4.5. As experimental evaluations then revealed, this modification was actually capable of creating a major increase in throughput. Therefore, it would be reasonable to implement this approach as a proper *cctrl* module extension, and do another set of experiments to see whether it still performs as expected.

Another idea for a *cctrl* module extension is based on the behavior of TSS and DTC, which are introduced in Section 3.1. Both protocols rely on non-duty cycling MAC protocols, which are able to overhear a great amount of traffic. TSS and DTC can use overhearing of forwarded packets by neighboring sensor nodes as an implicit acknowledgment and react accordingly by, e.g., clear the retransmission cache, spoof TCP ACKs, etc. With our *activity monitoring* extension, we demonstrated that it is possible to receive MAC layer feedback independently of the used MAC protocol and that overhearing also works with energy-efficient MAC protocols, at least to a certain extent. Hence, we could port some additional overhearing strategies from TSS and DTC, which are not only based on the actual reception of an overheard packet, as our approach is, but which evaluate the content of the packet as well, e.g. to gain implicit knowledge about successfully forwarded packets. However, whether the sparse reception of overheard forwarded packets by a subsequent sensor node with radio duty-cycling MAC protocols is sufficient for such an approach, is questionable and would have to be evaluated.

Finally, when working with wireless sensor nodes, besides TCP throughput and energy efficiency, memory usage is always a critical point. Our current *cctrl* implementation caches up to two TCP packets per connection (i.e. one for each flow direction) as a whole in a buffer of predefined size. Since for a particular TCP connection, many header fields always have the same value for every packet (e.g. source and destination addresses, port numbers, etc.), they could be stored only once and the cached packets would then be regenerated prior to their retransmission.

Additionally, or as an alternative, header compression algorithms could be used to store the information more efficiently. Furthermore, the buffer could be allocated dynamically, to maximize the memory available for other applications. We did not investigate any further in this direction, as memory usage has no direct influence in the achieved throughput and only little in energy consumption and is, therefore, not in the scope of this thesis.

List of Acronyms

ARP Address Resolution Protocol

CRC Cycling Redundancy Check

CSMA Carrier Sense Multiple Access

DSL Digital Subscriber Line

EWMA Exponentially Weighed Moving Average

IAM Institut für Informatik und angewandte Mathematik

IANA Internet Assigned Numbers Authority

ICMP Internet Control Message Protocol

IEEE Institute of Electrical and Electronics Engineers

IP Internet Protocol

ISM Industrial, Scientific, and Medical

LED Light-emitting Diode

LPP Low Power Probing

MAC Media Access Control

OS Operating System

RAM Random-access Memory

RFC Request for Comments

ROM Read-only Memory

RTT Round Trip Time

RVS Rechnernetze und Verteilte Systeme

TCP Transmission Control Protocol

TTL	Time To Live
UART	Universal Asynchronous Receiver Transmitter
UDP	User Datagram Protocol
USB	Universal Serial Bus
WLAN	Wireless Local Area Network
WSN	Wireless Sensor Network
XML	Extensible Markup Language

Bibliography

[1] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, “Wireless Sensor Networks: A Survey,” *IEEE Communications Magazine*, vol. 38, no. 4, pp. 393–422, March 2002.

[2] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh, “Fidelity and Yield in a Volcano Monitoring Sensor Network.” USENIX Symposium on Operating Systems Design and Implementation (OSDI), Seattle, USA, November 2006, pp. 27–39.

[3] G. Simon, M. Maroti, A. Ledeczi, G. Balogh, B. Kusy, A. Nadas, G. Pap, J. Sallai, and K. Frampton, “Sensor Network-Based Countersniper System.” ACM Conference on Embedded Networked Sensor Systems (SenSys), Baltimore, USA, November 2004, pp. 1–12.

[4] J. Postel and J. Reynolds, “Telnet Protocol Specification,” RFC 854 (Standard), Internet Engineering Task Force, May 1983, updated by RFC 5198. [Online]. Available: http://www.ietf.org/rfc/rfc854.txt

[5] J. Zhao and R. Govindan, “Understanding Packet Delivery Performance in Dense Wireless Sensor Networks.” ACM Conference on Embedded Networked Sensor Systems (SenSys), Los Angeles, USA, November 2003, pp. 192–204.

[6] P. J. S. Thomas Starr, John M. Cioffi, *Understanding Digital Subscriber Line Technology*. Prentice Hall, Upper Saddle River, USA, 1999.

[7] A. Dunkels, J. Alonso, T. Voigt, and H. Ritter, “Distributed TCP Caching for Wireless Sensor Networks.” Mediterranean Ad-Hoc Networks Workshop (Med-Hoc-Net), Bodrum, Turkey, June 2004, pp. 13–28.

[8] T. Braun, T. Voigt, and A. Dunkels, “TCP Support for Sensor Networks.” Wireless On demand Network Systems and Services (WONS), Obergurgl, Austria, January 2007, pp. 162–169.

[9] H. Balakrishnan, S. Seshan, E. Amir, and R. H. Katz, “Improving TCP/IP Performance over Wireless Networks.” International Conference on Mobile Computing and Networking (MobiCom), Berkeley, USA, November 1995, pp. 2–11.

[10] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, “TCP Selective Acknowledgment Options,” RFC 2018 (Proposed Standard), Internet Engineering Task Force, Oct. 1996. [Online]. Available: http://www.ietf.org/rfc/rfc2018.txt

[11] A. Dunkels, B. Groenvall, and T. Voigt, "Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors." IEEE Workshop on Embedded Networked Sensors (EmNets), Tampa, Florida, November 2004, pp. 455–462, http://www.sics.se/contiki/.

[12] Contiki team members. (2011, March) About contiki. [Online]. Available: http://www.sics.se/contiki/about-contiki.html

[13] A. Dunkels, F. Österlind, and Z. He, "An Adaptive Communication Architecture for Wireless Sensor Networks." ACM Conference on Embedded Networked Sensor Systems (SenSys), Sydney, Australia, November 2007, pp. 335–349.

[14] A. Dunkels. (2011, April) The uIP TCP/IP stack. [Online]. Available: http://www.sics.se/~adam/uip/uip-1.0-refman/

[15] R. Braden, "Requirements for Internet Hosts - Communication Layers," RFC 1122 (Standard), Internet Engineering Task Force, Oct. 1989, updated by RFCs 1349, 4379, 5884, 6093. [Online]. Available: http://www.ietf.org/rfc/rfc1122.txt

[16] A. Prayati, C. Antonopoulos, T. Stoyanova, C. Koulamas, and G. Papadopoulos, "A Modeling Approach on the TelosB WSN Platform Power Consumption," vol. 83. New York, NY, USA: Elsevier Science Inc., August 2010, pp. 1355–1363.

[17] J. Polastre, J. Hill, and D. Culler, "Versatile Low Power Media Access for Wireless Sensor Networks." ACM Conference on Embedded Networked Sensor Systems (SenSys), Baltimore, USA, November 2004, pp. 95–107.

[18] W. Ye, J. Heidemann, and D. Estrin, "An Energy Efficient MAC Protocol for Wireless Sensor Networks." IEEE International Conference on Computer Communications (INFOCOM), New York, USA, June 2002, pp. 1567–1576.

[19] T. V. Dam and K. Langendoen, "An Adaptive Energy Efficient MAC Protocol for Wireless Sensor Networks (TMAC)." ACM Conference on Embedded Networked Sensor Systems (SenSys), Los Angeles, USA, November 2003, pp. 171–180.

[20] A. El-Hoiydi and J. D. Decotignie, "WiseMAC: An Ultra Low Power MAC Protocol for Multihop Wireless Sensor Networks." International Workshop on Algorithmic Aspects of Wireless Sensor Networks (ALGOSENSORS), Turku, Finland, July 2004, pp. 18–31.

[21] M. Buettner, V. Gary, E. Anderson, and R. Han, "X-MAC: A Short Preamble MAC Protocol for Duty-cycled Wireless Sensor Networks." ACM Conference on Embedded Networked Sensor Systems (SenSys), Boulder, USA, November 2006, pp. 307–320.

[22] A. Dunkels, L. Mottola, N. Tsiftes, F. Osterlind, J. Eriksson, and N. Finne, "The Announcement Layer: Beacon Coordination for the Sensornet Stack." European Conference on Wireless Sensor Networks (EWSN), Bonn, Germany, February 2011, pp. 211–226.

[23] R. Musaloiu, C. Liang, and A. Terzis, "Koala: Ultra-Low Power Data Retrieval in Wireless Sensor Networks." ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN), St. Louis, USA, April 2008, pp. 421–432.

[24] P. Hurni and T. Braun, “MaxMAC: a Maximally Traffic-Adaptive MAC Protocol for Wireless Sensor Networks.” European Conference on Wireless Sensor Networks (EWSN), Coimbra, Portugal, February 2010, pp. 289–305.

[25] M. Anwander, G. Wagenknecht, T. Braun, and K. Dolfus, “BEAM: A Burst-Aware Energy-Efficient Adaptive MAC Protocol for Wireless Sensor Networks.” International Conference on Networked Sensing Systems (INSS), Kassel, Germany, June 2010, pp. 195–202.

[26] D. Moss and P. Levis, “Box-macs: Exploiting Physical and Link Layer Boundaries in Lowpower Networking,” Technical Report SING-08-00, Stanford University, Tech. Rep., 2008.

[27] S. Yanjun, O. Gurewitz, and D. B. Johnson, “RI-MAC: A Receiver Initiated Asynchronous Duty Cycle MAC Protocol for Dynamic Traffic Loads in Wireless Sensor Networks.” ACM Conference on Embedded Networked Sensor Systems (SenSys), Raleigh, USA, November 2008, pp. 1–14.

[28] C. Boano, T. Voigt, N. Tsiftes, L. Mottola, K. Roemer, and M. Zuniga, “Making Sensornet MAC Protocols Robust Against Interference.” European Conference on Wireless Sensor Networks (EWSN), Coimbra, Portugal, February 2010, pp. 272–288.

[29] “TelosB Datasheet,” MEMSIC Inc. [Online]. Available: http://www.memsic.com/support/documentation/wireless-sensor-networks/category/7-datasheets.html?download=152%3Atelosb

[30] P. Hurni, G.Wagenknecht, M. Anwander, and T. Braun, “A Testbed Management System for Wireless Sensor Network Testbeds (TARWIS).” European Conference on Wireless Sensor Networks (EWSN) Demo Session, Coimbra, Portugal, February 2010, pp. 33–35.

[31] I. Chatzigiannakis, S. Fischer, C. Koninis, G. Mylonas, and D. Pfisterer, “WISEBED: an Open Large-Scale Wireless Sensor Network Testbed,” *Sensor Applications, Experimentation, and Logistics*, pp. 68–87, 2010.

[32] B. Carpenter, D. Estrin, D. Farinacci, G. Finn, C. Graff, D. Katz, J. Postel, A. Malis, and R. Ullmann, “IP Option Numbers,” April 2011. [Online]. Available: http://www.iana.org/assignments/ip-parameters

[33] F. Österlind and A. Dunkels, “Approaching the Maximum 802.15.4 Multi-hop Throughput.” ACM Workshop on Embedded Networked Sensors (HotEmNets), Charlottesville, USA, June 2008.