University of Ljubljana
Faculty of Computer and Information Science

Jan Berdajs

Time Synchronization Issues in a MAC Protocol for Wireless Sensor Networks

DIPLOMA THESIS

UNIVERSITY STUDY PROGRAMME
COMPUTER AND INFORMATION SCIENCE

Mentor: dr. Andrej Brodnik
Co-mentor: dr. Evgeny Osipov, Luleå tekniska universitet

Ljubljana 2012

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Jan Berdajs

Problematika usklajevanja časa v protokolu MAC za brezžična senzorska omrežja

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in informatika

Mentor: dr. Andrej Brodnik
Somentor: dr. Evgeny Osipov, Luleå tekniska universitet

Ljubljana 2012

Besedilo je oblikovano z urejevalnikom besedil LaTeX.
Text written in LaTeX.

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 01867/2012
Datum: 05.09.2012

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **JAN BERDAJS**

Naslov: **PROBLEMATIKA USKLAJEVANJA ČASA V PROTOKOLU MAC ZA BREZŽIČNA SENZORSKA OMREŽJA**

TIME SYNCHRONIZATION ISSUES IN A MAC PROTOCOL FOR WIRELESS SENSOR NETWORKS

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Zmogljivost strojne opreme se neprestano povečuje, medtem ko se njena fizična velikost zmanjšuje. Zaradi tega lahko srečamo bolj ali manj zmogljiv računalnik na vsakem koraku. Ena od najobičajnejših funkcij, ki jo opravljajo takšni sistemi, je opazovanje in zaznava okolja ter odziv na spremembe. Za povečanje učinkovitosti in kakovosti odzivov posamične enote povezujemo v (brezžična) omrežja - govorimo o vseprisotnih sistemih.

V nalogi preučite možnosti zmanjšanja porabe energije pri posamezni enoti za potrebe komuniciranja. Pri tem posvetite posebno pozornost potrebi po časovni usklajenosti med enotami sistema. Zahtevane spremembe implementirajte in izmerite prihranke.

Mentor:
doc. dr. Andrej Brodnik

Somentor:
doc. dr. Evgeny Osipov

Dekan:
prof. dr. Nikolaj Zimic

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

No. of dissertation: 01867/2012
Date: 14.12. 2012

University of Ljubljana, Faculty of Computer and Information Science issues the following dissertation:

Candidate: **JAN BERDAJS**

Title: **TIME SYNCHRONIZATION ISSUES IN A MAC PROTOCOL FOR WIRELESS SENSOR NETWORKS**

Type of dissertation: Undergraduate dissertation

Topic of the thesis:

Hardware is constantly getting more and more capable, while its physical size shrinks. Consequently, we meet some kind of computer more or less everywhere. One of the most common use cases is monitoring and sensing, and then responding on the changes in the environment. To improve efficiency and quality of responses we connect individual units in network - we are talking of ubiquitous systems.

In your research study the possibilities of decreasing the energy use for communication purposes at the individual unit. Special attention should be payed to the importance of time synchronization between the units. Implement the necessary changes and measure the obtained savings.

Mentor:

Ass. Prof. Andrej Brodnik, PhD

Co-mentor:

Univ. Lekt. Evgeny Osipov, PhD

Dean of Faculty of Computer and Information Science:

Prof. Nikolaj Zimic, PhD

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Jan Berdajs, z vpisno številko **63070037**, sem avtor diplomskega dela z naslovom:

Time Synchronization Issues in a MAC Protocol for Wireless Sensor Networks

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom dr. Andreja Brodnika in somentorstvom dr. Evgenyja Osipova,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 14. decembra 2012 Podpis avtorja:

Zahvaljujem se mentorju dr. Brodniku za pomoč pri izbiri teme in iskanju somentorja na tuji univerzi. Rad bi se mu zahvalil tudi za podporo, ki mi jo je izkazal tekom izdelave diplomskega dela in pri objavi članka. Rad bi se mu zahvalil, da mi je omogočil obisk MACOM konference v Dublinu, kjer smo članek objavili. Poleg tega bi se mu rad zahvalil tudi za njegovo predanost delu na fakulteti in njegov izjemno pozitiven odnos do študentov.

Rad bi se zahvalil tudi dr. Osipovu za pomoč predvsem pri teoretičnem delu diplomskega dela, ter še posebej asistentu Laurynasu Riliskisu za potrpežljivost in pomoč tako pri teoretičnem, kot pri praktičnem delu.

Zahvalil bi se tudi staršem, za podporo in potrpežljivost tekom celotnega študija.

I give my acknowledgement to my mentor dr. Brodnik for helping me determine the research area for my diploma thesis and finding a co-mentor for me at Luleå tekniska universitet. I would also like to acknowledge his continued support during my work and also enabling my visit of the MACOM conference in Dublin, where I published a short paper together with Laurynas Riliskis.

I would like to acknowledge dr. Osipov for helping with the theoretical part of the paper. Especially, I would like to acknowledge PhD candidate Laurynas Riliskis for his extensive help with both the theoretical part and implementation, and for sharing his knowledge about HMAC and networks in general.

I would like to thank my parents for supporting me during my studies.

To the Ruby programming language and its chief designer Yukihiro Matsumoto, for a constant source of inspiration and for making programming fun. Matz is nice and so we are nice (MINASWAN).

Contents

Povzetek

Področje brezžičnih senzorskih omrežij je vse bolj pomembno raziskovalno področje. Razlog je v tehnološkem napredku, katerega posledica je zniževanje cene strojne opreme ter zniževanje porabe energije, kar pozitivno vpliva na uporabnost brezžičnih senzorskih omrežij.

Brezžična senzorska omrežja navadno sestojijo iz večjega števila poceni vozlišč, ki so navadno baterijsko napajana. Če za polnjenje baterij uporabimo fotovoltaične celice, lahko omrežje postane tudi energetsko samozadostno. Zaradi omejene količine energije je pomembna nizka poraba. Pogosto se poleg strojne opreme z nizko porabo uporabljajo dodatni prijemi, kot npr. periodično izklapaljanje (spanje) naprav. Kot posledica energetskih omejitev je zelo omejen tudi domet naprav, njihova fizična velikost ter zmogljivost.

Diplomsko delo obravnava izboljšave MAC (*Media Access Control*) protokola HMAC (*Hash MAC*), ki je namenjen uporabi v brezžičnih senzorskih omrežjih. Naloga MAC protokola je upravljanje dostopa do medija za komuniciranje (v tem primeru gre za brezžično omrežje). Protokol HMAC zagotavlja usklajen dostop do medija na podlagi razporejanja komunikacije posameznih naprav po času ter frekvenčnem prostoru. Časovno razporejanje je še posebej energetsko učinkovito, saj lahko naprave izklopimo izven njihovega časovnega okna.

Zaradi uporabe časovnega razporejanja je v protokolu HMAC pomembna sinhronizacija časa med posameznimi napravami. Je pravzaprav ključnega pomena za zanesljivo delovanje protokola. Obstoječa implementacija protokola ni podpirala sinhronizacije časa, zato je bil eden glavnih ciljev diplom-

skega dela izbira tehnike sinhronizacije ter implementacija le-te.

Primer uporabe izboljšanega protokola HMAC, predstavljenega v diplomskem delu, je projekt iRoad [Ril10], ki ga razvijajo na Luleå tekniska universitet (LTU). Namen projekta je razvoj inteligentnih opozorilnih znakov na cestah in temelji na uporabi brezžičnega senzorskega omrežja. Naprave, postavljene ob cestišču, so opremljene s senzorji za zaznavanje bližajočih se vozil. Ob zaznanem vozilu se preko omrežja pošlje sporočilo proti napravam v krožnem križišču, ki so opremljene z močnimi LED (*Light emitting diode*) diodami. Namen sistema je opozarjanje ostalih udeležencev v prometu o bližujočem se vozilu.

Glavna motivacija diplomskega dela sta bili naslednji ključni zahtevi projekta iRoad:

1. Dovolj nizka poraba energije, da lahko naprave zdržijo brez polnjenja čez noč.

2. Hitrost prenosa sporočil mora presegati hitrost vozila, ki se približuje krožnemu prometu.

Protokol HMAC je bil zasnovan za nizko porabo energije. Pri našem delu je bilo pomembno, da je sinhronizacija časa po eni strani energetsko nepotratna in po drugi strani dovolj natančna, da omogoča dovolj visoko hitrost prenosa podatkov. Pri komunikaciji morajo namreč naprave implementirati določene zakasnitve, ki premoščajo razliko v percepciji časa med njimi. Te zakasnitve so v veliki meri posledica prav razlik v percepciji časa.

Pogoj za izbiro tehnike sinhronizacije časa so bile meritve razlik v frekvenci ure med posameznimi napravami. Namreč, ker se v brezžičnih senzorskih omrežjih uporablja relativno poceni strojna oprema, ure v posameznih napravah ne tečejo s popolnoma enako frekvenco, tudi če gre za naprave iz iste serije. Te razlike so majhne, vendar v daljših časovnih intervalih povzročijo veliko odstopanje. Za premoščanje teh odstopanj je potrebna sinhronizacija časa in za izbiro tehnike ter določanje parametrov pri sinhronizaciji je pomembno ugotoviti, za kakšno odstopanje gre. V Poglavju 3.1 smo na naši

strojni opremi izmerili povprečno odstopanje 800 ppm, kar pomeni, da se odstopanje vsako sekundo poveča za 800 mikrosekund.

V Poglavju 2.1 so predstavljene različne tehnike sinhronizacije časa ter ocenjena njihova primernost za uporabo v protokolu HMAC. Na podlagi primernosti ter drugih dejavnikov (obstoječe implementacije, razširjenost) smo izbrali tehniko FTSP (*Flooding time synchronization protocol*). V Poglavju 3.1.2 je predstavljena evalvacija tehnike na naši strojni opremi.

Za učinkovito uporabo FTSP v protokolu HMAC je bilo potrebno na novo zasnovati implementacijo protokola, v manjšem obsegu pa tudi prilagoditi samo specifikacijo. V Poglavju 3.2 so podrobno opisane potrebne spremembe, implementacija FTSP, evalvacija nove rešitve ter meritve porabe energije.

Rezultat diplomskega dela je protokol HMAC z zmožnostjo sinhronizacije časa, ki dosega zahteve projekta iRoad. Obsega tako teoretični del - rešitve problemov obstoječega protokola HMAC, izbira tehnike sinhronizacije časa ter način vključitve le te, kot tudi praktični del - delјuča implementacija opisane rešitve v celoti. V zaključku so predstavljeni še predlogi za nadaljno delo.

Ključne besede: brezžična senzorska omrežja, TinyOS, MAC protokol, HMAC protokol, sinhronizacija časa, zamik ure, FTSP

Abstract

HMAC is a MAC protocol developed for wireless sensor networks at LTU and is based on time division. Consequently it is highly susceptible to time deviations between nodes, but is good in terms of low power consumption. There is currently no time synchronization implementation for the HMAC protocol, which has an impact on the reliability and performance of the protocol. In this thesis we focus on implementing time synchronization for HMAC on TinyOS, ensuring reliable function of HMAC, therefore enabling its use in production environments. Clock drift was measured on our target platforms and the performance of the reference flooding time synchronization protocol (FTSP) implementation was evaluated. We optimized the performance of FTSP on our platforms by adding microsecond precision support and fixing FTSP-related bugs present in the radio driver layer. The implementation of HMAC for TinyOS was redesigned, with the goal of improving performance, reliability and code manageability. We also integrated FTSP directly into HMAC, improving HMAC reliability even further. The final tests in this thesis show the achieved reliability, synchronization and low power consumption.

Keywords: wireless sensor networks, TinyOS, MAC protocol, HMAC protocol, time synchronization, clock drift, FTSP

Chapter 1

Introduction

Wireless sensor networks (WSN) are becoming an increasingly popular research topic. Due to constant technological advancements, hardware is becoming cheaper and hardware power consumption is decreasing, which both make WSNs increasingly more viable for use in real-world applications.

WSNs are generally comprised of numerous low-cost nodes, which are typically battery-powered. The batteries can be charged by solar energy to make the network self-sufficient. Due to the limited power supply, low power consumption is also a factor, thus besides using low-power hardware, devices are often periodically put into sleep mode to preserve power. Furthermore, the range at which these devices can communicate is limited, the devices tend to be small in size and they are limited in computational abilities, as well.

In this thesis, we work on improving HMAC, a MAC protocol developed at Luleå University of Technology, which takes WSN capabilities and limitations into account. A MAC protocol is used for communication and controls access to the communication medium - in the case of WSNs, the wireless network. HMAC uses time and frequency division to control access to the network. Time division provides benefits in regard to power consumption - the device can be put to sleep outside of its allocated time slot.

Due to the use of time division in HMAC, maintaining a common notion of

time becomes important, i.e. time needs to be synchronized between nodes. A time synchronization technique is one of the tools that can be used to ensure reliable function of HMAC.

WSN applications have different performance requirements, thus there are many different communication and synchronization protocols for use in WSNs. For HMAC, it is desirable that the whole network is synchronized to the same time, as opposed to synchronizing localized clusters in the network. This somewhat limits the choice of time synchronization protocols that are suitable. There is no time synchronization currently implemented in HMAC, which prevents it from working reliably for extended periods of time, thus making this an interesting research topic.

An example of a real-world application where HMAC can be used, specifically with improvements suggested in this thesis, is the iRoad project [Ril10] that is being developed at LTU. The aim of iRoad is to make intelligent road markings through use of low-power devices such as mulle, which was used in this thesis. One of the use cases is to notify pedestrians of cars approaching a roundabout. To accomplish this, nodes are placed beside the road leading to the roundabout. The nodes are equipped with sensors that can detect cars passing by - when a car is detected, a notification is sent through the network towards the nodes closest to the roundabout. Receiving nodes light up LEDs, which lets pedestrians know that a car is approaching. To make this possible, a network protocol is needed that allows for notifications to travel through the network faster than the car is traveling on the road, while still not consuming a lot of power so that the devices can operate only on batteries recharged during the day by solar power.

The work in this thesis addresses the following research questions in particular:

1. How accurate are clocks used in WSN nodes and how fast will they drift apart between different nodes?

2. How do time synchronization protocols work and how well does the FTSP protocol perform on our platforms?

3. How to redesign HMAC to improve reliability, performance and code manageability?

4. How to integrate the FTSP protocol into HMAC to achieve better time synchronization?

5. How to improve the HMAC protocol and its implementation even further, with regard to performance?

The thesis is organized as follows: Chapter 2 introduces the time synchronization problem in the context of wireless sensor networks and provides an overview of existing work. An overview of medium access control protocols for use in WSNs, such as HMAC, is also provided. Chapter 3 discusses the experiments and work done to improve HMAC and choose a suitable time synchronization protocol that can be used in conjunction with it. Chapter 4 summarizes the work done and suggests further research topics.

Chapter 2

Background and Related Work

2.1 Time Synchronization

Time synchronization plays an important role in wireless sensor networks, as it does in distributed systems in general. Time synchronization is necessary in certain applications to fulfill timing demands. Due to clock crystal impurities and environmental conditions, devices experience clock drift [KW05].

Time synchronization is also important for MAC protocols based on TDMA (Time Division Multiple Access), MAC protocols with coordinated wakeup (e.g. Zigbee) and MAC protocols with duty cycling.

To have a better understanding of the magnitude of error caused by clock drift, a consequence of unmatched oscillator frequencies, let us examine some typical values. In [vGR03], authors observed a 50 ppm (parts per million) drift. In other words there were 50 additional (or missing) oscillations in the amount of time needed for one million oscillations at the nominal rate. They also state that a 20 - 50 ppm drift is typical for quartz crystals. A 50 ppm drift means a drift of 50 μs per second. In [EGE02], authors estimated the typical drift for crystal oscillators to be in the range of $1 - 100$ μs per second. To put the number in perspective, 50 ppm drift accounts for a 100 ms drift around every half hour. In many cases such deviation is unacceptable, and that is one of the reasons that time synchronization protocols are necessary.

2.1.1 An Overview of Existing Time Synchronization Protocols Designed for WSNs

Many time synchronization protocols appeared over time. In the following we describe some of the more important ones for use in wireless sensor networks. Characteristics of such protocols are, in particular, low power consumption and the ability to be used in large-scale networks.

Following the classification in [KW05], we divide the protocols into two groups:

1. Sender/receiver synchronization: the receiver synchronizes to the sender's clock. A widely known protocol like this is the network time protocol (NTP), often used in the Internet [MMBK10].

2. Receiver/receiver synchronization: multiple receivers of the same timestamped packet synchronize with each other, but not with the sender.

Lightweight Time Synchronization Protocol (LTS)

The lightweight time synchronization (LTS) protocol [vGR03] works by using pair-wise synchronization to synchronize the network to reference nodes. LTS can be classified as a sender/receiver synchronization protocol.

There are two major flavors of LTS. Both depend on pair-wise synchronization to synchronize the clock of some node A to that of some node B. Pair-wise synchronization is based mostly on packet time-stamping and is carried out as follows:

1. Node A sends a timestamped synchronization request packet to node B. One should note the propagation delay (caused by the operating system, the MAC layer etc.) and the packet transmission time that is required before the packet reaches node B.

2. Node B constructs an answer packet containing the timestamp of reception (before being processed) and the original timestamp sent by

node A. Before sending the answer packet to node A it is timestamped one more time (after processing).

3. By making the assumption that incurred delays are the same in both directions, node A can estimate its clock offset from the clock of node B. This offset can then be used to synchronize node A to node B.

4. If required node A will send the offset back to node B.

By using pair-wise synchronization, LTS can then synchronize the whole network to a reference node. In case of **centralized multi-hop LTS**, the whole network is synchronized to one reference node, which may be using a high-quality time reference such as GPS. The reference node is responsible for triggering synchronization and all nodes in the network must participate. The synchronization is performed by constructing a spanning tree and let each node perform pair-wise synchronizations with its children. In contrast, one can use **distributed multi-hop LTS**. In this case, there can be multiple reference nodes. Synchronization can be triggered on-demand by any individual node and will synchronize directly to a reference node (can be through multi-hop).

Distributed multi-hop LTS performs better when synchronization of the whole network is not required at the same time. In cases where synchronization of the whole network is required, centralized multi-hop LTS is up to twice more efficient. Distributed multi-hop LTS has, however, the advantage of being able to synchronize just certain parts of the network, and the nodes are able to decide by themselves when to synchronize (on-demand).

Timing-sync Protocol for Sensor Networks (TPSN)

Like LTS, Timing-sync protocol for sensor networks (TPSN) [GKS03] is based on pair-wise synchronization. It can also be classified as a sender/receiver synchronization protocol. In the case of TPSN, there are a few differences concerning pair-wise synchronization:

1. Node A will synchronize to Node B but not vice versa.

2. Time-stamping on the transmission and reception side is done in the MAC layer immediately before transmission or reception. This way, most sources of uncertainty are eliminated - namely OS incurred delays and the medium access delay. The only remaining uncertainty is now the very small delay between time-stamping and actual transmission [KW05].

In contrast to LTS, TPSN requires support from the MAC layer, which is easier to achieve in sensor nodes than with commodity hardware.

By using pair-wise synchronization, TPSN can then synchronize the whole network to a **root node**. Only one root can exist in the network. Similarly to LTS, a spanning tree is built. This is accomplished by sending *level_discovery* packets containing the current level, starting from the root node. At the end of this process, each node knows its level in the tree and its parent node. After a node has found a parent, it periodically resynchronizes to the parent's clock using pair-wise synchronization.

TPSN also defines rules for when a parent of a node becomes inaccessible. If the dead node is not the root node then a *level_request* package is sent to find a new parent. In case the root node dies then a leader election protocol based on contention is initiated.

Reference Broadcast Synchronization (RBS)

The reference broadcast synchronization (RBS) protocol [EGE02] works by estimating the clocks of neighboring nodes. It does not use the concept of global time but rather converts from the local time of one node to another. It can be classified as a receiver/receiver synchronization protocol.

RBS works by having a sender transmit a *pulse packet* into a broadcast channel. The only purpose of this packet is to serve as a sort of common trigger for all the nodes in the *broadcast domain* (nodes who can hear each other), and does not need to contain a timestamp. All receiving nodes times-

tamp this pulse packet and then exchange their timestamps, thus learning about each other's clock offsets.

An important difference compared to previously described approaches is that the nodes do not change their clocks, they just know how to convert them to the clocks of all their neighbors. By keeping a history of this information, a node can approximate not only the clock offset but also the drift for a neighbor node (the speed at which the clock of the neighbor drifts apart from its clock). To approximate the drift, least-square linear regression is proposed in RBS. It has been shown that keeping a history of more than 30 offset observations does not provide any additional gains in precision. Similarly to TPSN, RBS can also benefit in precision from early time-stamping, since it reduces the number of uncertainties.

The pulse senders can either be dedicated nodes or they can be regular nodes acting a both senders and receivers. The former case is especially convenient in networks with stationary nodes like IEEE 802.15.4 in beaconed mode. These dedicated nodes can also compute the offsets and drifts for all the regular nodes.

For multi-hop, a time conversion approach is used. Since the forwarding node can translate from the time of the sending node to the time of the receiving node, it can translate timestamps accordingly before forwarding the packet. When the receiving node receives the packet, it will already be timestamped in its local time.

It is worth mentioning that RBS can produce quite a bit of overhead. In networks that do not have some central node to process offsets and drift, n * (n - 1) packets are required to exchange observations between n nodes. Also, the proposed least-square regression calculation is relatively computationally expensive [MKSL04]. However due to its nature it can be used in cases where event-triggered synchronization is required, with synchronization happening after the event. This may be especially useful when not using a MAC protocol that depends on time synchronization, since the nodes can go into sleep mode for longer periods of time without the need to wait for explicit

resynchronization after waking up. In [MKSL04] authors also point out an advantage of RBS to be the elimination of transmitter-side non-determinism (since any uncertainties on the pulse sender side do not affect RBS), and as a disadvantage the lack of extension to large multi-hop networks.

When setting up RBS, determining the broadcast domains can also provide a challenge. If the broadcast domains are small, more time conversions need to take place which constitutes loss of precision. On the other hand, larger broadcast domains increase the amount of packets that need be exchanged between nodes and the radio power needs to be increased, as well. In either case, each node must be a part of at least one broadcast domain.

Flooding Time Synchronization Protocol (FTSP)

The Flooding time synchronization protocol (FTSP) [MKSL04] is based on the ideas from RBS and TPSN. It is essentially an improvement on RBS and can likewise be classified as a receiver/receiver synchronization protocol.

In contrast to RBS, FTSP assumes a single **synchronization-root** node. For nodes that are not in the broadcast domain of the root (in multi-hop networks), any other already synchronized node can act as the root. Instead of sending simple beacon packets like in RBS, the root sends **synchronization messages** which include a *timestamp* (the estimation of global time), a *root ID* (the root as known to the sending node), and the *sequence number* (incremented after each synchronization round). Since each synchronization round all synchronized nodes will flood the network with synchronization messages, each node maintains a *myRootID* and *highestSeqNum* variable. With the help of this information the node can ignore redundant synchronization messages and synchronize to a single root.

Since FTSP has only a single root node, a mechanism is also proposed to reelect the root node in case of failure of the current root. To detect failure a simple timeout approach is used. After timing out, all nodes declare themselves as the root and begin sending synchronization messages again. If a node receives a synchronization message with a lower root ID than its own

ID, it will give up their root status. Eventually the node with the lowest ID will become the only root in the network. During this process some special measures are also taken to avoid inconsistencies and maintain the synchronized state of the network as much as possible [MKSL04].

Some other improvements of FTSP over RBS are:

1. Compensation for byte alignment: some radio chips cannot capture the byte alignment of the transmitted message stream on the receiver side and the radio stack has to determine the bit offset of the message from the alignment of a known synchronization byte and then shift the message accordingly [MKSL04]. FTSP can compensate for this delay.

2. Time-stamping in the MAC layer: eliminates the jitter of interrupt handling and decoding time.

3. Several jitter reducing techniques to eliminate the send, access, interrupt handling, encoding, decoding and receive time errors.

4. Since nodes need not exchange their observations as in RBS, less network resources are required for FTSP to function.

Hierarchy Referencing Time Synchronization (HRTS)

The Hierarchy referencing time synchronization (HRTS) protocol [DH04] works by synchronizing the broadcast domain, as with RBS, but without the need for neighbors to exchange observations. Like RBS, it can be classified as a receiver/receiver protocol.

HRTS requires one or more dedicated **root nodes** or **base stations**, preferably with an accurate time reference. The protocol is set up so all nodes will synchronize to their closest root node. It works as follows:

1. The root node broadcasts a *sync_begin* packet, which includes, similarly to TPSN, a *level* value and additionally an ID of any neighboring node, let us call it node A.

2. All nodes that receive the sync_begin packet, including node A, timestamp it upon reception.

3. Node A responds to the root node with a packet containing the time of reception, and also timestamps the response before transmission.

4. The root node can now calculate its clock offset compared to node A. This is achieved similarly as with the pair-wise synchronization of LTS and TPSN.

5. The root node broadcasts both its offset and node A's timestamp of reception.

6. Now all nodes in the broadcast domain of the root node can calculate their offsets, without any further interaction with their neighbors (unlike RBS). Node A's offset is simply the offset calculated by the root node. Other nodes can calculate their offsets by taking into account the difference between theirs and node A's local time of reception of the sync_begin packet and adding this relative offset to the offset reported by the root node.

7. This process can be repeated for other levels if the network is multihop. Contention is not necessary as the listening nodes will accept the first received sync_begin sender as their root unless the level value is smaller than the level of their current root.

It is proposed in [DH04] (although not required) to run this protocol over a separate MAC channel to reduce collisions and medium access delays. Through experimentation they also measured the overhead traffic of RBS to be much higher than that of HRTS. HRTS' efficiency does however suffer if there is a very small amount of children per each node. An example would be a line topology.

2.2 MAC Protocols

The Medium Access Control (MAC) protocol layer is a network layer just above the physical layer. Thus, MAC protocols must take into account the properties of the physical layer being used. MAC protocols are used to regulate access to a shared medium, in order to satisfy performance requirements of the target application [KW05].

Because of limited resources, MAC protocols play an important role in WSNs. For some WSN applications, layers above MAC are not even necessary.

2.2.1 An Overview of Existing MAC Protocols Designed for WSNs

Most common categories of MAC protocols that are used in wireless networks are:

1. Protocols with low duty cycles or wakeup concepts.
2. Contention-based protocols.
3. Schedule-based protocols.

Typical performance criteria for MAC protocols are fairness, throughput and delay. In WSNs, however, these criteria are only of minor importance compared to energy efficiency. Besides energy consumption, some other criteria important in WNSs are scalability and robustness against frequent topology changes (due to nodes being replaced or running out of power).

MAC protocols that are particularly well suited to WSNs, and that will be described more closely, are Time Division Multiple Access (TDMA) protocols and Frequency Division Multiple Access (FDMA) protocols. Here, nodes are assigned time or frequency slots for communication with each other. This eliminates collisions and the radio can be switched off outside of the node's assigned slot. It also eliminates overhearing communication between other

nodes, which may be desired in order to not consume power by listening to irrelevant communication. In some MAC protocols, however, overhearing can also be a desired effect - e.g. in Sensor-MAC (S-MAC) and Overhearing Based MAC (OBMAC).

Using TDMA gives rise to new challenges, however. TDMA depends on very tight time synchronization. Nodes in WSNs use cheap hardware which includes components like oscillators and clocks. This means that for TDMA protocols to work reliably, frequent time synchronization is required, which can consume significant amounts of energy. With FDMA, frequency synchronization is necessary, and the ability of a receiver to tune to the channel used by a transmitter - therefore, FDMA transceivers tend to be more complex. Of course, the number of frequency slots is even further limited by the width of the available frequency band.

In this Section, we will provide an overview of MAC protocols that implement solutions to some common problems arising with the use of TDMA or FDMA protocols.

LEACH

The LEACH protocol (Low-energy Adaptive Clustering Hierarchy) [HCI+02] is a TDMA-based MAC protocol and assumes all nodes report their data to a sink node.

It organizes nodes into clusters, and each cluster has a *clusterhead* node, which assigns TDMA slots to *member* nodes in the cluster. Member nodes can only communicate to the clusterhead, but not to each other. The clusterhead can communicate with the sink node, which may take a lot of power if the sink node is far away. In either case the clusterhead will consume more power since it has to be on for more time than the member nodes. Because of this fact, the clusterhead is periodically reelected, so that one node will not run out of power sooner than other nodes. To avoid distortion between neighboring clusters, a clusterhead will also assign a CDMA (Code Division Multiple Access) code to the cluster.

The protocol itself is organized in rounds, which are divided into phases:

1. The setup phase: Nodes self-elect themselves as clusterheads, based on how long ago they were last a clusterhead.

   The advertisement phase: The clusterheads inform their neighborhood about their clusterhead status. Each node will pick a clusterhead based on signal strength.

   The cluster-setup phase: Nodes inform their clusterhead that they want to join the cluster.

   The broadcast schedule phase: The clusterhead builds a TDMA schedule and a CDMA code for the cluster, and broadcasts it to the members.

2. The steady-state phase: Nodes can communicate with their clusterheads according to their TDMA schedule.

Because according to protocol specification, a clusterhead will communicate directly to the sink node, a problem for LEACH would be if the sink node is very far away from a clusterhead, which can happen in bigger WSNs. This could be solved by adding forwarding. Another problem with LEACH is the requirement that nodes can only communicate to the clusterhead and the sink node, which is not always enough.

HMAC

The HMAC (Hash MAC) protocol [Ril10], developed at Luleå University of Technology, is a protocol based on TDMA and FDMA. It relies on global time synchronization to work reliably.

The protocol is organized into epochs, each being divided into phases:

1. The start phase: The purpose of this phase is to let nodes power up their radios and tune to the broadcast channel.

2. The synchronization phase: This phase does not need to happen every epoch. In this phase, the nodes synchronize to each other, so they can properly follow their TDMA schedule.

3. The broadcast phase: In this phase, nodes that wish to send something announce it with an RTS (request to send) packet. There is no acknowledgement or clear to send packet required. Each node has a time slot inside the broadcast phase when it can send an RTS.

4. The unicast phase: Process received RTS to schedule receive and/or schedule send if the node has any data to send.

   The unicast receive phase: The node expects another node to transmit during this phase.

   The unicast send phase: The node transmits data.

Except for synchronization purposes, there are no central nodes in the network. The nodes can communicate peer-to-peer. Their schedules are calculated based on a hash function and a unique identifier of each node (e.g. TOS_NODE_ID), which determines both the time at which to send as well as the channel to use. All nodes are required to have their radio on only in the broadcast phase. If a node does not have anything to send or receive, it can power its radio off during the other phases.

Each of the above phases will take a set amount of time. In the following table we present some durations that were used in [Ril10]:

	Application 1	Application 2
Unicast slot	3 ms	24 ms
Unicast frame	60 ms	1752 ms
Broadcast frame	0	5000 ms
Start frame	100 ms	100 ms
Super frame	160 ms	6852 ms

Table 2.1: HMAC parameters for two applications.

By tuning the length of the frames, HMAC can be adapted to meet the requirements of specific applications. For example, in Application 1 a line topology was used, therefore the broadcast frame was not necessary.

The fixed unicast slot length determines the maximum transfer unit. The amount of time that a node needs to be awake each super frame can be determined by adding the durations of the start (to prepare radio) and broadcast frame (to listen for incoming RTS packets). It can also be noted that these times will determine how much delay will be experienced when sending. For example, with Application 1, a send delay of up to 160 milliseconds (duration of one super frame) can be experienced.

2.3 Hardware platforms

There are two platforms of interest to this thesis:

1. The mulle platform.
2. The mulleiroad platform.

The mulle platform, an educational platform, has been in use for a longer time and support for it is somewhat more extensive. The mulleiroad platform was designed as a proof of concept for a production-value application. Since both were designed at LTU, it will be interesting to compare their performance in the following experiments.

Chapter 3

Work and Results

3.1 Time Synchronization

3.1.1 Measuring clock drift on target platforms

As a basis for more exhaustive experiments, it is beneficial to determine the average clock drift on target platforms. We performed the experiment under two regimes of operation:

1. Full power operation: the devices were fully powered throughout the experiment and were not put to sleep.

2. Low power operation: the devices were put into sleep mode periodically. This kind of operation is more commonly encountered in production-value applications.

The assumption is that clocks will be more stable in the former of the two regimes. The experiments were ran for 30 minutes. The expected result is a drift between 1 - 100 ppm. We base these expectations on [vGR03, EGE02].

Results

The results were obtained by letting two nodes run for 30 minutes and measuring their local time at the start and end of the experiment, in millisecond

precision. The drift in ppm (i.e. drift in microseconds per second) was calculated using

$$\text{drift} = \frac{(t_e - t_s) \cdot 1000}{30 \cdot 60} \tag{3.1}$$

where

Parameter	Meaning
drift	Clock drift estimation in parts per million
t_s	Time in milliseconds when the experiment was started
t_e	Time in milliseconds when the experiment was ended

Table 3.1: Parameters used in (3.1).

Drift	Mulle	Mulleiroad
Idle application	800 ppm	0
FTSP application	800 ppm	3000ppm

Table 3.2: Measured clock drift on our platforms.

The results were similar for full power and low power modes, but seem to depend more on other factors like the load of the CPU, especially on mulleiroad (see Table 3.2).

For mulleiroad, we did not measure any drift when the MCU was not under load (idle application). But when observing drift in a FTSP test application, the drift would increase and was measured to be around 3000 ppm.

These somewhat unexpected results may be attributed to the fact that mulleiroad is using the internal clock on the MCU chip. Due to this fact, higher MCU load can affect the internal clock operation. It may also be attributed to implementation details in TinyOS.

The results for mulle are surprising as well, as the drift is much higher than in [vGR03, EGE02]. This may be attributed to cheaper clock crystals being

used. Mulle uses an external clock, but as with mulleiroad, implementation details in TinyOS may also have an impact on the results.

It is also worth noting that [vGR03, EGE02] were published in 2002 and 2003, respectively, while the hardware we used was produced more recently.

3.1.2 Synchronization with FTSP (reference implementation)

An existing FTSP (see Section 2.1.1) test application was used to measure how well FTSP performs on our target platforms. We used the following configuration:

1. Unmodified FTSP reference implementation available in TinyOS.
2. A beaconing node running RadioCountToLeds. The only purpose of this node is to beacon to FTSP nodes.
3. Nodes running FTSP to synchronize their global times. 8 such nodes were used for mulle and 2 such nodes were used for mulleiroad.
4. A base station node running BaseStation to receive global times from FTSP nodes and relay them to a PC for analysis.
5. The FTSP root sends out synchronization messages every 10 seconds.

Results

The motes ran for around 3 hours. Results are represented graphically in Figure 3.1 and Figure 3.2, separately for each platform. Each Figure represents time synchronization errors - what is the difference between a node's global time estimation and the actual global time (commonly the local time of the FTSP root node).

Our results show errors higher than those observed in [MKSL04]. Their time synchronization errors were in the range of a few microseconds, while our results show errors in the range of a few milliseconds. This could be

explained by several reasons. First, the authors used 32 KHz and even microsecond precision in their work, while we used millisecond precision, due to lack of software implementation supporting 32 KHz FTSP. Also, we are using different platforms than the authors, but we are using their reference implementation, which could account for errors due to assumptions made by the authors regarding hardware and implementation. Time-stamping is also performed at different times in our case, depending on driver implementations on our platforms (e.g. on mulle, its slow SPI - Serial Peripheral Interface bus somewhat limits options where time-stamping can be performed). It was also mentioned in [YL10] that in low-power modes, FTSP can result in errors of several hundred milliseconds. On mulleiroad, stop mode operation, which makes the MCU go to sleep when idle, is enabled by default in hardware configuration code.

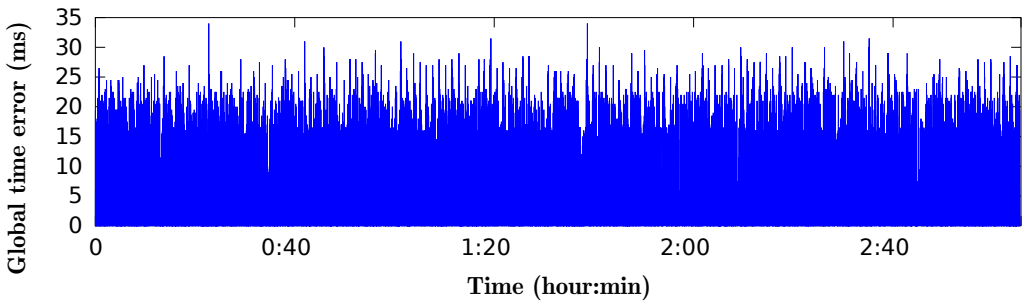

Figure 3.1: Mulle FTSP test results

Mulle

From Figure 3.1, we can observe that the difference between global time estimations was as high as 35 milliseconds. Resynchronization was every 10 seconds. Comparing with measurements from Section 3.1.1, there is a 35 ms error compared to 8 ms error due to clock drift in 10 seconds. However after 3 hours, clock drift would account for an error of 8640 ms, while the error of around 35 ms will be constant when using FTSP, as long as nodes can periodically resynchronize. After 50 seconds of operation, clocks synchronized with FTSP will already be more accurate than without synchronization. Based on this results, a feasible FTSP resynchronization period for mulle would be around 30 seconds (clock drift on mulle in 30 seconds is around $3 \cdot 8ms = 24ms$, which is still lower than the 35 ms FTSP synchronization error).

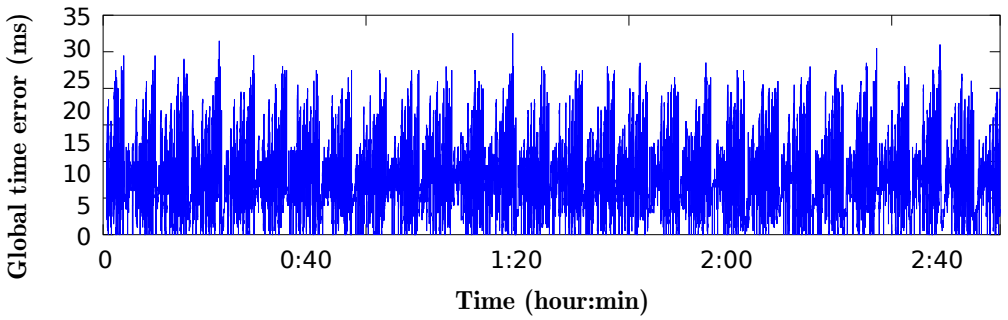

Figure 3.2: Mulleiroad FTSP test results

Mulleiroad

Figure 3.2 shows the time synchronization error is lower than with mulle, only up to 25 milliseconds. We observed varying drift depending on MCU load in Section 3.1.1, thus when FTSP will pay off depends more on the target application, more accurately on the MCU load (due to clock drift, see Table 3.2).

On both platforms, one should also take into account that our experiments were performed in a controlled environment. The clock drift in harsh weather conditions may be higher than what we measured. As long as there is clock drift, any kind of synchronization will pay off eventually.

3.2 Improved HMAC

3.2.1 Redesigning the HMAC protocol implementation

An important part of this thesis is a complete redesign of the HMAC protocol implementation in TinyOS. A reference implementation was available to us, however it was written in a proof-of-concept manner. It soon proved tedious to implement and test different time synchronization techniques using this code base. This was due to low modularity of the design, low code manage-

ability and several bugs. The code was also very tightly coupled and it was therefore even more difficult to switch between synchronization techniques. We decided to completely redesign the implementation and re-implement HMAC for TinyOS.

The reference HMAC implementation was composed of 5 main components, the most important being a scheduler component. In our work, some parts of the reference implementation were reused, e.g. the radio control state machine.

A design decision was made to base our own design on the design of existing networking layers in TinyOS, especially the partial implementation of the IEEE 802.15.4 protocol (referred to as Ieee154 in the code, a notation we will use from here on as well). This partial implementation is split into two components or sublayers - the packet component and the message component. The packet component provides commands to perform operations on packets (i.e. to manipulate the Ieee154 header), and provides packet-related interfaces to upper layers (e.g. to ActiveMessage). The message component provides the actual implementation of the layer — handling sending and receiving messages.

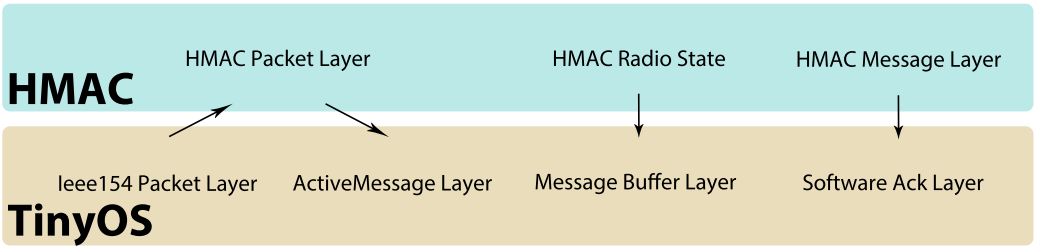

Figure 3.3: Redesigned HMAC.

Due to complexity arising from the TDMA/FDMA nature of HMAC, we divided the code into several components to achieve better modularity. The components are presented graphically in Figure 3.3. An arrow signifies that the component from which the arrow is pointing is being used by the other component. Ieee154PacketLayerC is the layer below HMAC. ActiveMessage-

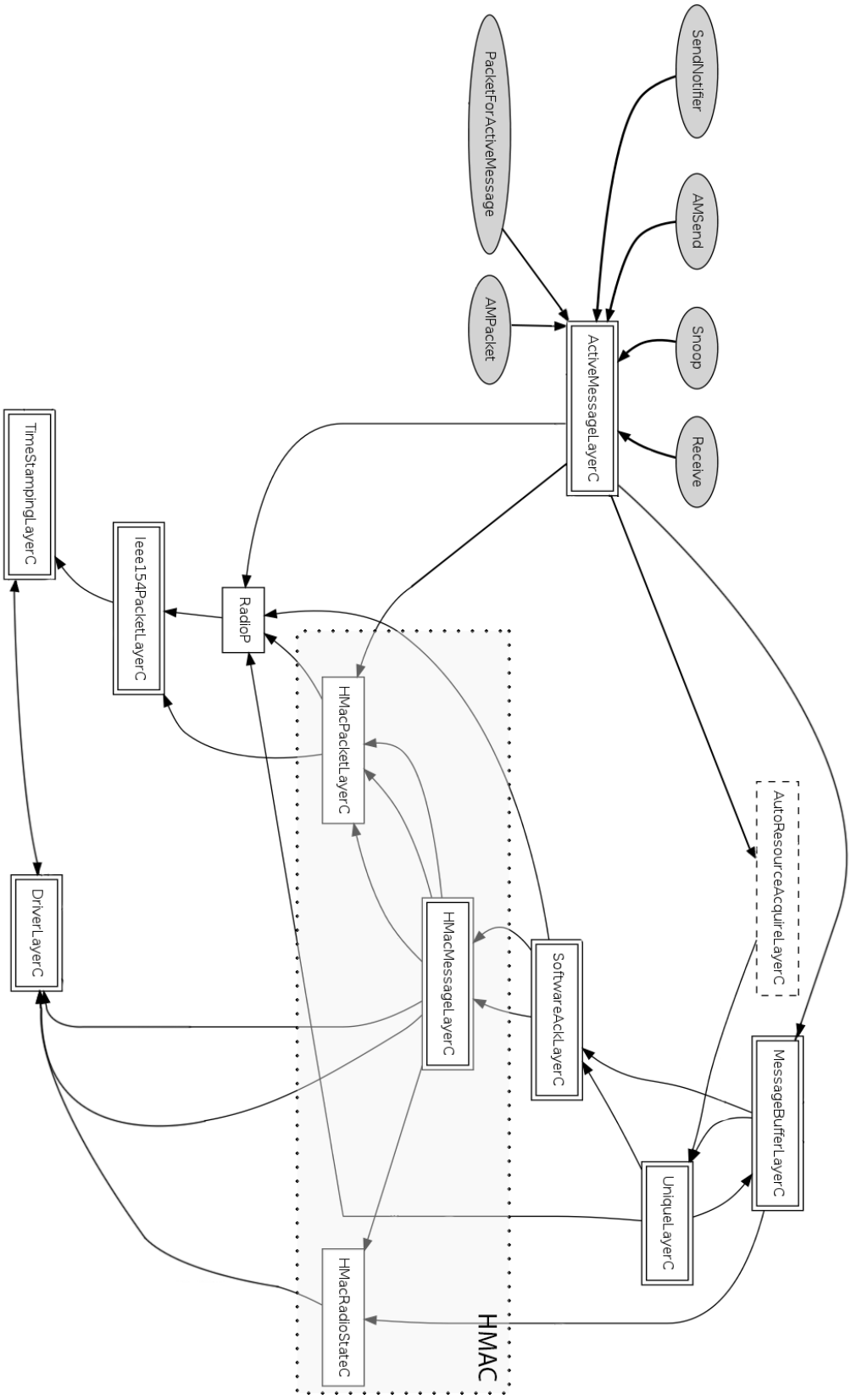

Figure 3.4: Radio components with the new HMAC.

LayerC uses RadioPacket from HMacPacketLayerC, which adds the HMAC packet header. MessageBufferLayerC uses RadioState provided by HMacRadioStateC and passes it on to upper layers. Instead of manipulating the actual radio state, it toggles the HMAC scheduler on/off state. SoftwareAckLayerC uses (and provides to upper layers) the RadioSend and RadioReceive interfaces implemented in HMacMessageLayerC. Some of HMAC components are also interconnected, but this has been left out for clarity. A more detailed and accurate view of the components and their connections is shown in Figure 3.4.

The purpose of these components is described in the following.

1. HashC and HSchedulerQueueC: Essentially reused from the reference implementation. HashC provides the hash function that is used to determine time slots and channel used for communication. HSchedulerQueueC provides a job queue that is used by the scheduler.

2. HMacSlotManagementC: Provides commands for determining time slots and channels for communications, based on HashC.

3. HSchedulerC: Only includes the actual scheduler and things that could not be separated from it. Actual packet sending and manipulation is not done here, it only tracks which phase the scheduler is currently in, based only on time.

4. HMacRadioStateC: This was refactored out of HMacLayerC in the reference implementation. It is a state machine that carries out basic operations on the radio state, such as switching on/off and changing the channel. It also provides a radio state interface for the upper layers, which was redesigned so that when an upper layer wishes to switch on/off the radio, the scheduler is started/stopped instead, but all actual radio state manipulation is performed by the HMAC layer, i.e. the upper layers do not have direct access to the radio state anymore. The scheduler will switch the radio on/off automatically to conserve power.

Because HMAC is FDMA based, upper layers also cannot change the channel the radio is tuned to.

5. HMacPacketLayerC: Provides everything related to packet manipulation. We added a new HMAC header (Figure 3.5) to packets, which contains a type (either an internal packet like request to send - RTS, SYNC, or a user message passed to upper layers), something that was not done in the reference implementation (ActiveMessage was used there). It also provides commands for manipulating the RTS and SYNC messages. Figure 3.6 shows the structure of the RTS message, while the SYNC message is described in more detail in the next section.

6. HMacMessageLayerC: Handles only sending and receiving messages. It provides a send command for upper layers and can signal upper layers when a new user message is received. It works with the scheduler to handle the receipt of internal messages like RTS and SYNC, and provides commands to send these messages, as well.

7. There is also an interface that is implemented in the radio driver called HMacConfig, which only includes helper commands to determine source and destination addresses from the layer below HMAC. If desired, these addresses could now very simply be included in the HMAC header and some intermediate layers could be removed. There are also some other minor additions like a few new interfaces and header files.

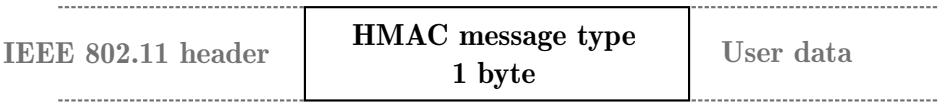

Figure 3.5: The newly added HMAC header.

The result is not only a completely redesigned implementation of HMAC but also several bug fixes and improvements. The most important new features that were added are an actual HMAC header, removing all references

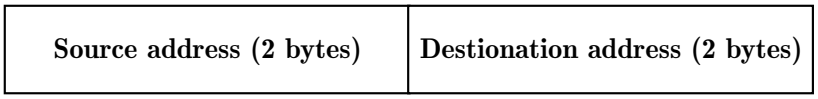

Figure 3.6: HMAC request to send (RTS) message.

to ActiveMessage, proper radio state control and proper wiring into upper layers (Packet and RadioState, in particular). The code was also decoupled so it is now much easier to modify and add new features such as new synchronization techniques.

3.2.2 Simple time synchronization

A simple way to synchronize nodes is to send a synchronization message (Figure 3.7) with the epoch number from one node, and other nodes set

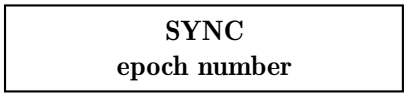

Figure 3.7: Simple SYNC message.

their epoch to this number. After that, all of the nodes should enter the same phase (e.g. the broadcast phase). This solution works, but only in networks where all nodes can hear each other. For multi-hop networks, this approach can cause interference between broadcast domains. Since this technique does not account for clock drift it is also required to resynchronize more regularly than e.g. with FTSP.

There is also the question of what to do after the synchronization messages are processed. One possibility is to immediately start normal operation by proceeding into the broadcast phase, losing less time due to synchronization. A problem with this approach can be that the current super frame duration is slightly longer due to the added the extra synchronization phase, which is normally not present. To solve this issue, we could wait until the end of

the current super frame before resuming normal operation. The drawback of such an approach is that more time is lost for synchronization.

This kind of approach was used in the reference implementation, and before moving to a more complex synchronization technique, we implemented this technique first. Since only one synchronization packet is needed at a time, memory for it was allocated during initialization and kept reference to it in the HMAC packet layer. HMAC message layer provides commands and events for sending and receiving the SYNC messages. The scheduler uses these commands and events to perform synchronization according to HMAC configuration (e.g. how often to synchronize).

Using this synchronization approach, we were able to test if our HMAC implementation works properly. We ran a ping pong application with 2 nodes, and after 1 hour it was still working properly, which means that the implementation is usable.

3.2.3 FTSP time synchronization

The goal was to integrate FTSP into HMAC, so that it would be transparent to the user, while still providing them global time if desired. Internally, global time should be used to ensure reliable function of HMAC. An ideal solution would be to rewire FTSP components so that they would use HMAC. In this way, it would be much easier to use possible newer versions of FTSP later, without having to update all the patches.

First, components used in FTSP and their dependencies were identified. Components TimeSyncC and TimeSyncP only use TimeSyncMessageC from the network-related dependencies. This is beneficial as it means that an alternative implementation needs to be provided only for this component, and TimeSyncP need not be touched. The current implementation allows the user to trigger sending of a synchronization message, which we used to send these messages based on HMAC configuration and which phase the HMAC scheduler is in.

TimeSyncMessageC (more specifically, RF212TimeSyncMessageC in our

case) mostly rewires to TimeSyncMessageLayerC, however it also uses RF212ActiveMessageC. Avoiding ActiveMessage-related dependencies was one of the design goals. However, only things that come directly from a lower layer, TimeStampingLayerC, are being used from RF212ActiveMessageC. This layer is below HMAC in the network stack, thus it can be used directly.

The last component significant component is TimeSyncMessageLayerC. Internally, it uses ActiveMessageC and AMSenderC. Because ActiveMessage-related interfaces are also used in the implementation, and the implementation is trivial, we made a decision to rewrite this component completely.

Based on this initial research, we assume that we will need to provide a new TimeSyncMessageC and TimeSyncMessageLayerC, which should only use layers below the HMAC layer. Additionally, we have to determine when to trigger sending of FTSP messages and make HMAC use the global time. Because HMAC hash functions take the current epoch as a parameter, nodes must also all use the same epoch number. Since we need to modify TimeSyncMessageC, it would be a good idea to include this information therein.

Results

During the implementation process, we realized that using the millisecond precision FTSP introduces many errors. First, the packet time-stamping done in the radio driver is done with microsecond precision. Second, we use a microsecond precision alarm in the HMAC scheduler. Using millisecond precision FTSP thus means we need to perform time conversions quite frequently. FTSP approximations are also slightly better when using microsecond precision, however we did not measure a substantial difference, since approximation errors can be in the tens of milliseconds range.

Based on these findings, we moved to microsecond precision. To accomplish this, we used the CounterMicro32C counter, available for the mulleiroad platform. We are already using this same counter in the HMAC scheduler, thus we do not need to convert time values, leading to less processing time

and better precision.

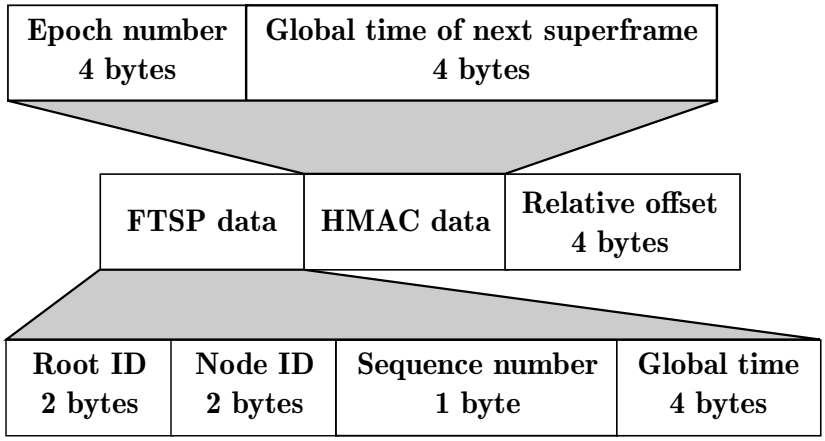

Figure 3.8: FTSP SYNC message with added HMAC data.

We were able to remove all ActiveMessage dependencies from FTSP, and instead use the HMAC layer and the time-stamping layer. Additionally, we incorporated the current HMAC epoch number and the global time of the next super frame into the FTSP synchronization message. The epoch number is vital to the proper function of HMAC and must be synchronized. To give the nodes enough time to process FTSP data, we wait for the next super frame after sending the FTSP message, instead of proceeding with normal operation immediately. For this reason, the global time of the next super frame is included in the FTSP message. The FTSP message with the HMAC-related information is represented graphically in Figure 3.8. The FTSP data part of the header is described in more detail in [MKSL04]. The relative offset at the end of the message is appended by the radio driver and is the delay between the packet send request and actual start of transmission.

FTSP parameters

During testing, we observed very poor performance. One of the causes was the throwout limit parameter of FTSP. Upon receiving a new synchronization point, FTSP compares the received global time to its prediction based on previous data. If the difference between the prediction and the actual value

is larger than the throwout limit, all synchronization points are removed — essentially, FTSP resets. This parameter does not have an implicit unit and is set to 500 by default. When using millisecond unit, the limit is therefore 500 milliseconds, which is quite large, but when using microsecond unit, the limit is 500 microseconds. In our case, errors between FTSP predictions and actual global times were in the range of a few milliseconds, up to 60 milliseconds. Therefore, FTSP was resetting very frequently, resulting in a desynchronized state. After adjusting this parameter to a more reasonable value (in relation to our error measurements) of 60,000 microseconds, performance improved substantially according to initial testing.

We also noticed that the resynchronization period of FTSP has an impact on the quality of the predictions. We found too frequent synchronization to be detrimental to prediction quality. The reason is the limited size of the FTSP regression table, which is 8 entries by default. If the synchronization points are closer together, the drift prediction is less accurate. We found a feasible setting is to synchronize around every 10 seconds, which is also the default setting in FTSP. For the new implementation, we need to express this parameter in the number of HMAC super frames between each synchronization. We used 950 ms super frames, therefore we set resynchronization to every 12 super frames. When changing the super frame length, it is important to adjust this parameter accordingly. Also note that FTSP requires at least 4 synchronization points to become synchronized by default. This directly affects how much time a new node added to the network will need before it becomes synchronized (40-50 seconds if resynchronization is every 10 seconds).

Just before sending a FTSP message, the radio driver adds the time elapsed between the message send request and the actual transmission start to the message. In the reference implementation, the receiver adds this relative value to its local time and not the received global time. We observed that this relative time can be high enough that when coupled with high clock drift, this way of calculation can cause a noticeable error in predictions. We

modified the implementation so that this relative value is added to the received global time instead of the local time.

Radio driver related issues

Even after these changes, FTSP was not giving satisfactory results. We confirmed that FTSP linear regression calculation is working properly by performing the calculation manually on the same dataset. The reason for bad FTSP results was therefore narrowed down to the reported global time values. There are two parts to a global time report - the global timestamp and the delay between that timestamp and the time of radio transmission. The latter is calculated and appended to the FTSP message by the radio driver (an important feature of FTSP). Observing the global timestamp did not give any indication of error. However, when observing the delay value, it was negative on occasion. By debugging the radio driver layer, we discovered that the delay calculation was flawed. After fixing this bug, we observed a significant improvement in FTSP global time prediction.

While the performance did improve substantially with the above correction, the predictions were still occasionally off by a large margin. We managed to trace this back to a yet another bug in the radio driver, this time in the radio reception code. Again, the incoming transmission was timestamped in a similar fashion as with transmission, resulting in an erroneous timestamp with an occasional error of around 65 milliseconds.

After correcting this problem, along with the other modifications, FTSP performance improved dramatically, moving into microsecond range. Figure 3.9 shows the global time error measurements. The maximum error was 8 microseconds, the median error was 0 and the average error was 0.5 microseconds.

Initial synchronization

Another issue that should be addressed is how the nodes should synchronize when they are first connected to the network. Let us consider the following options:

1. Begin sending FTSP messages to determine the root node and then

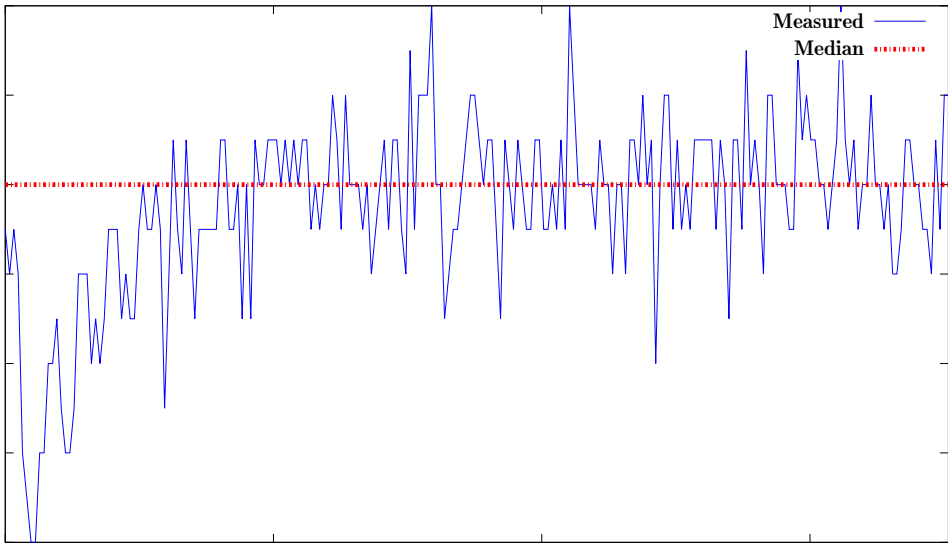

Figure 3.9: FTSP average global time error measured over 30 minutes.

synchronize to it, or become root.

2. Statically pick the root node before-hand. All other nodes wait for a FTSP message from the root node and synchronize to it, before sending any messages into the network.

The problem with the first option is that new (or desynchronized) nodes will disrupt the network and may cause collisions. The second approach resolves this issue, but does not allow for dynamically electing the root node. This means that if the assigned root node is or becomes unavailable, the whole network will stop working.

We propose (and implemented) the first option, but using a separate radio channel for synchronization. This way, FTSP can perform its dynamic root election without disrupting the network.

3.2.4 Multi-hop test

We used our improved HMAC with simple static routing to determine performance in a multi-hop network. We wanted to measure the latency and the packet-loss rate. The test application sends a message with a counter value to the next node. The other nodes relay the packet to the next node using static routing.

Radio snooping is not possible in HMAC, due to channel hopping features of the protocol and the fact that there can be more than one transmission in progress at the same time on different channels. Therefore, we included global times of reception for all the nodes in the message that was relayed through the network. At the sink node, we could therefore see the latency per hop.

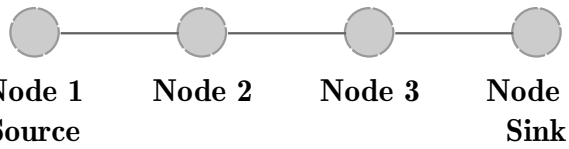

Figure 3.10: An example of a line network topology.

The test was performed as follows:

1. FTSP resynchronization every 10 seconds.
2. HMAC super frame length of 100 milliseconds.
3. The network was composed of 5 nodes in a line topology (c.f. Figure 3.10).
4. The source node sends a message every 3 seconds.
5. The test was ran for approximately 30 minutes with 604 packets sent.

Results

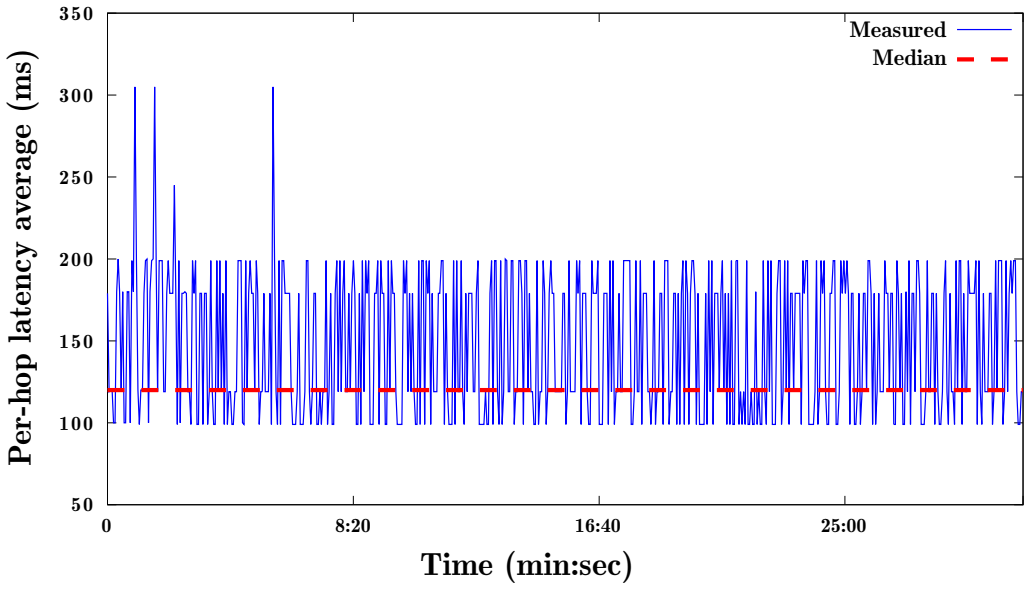

Figure 3.11: Per-hop latency measured over 30 minutes.

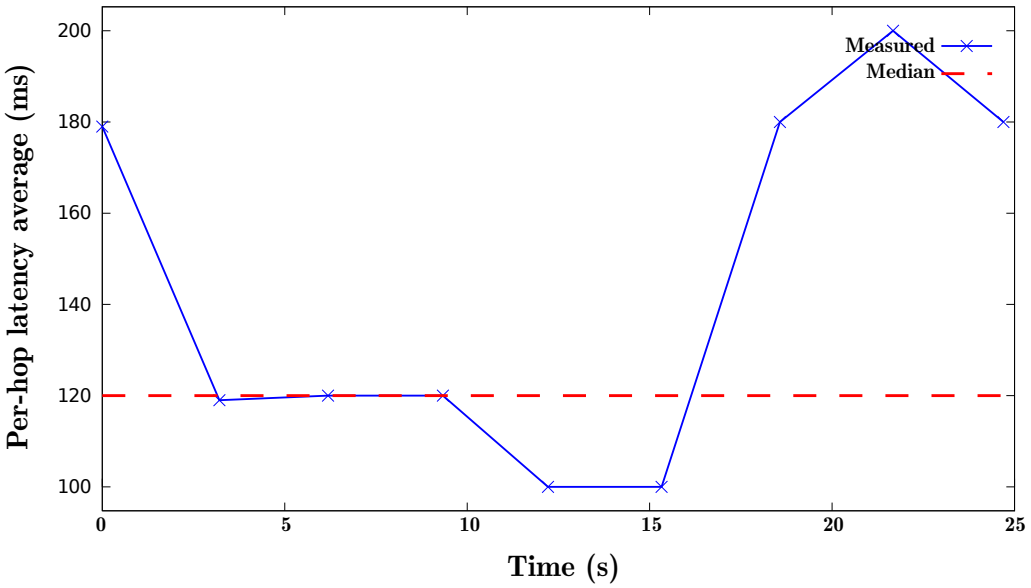

Figure 3.12: Per-hop latency in detail over 30 seconds.

We observed a per-hop latency between 99 and 305 milliseconds. The results are presented graphically in Figure 3.11. Figure 3.12 shows the results for the first 30 seconds in more detail.

The main factor that affects the latency is the length of the HMAC super frame. When a send is requested, it cannot be processed before the current super frame ends. With the used HMAC parameters, the unicast phase of HMAC in which the message is sent comes 60 ms after the start of the super frame, and an unicast slot is 20 ms long. The lowest measured latency was 99 ms, which is around the length of one super frame. The **median latency was 120 ms**, and the **average was 149 ms**.

Higher latency (spikes in Figure 3.11) can occur due to several reasons:

1. FTSP synchronization takes up a whole super frame every 10 seconds, increasing the latency by one super frame for messages sent during this time.

2. Send failures and collisions cause a delay. If the failure happens during RTS transmission, our HMAC implementation will automatically resend the message in the next super frame - the added delay is the length of one super frame. However, if the failure occurs when sending the user data (in the unicast frame), sending fails and is signaled back to the user. The user needs to submit a new send request, which can take an arbitrary amount of time.

The median latency for a message to arrive from the source node to the sink node was 480 ms for the 5 node network. The result is reasonable as it is 4 times the median per-hop latency, and there are 4 hops between the source and sink node. There were 604 packets sent in total, and no packets were lost.

3.2.5 Power consumption in a line network topology

In case of a line topology in a network with a source node, a sink node, and intermediate nodes relaying the packet from the source to the sink, we can

optimize HMAC further. In this case, a node will only receive packets from the previous node, and only send packets to the next node in the network.

To provide better support for such cases, we introduced the option to remove the broadcast frame. In this case, nodes use a single unicast slot for communication, and when a node can send or should listen for incoming packets is determined by its position in the network and the current HMAC epoch number.

Super frame type	Node 1	Node 2	Node 3	Node 4
Epoch #1	SEND	RECV	SEND	RECV
Epoch #2	RECV	SEND	RECV	SEND
Epoch #3	SEND	RECV	SEND	RECV
Epoch #4	RECV	SEND	RECV	SEND

Table 3.3: An example of the adapted HMAC protocol.

The benefit of this approach is that less network traffic is generated, since HMAC RTS packets need not be exchanged. Additionally, all nodes must be listening during the broadcast frame, thus removing the broadcast frame leads to lower power consumption, as well. In this adapted version of HMAC, we denote a super frame in which a node can send a *send super frame* and a super frame in which a node must listen a *receive super frame*. In case a node has nothing to send, it can power off its radio during the entire duration of the send super frame, which is beneficial in terms of power consumption.

To avoid collisions, FDMA (different radio channels) can be used. Since only the source node generates new packets, collisions can also be avoided by imposing a limit on how often the source node can send a new packet, depending on how long it takes for the packet to propagate to the sink node.

We evaluated the power consumption of this adapted version of the HMAC protocol in comparison to normal HMAC described in Section 3.2. The results are presented in Figure 3.13. The HMAC protocol adapted for line topology lowered power consumption considerably compared to the normal

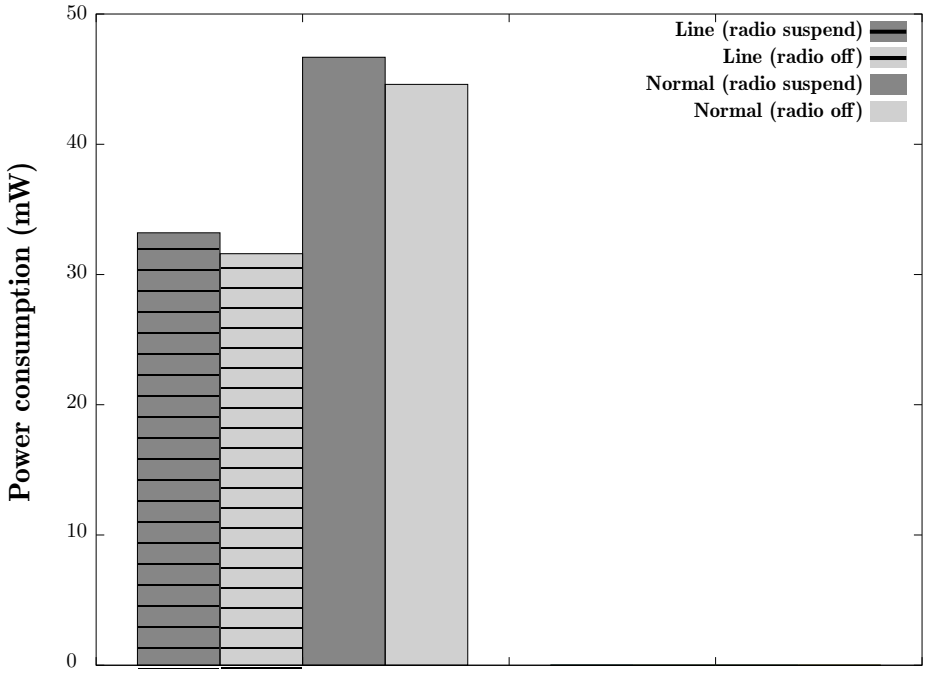

Figure 3.13: Power consumption with the normal HMAC protocol and the version adapted for line topology.

HMAC protocol. Power consumption was further reduced if radio was powered off outside allocated time slots, instead of putting the radio into suspend mode. The lowest power consumption was achieved by using the adapted HMAC protocol for line topologies, and periodically turning the radio on and off. In this mode, the device was consuming around 31 mW of power on average.

3.3 Some additional work-related challenges

During our work there were challenges that we had to overcome in order to achieve set goals. We will describe some of these challenges briefly in the following.

Serial communication problems on mulleiroad

There was a problem with mulleiroad's serial communication, where it was not possible to receive data from the device on the PC. This problem was related to the programmer chip, which required a DTR (Data Terminal Ready) control signal to be set properly before it could receive data. Because we relied on serial communication for debugging purposes, this was very important to solve. We had a program called *mulle_term* available that could communicate with the device, but there was no documentation nor source code available. This program did not suffice for all our needs, so it was necessary to find a better solution.

First, we tried piping the output of mulle_term into java applications that were normally used for communication with TinyOS devices. However, this did not work, because the program left out certain data which was important for these applications. Using the *strace* program on Linux, we were then able to reverse-engineer mulle_term to figure out how it works. By doing this we found out about the DTR control signal, and after that we could write our own software to communicate with the device.

Low power listening on mulle

In order to obtain more FTSP performance measurements for comparison, we wanted to run the low power FTSP test on mulle. We used the reference implementation that comes with TinyOS, but it did not work. By debugging the program inside the drivers themselves, we were able to determine that the packets were never being received by the radio. We were not able to determine exactly why this was the case. We did ask the maintainers if they could look into it, but it was not fixed in time for us to conduct the experiment for this thesis.

Permanent stop mode on mulleiroad

On mulleiroad, stop mode was forcibly enabled in the hardware configuration file, *hardware.h*. Stop mode makes it so the device will enter sleep mode when there is nothing to be done. It was not critical for us to solve this, but it should be noted that this may have affected some of our results in the other tests.

Chapter 4

Conclusion and Future Work

We were able to fix critical driver bugs to enable correct function of FTSP with our hardware. We observed around 10.000 times better FTSP performance with these modifications. We improved HMAC by adding a HMAC packet header, redesigning the code base for better code manageability, and improved performance and reliability. Integrating FTSP time synchronization into HMAC also improved its function substantially. Even with these changes, there is still room for improvement. In the following, we list some possible improvements and current issues we encountered:

1. Debugging with radio snooping in development. As a security feature of HMAC, it is not possible to snoop radio messages (listen for all messages being sent in the network, regardless of the recipient). This is due to the fact that in HMAC, two transmissions can occur at the same time but on different radio channels. This kind of operation would be beneficial for research and debugging purposes. It can be achieved by disabling channel hopping and if necessary, modifying the hash function or increasing the number of unicast slots in order to avoid too many collisions in the time domain.

2. Dynamic broadcast channel. With the introduction of the separate synchronization channel, the broadcast channel can change dynamically.

Since the epoch is synchronized through the synchronization channel, the broadcast channel can depend on the epoch, similarly to unicast channels. This would provide some additional security to the protocol.

3. Multiple transmissions in one super frame. The design of HMAC allows for multiple transmissions in one super frame, if the recipients are not the same. By using a send buffer, this would be relatively easy to implement. This kind of operation would be beneficial for applications that need to frequently send messages to multiple recipients.

4. Collisions. With the current hash function used in HMAC to assign unicast time slots, collisions are possible. We experienced collisions on several occasions, which caused unwanted delays and also had negative effects on reliability. We used the same hash function as was used in the reference HMAC implementation. It is a simplified version of an existing hash algorithm, in order to decrease computational complexity. With the original hash function, collisions would be less frequent, but computational complexity would be too great.

5. Radio sleep during unicast. We already put the radio to sleep in the unicast frame after all unicast jobs have been processed. However, the radio could be put to sleep during unused unicast slots, as well.

Bibliography

[DH04] Hui Dai and Richard Han. Tsync: a lightweight bidirectional time synchronization service for wireless sensor networks. *SIGMOBILE Mob. Comput. Commun. Rev.*, 8:125–139, January 2004.

[EGE02] Jeremy Elson, Lewis Girod, and Deborah Estrin. Fine-grained network time synchronization using reference broadcasts. *SIGOPS Oper. Syst. Rev.*, 36:147–163, December 2002.

[GKS03] Saurabh Ganeriwal, Ram Kumar, and Mani B. Srivastava. Timing-sync protocol for sensor networks. In *Proceedings of the 1st international conference on Embedded networked sensor systems*, SenSys '03, pages 138–149, New York, NY, USA, 2003. ACM.

[HCI+02] Wendi B. Heinzelman, Anantha P. Ch, IEEE, Anantha P. Chandrakasan, Member, Hari Balakrishnan, , and Hari Balakrishnan. An application-specific protocol architecture for wireless microsensor networks. *IEEE Transactions on Wireless Communications*, 1:660–670, 2002.

[KW05] Holger Karl and Andreas Willig. *Protocols and Architectures for Wireless Sensor Networks*. John Wiley & Sons, 2005.

[MKSL04] Miklós Maróti, Branislav Kusy, Gyula Simon, and Ákos Lédeczi. The flooding time synchronization protocol. In *Proceedings of*

the 2nd international conference on Embedded networked sensor systems, SenSys '04, pages 39–49, New York, NY, USA, 2004. ACM.

[MMBK10] D. Mills, J. Martin, J. Burbank, and W. Kasch. Network Time Protocol Version 4: Protocol and Algorithms Specification. RFC 5905 (Proposed Standard), June 2010.

[Ril10] Laurynas Riliskis. *On the design of dependable communication protocols for wireless sensor networks.* Licentiate thesis, Luleå University of Technology, Luleå, Sweden, 2010.

[vGR03] Jana van Greunen and Jan Rabaey. Lightweight time synchronization for sensor networks. In *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, WSNA '03, pages 11–19, New York, NY, USA, 2003. ACM.

[YL10] Jue Yang and Xinrong Li. Design and implementation of low-power wireless sensor networks for environmental monitoring. In *WCNIS*, pages 593–597, 2010.