On Update Protocols in Wireless Sensor Networks

Tobias Schwindl, Klaus Volbert, and Maximilian Schwab(✉)

Ostbayerische Technische Hochschule Regensburg, Prüfeninger Straße 58, 93059 Regensburg, Germany
{tobias2.schwindl,klaus.volbert,maximilian1.schwab}@oth-regensburg.de

Abstract. There has been a lot of research been done in the domain of Wireless Sensor Networks in recent years. Nowadays, Wireless Sensor Networks are in operation in a wide range of different scenarios and applications, like energy management services, heat and water billing as well as smoke detectors. However, research and development will be continued in this domain. During the operation of such a network, software updates need to be done seldom. In contrast to this, software updates need to be done very frequently during development and testing for uploading a new firmware on umpteen nodes. In this paper, we examine such a software update for a particular, but popular and often used sensor network platform. There are already interesting research papers about the process of updating sensor nodes. Our specific focus relies on the technical part of such an update process. We will argue why these already existing update processes do not cover our defiances. The objective of our software update protocol is to enable the developer to update many nodes in a reliable and very fast fashion during the development and testing process. For this reason, energy consumption is considered only marginally. We do not need a multi-hop protocol, due to the fact that all devices are in range, e.g. in a laboratory. In this paper we survey well known update protocols and architectures for software updates in WSN, discuss the solutions and compare them to our approach. As a conclusion of our extensive simulation follows to sum up that the developed protocols do a fast and scalable as well as a reliable update.

Keywords: WSN · Software update · Low-power devices

1 Introduction

Wireless Sensor Networks are used in different environments. Some of the basics of such wireless networks are described in [4,6,8,14,22]. Special constraints regarding power and time management in these systems are shown in [7] and [2]. Many example applications and its areas are shown in [3,12,17,21]. Since most of these sensor networks are energy constrained one of the main goal of each application in such an environment is to make sure that the lifetime of every node is as long as possible and at the same time reach a satisfactory performance level.

C. Benavente-Peces et al. (Eds.): SENSORNETS 2017/2018, CCIS 1074, pp. 74–97, 2019.
https://doi.org/10.1007/978-3-030-30110-1_5

In specific circumstances, e.g., after production of devices, a initial firmware must be flashed to the nodes. This requires that every device gets the firmware and could be realized with a wireless update mechanism.

A company which produces a lot of radio hardware, e.g., smart devices like wireless smoke detectors or heat cost allocators does not want to flash and/or update all devices one after the other. They usually want to update all devices at once. In this laboratory-like use case, the update process must ensure that the firmware is flashed entirely and without any errors. All nodes should receive the firmware. It should not happen that a node does not receive the update. The whole update process should happen automatically, this means that no user is required to update the nodes. Due to the fact that many wireless sensor nodes are driven by battery most update processes have the aim to reduce power consumption as well as minimizing the update time. Most of the update protocols use different approaches to ensure these points. We show in Sect. 2 that most of the update mechanisms resolve more issues than needed, hence this results in additional effort to realize this update protocols on a real hardware.

This is one reason for us to develop a new update protocol optimized for a specific use case. This use case is the flashing of software on all used sensor nodes during the development process for a wireless sensor network. This means sensor nodes need an update for either to test new code or to extend the current code base to more than one device. This includes the possibility that the firmware update is needed for a small amount of nodes at the beginning of the development as well as a very large amount of nodes in later stages of the development that need to be updated. If the sensor nodes do not support a wireless update protocol they usually are programmed with a simple hardware tool like the so called Gang-Programmer [28]. This kind of programmer can update a small amount of nodes in parallel. Such a device is used, e.g., in Germany to flash nodes which are ready for production use. Since we had known the different scenarios for the update process itself, the scalability of such a design for the complete process was one important requirement. The next goal of our update model was the minimization of the programming time of each and every node. Every time a new software is flashed to the nodes in the sensor network there is a delay until new software can be tested, i.e. the programming time of the nodes. Furthermore, in a deployed wireless sensor network the amount of updates should be minimized because the update itself uses a lot of power and therefore battery lifetime is strongly reduced. This implies that power consumption of an update process is one of the most important things in a deployed network, but in our scenario not the main goal.

The protocol we designed tries to reduce the power consumption as much as possible, but there are possibly more sophisticated approaches to achieve this aim (done in other protocols). The flawless and complete transmission of the new code are naturally important factors because the update protocol would otherwise not be reliable. As a result of either manual intervention or many update runs would be needed to secure that all devices receive the new software. This said, such an update mechanism would not be very useful.

A preliminary version of this paper was presented at the 2018 SENSORNETS Conference [23].

2 Background and Related Work

The problem of programming nodes in wireless sensor networks is discussed in several articles. A survey is given by [20] and in [18], a brief overview of some update mechanism is also shown in [5]. There are many solutions for several hardware platforms and specific use cases. While some of these protocols are designed for a specific hardware to run on, or only allow partial updates efficiently, e.g, by incremental/compressed or differential updates shown in [13] and [16], try others to be a more generic solution for the process of software updates in WSNs.

Software updates in a WSN require data dissemination protocols. The main differences between protocols for gather sensor data and updates are described in [20]. In the following section we discuss some of the update protocols and compare them to our solution.

2.1 Trickle

Nodes in a wireless sensor network that implement the Trickle [15] protocol transmit code updates throughout the network. These code updates are sent by an node if it has not heard a few other nodes nearby transmitting the same updates using a maintenance algorithm ("polite gossip"). All messages sent by Trickle will be sent to the local broadcast address. The two possible options to a Trickle broadcast are: every node that receives the updates are up to date or the code version of a receiving node is out-of-date. The detection itself can be the result of an out-of-date node receiving new code updates by another node or an updated node hearing a node nearby has outdated code.

It does not matter which node transmits first as long as all the nodes in the WSN can be reached somehow. There is no master node that needs to be in range of all the other nodes although there must be one node injecting updates from outside the network. Due to the nature of Trickle the need for an update can be detected through a reception or transmission of a message. This allows the protocol to operate in sparse as well as dense networks.

In comparison to our update protocol, Trickle allows multi-hop updates throughout the network. Not every node needs to be in range of the node transmitting new code updates. These multi-hop updates have a negative effect on the power consumption because nodes must forward data to other nodes regardless of the need to send own data. During the development process we can arrange the nodes in range of a single transmitter to update all devices at once to save energy. Disseminating data via multi-hop in a large wireless sensor network can be very time consuming. To achieve a rapid propagation as well as a low maintenance overhead, the nodes adjust the length of their gossiping attention spans. This means the nodes in the network communicate more often when there is an

update. One of the biggest drawbacks of Trickle is that it assumes the nodes are always on. Due to this fact, nodes can not be run on battery because of the high energy consumption resulting from continuous listening for code updates. The probability of missing code updates by a node which is not always on and wakes up occasionally for receiving code updates is given. Such a node needs to be either listening for update packages until it receives an package or needs to define times for code updates within the WSN. A WSN usually has very low duty cycles to save energy. Therefore nodes are most of the time in sleep mode and not often able to receive messages.

2.2 Deluge

Deluge [11,20] is a reliable data dissemination protocol based on Trickle to distribute large amounts of data among a WSN. It allows to transfer code updates from one or more source nodes to multiple other nodes via multi-hop. Representing the data object as a set of fixed-sized pages provides a manageable unit of transfer which allows for spatial multiplexing and support for efficient incremental updates distinguished by incremental version numbers. The identical data object is distributed to all of the nodes. Therefore the data is split into fixed size pages which is the basic unit of transfer to provide the following advantages: it restricts the number of states a receiver has to maintain while receiving the data, it allows efficient incremental upgrades from prior versions and spatial multiplexing. The pages itself are split into fixed sized packets. The packets and pages include a 16-bit cyclic redundancy check (CRC). The nodes broadcast advertisement packets containing a version number as well as a bit vector for all new pages received. The broadcast is a variable interval based on the updating activity. For incremental updating its image to a more recent version, a node listens to supplementary advertisements. After that it requests the required page numbers from a distinct neighbouring node. When the node receives the last package to complete a page it sends an advertising broadcast prior requesting further pages to improve pipe lining within the network. Deluge does neither support ACKs or NACKs. The requesting node pulls data by inquiring packets for a new page or for a missing packet from a previous page.

In comparison to our approach Deluge is robust to asymmetric links, where a link in one direction can have a significantly different loss rate in the other direction. A three-phase handshake protocol helps to ensure a bidirectional link exists prior transmitting data. Furthermore, if a node has not completely received its update after making a certain amount of requests, it searches for a new neighbour to request data, rather than hanging to a bad link. This approach helps to send updates fast, because sticking to a bad link leads to packet loss and therefore to a longer transmission due to the fact that packages need to be requested more often. Because the packages arrive more reliable and faster, less energy is wasted through slow or repeated transmission of packages. Bad links are a common problem for field deployments and can be minimized through decreasing the distance to the transmitter during the development. Deluge dynamically adjusts the advertisement rate to enable quick propagation. The use of spacial

multiplexing allows parallel transfers of data. A downside of the use of spacial multiplexing is that the entire network must remain powered on to achieve the full benefits of spatial multiplexing. This increases the power consumption of the nodes and makes it impossible to run them on battery. Another drawback is that Deluge is not supporting fault detection and recovery. As we said we assume for the development and testing each of the nodes to be in range of a transceiver for programming the nodes, a multi hop protocol is not needed. Furthermore, a multi-hop protocol is usually harder to implement than a single-hop protocol.

2.3 Deployment Support Network (DSN)

A Deployment Support Network [9] is basically an architecture instead a particular implementation of a protocol for software updates. It is a possible choice for accessing the sensor nodes independently by providing a parallel network for maintenance. Accessing each node individually for software updates or maintenance in general is usually unattainable because of the inaccessibility of the nodes and the huge amount of nodes. Accessing the nodes via the WSN has distinct downsides. It postulates on the network being working, it has an negative effect on the network performance due to increased load and increases the energy consumption of the nodes. A DSN can face some of those problems by temporarily attaching small and portable nodes the WSN. This allows a host, e.g. a PC to connect to the WSN and open a virtual connection to any of the nodes and communicate with both, the attached DSN-Node and WSN-Node.

A DSN is advantageous during the whole development cycle. It allows convenient distribution of new code to all of the devices. In a later production and field deployment it allows to monitor, validate and measure with minimal impairment of the WSN.

In contrast to our update protocol, a DSN allows large-scale deployment without losing the ability to observe and control all nodes without the burden of fixed, wired infrastructure or changes to the target system. Due to the fact that it is an architecture it does not specify a protocol for node updates. Another disadvantage is that a parallel DSN network has to be managed and deployed in parallel to the WSN.

2.4 Four Step Update Process

Many update protocols in wireless sensor networks, e.g., Deluge [11], MOAP [24] or Trickle [15] use the idea of a four step process [20] to ensure the demanded functionality. This process is shown in Fig. 1 and includes four steps.

The first step (advertising) ensures that all nodes know the current software version. If there is more than one source that could provide the needed software, a target must choose the best source, e.g., by checking the link quality of the radio channel to the different possible sources (selection). These two steps include the concept of multiple sources and therefore multiple senders. At the same time these protocols often use the idea of multi-hop. This approach allows nodes to reach other nodes in a wide sensor network outside their own transmission area.

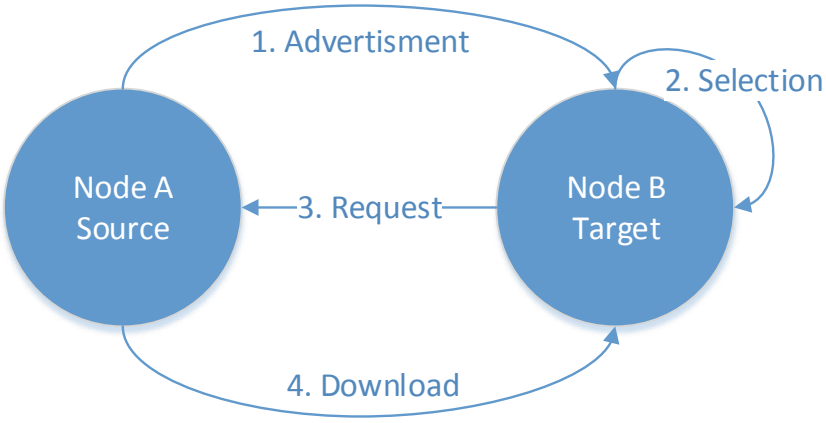

Fig. 1. Dissemination idea for update protocols [23].

So-called multi-hop protocols can distribute code to nodes which are not in direct reach of the source, but receive new software from in between nodes. This idea may be the best solution for some problems with appropriate hard- and software but does not fulfil our needs for the complete application programming of a large amount of wireless sensor nodes while developing software. All these programmable devices are within a small area, therefore reachable for one sender within this area. If the concept of multiple senders would be used, this could lead to interference with other sources in this network. Consequently, a mechanism would be needed to ensure that this kind of interference, i.e. colliding messages, does not occur.

Our update model does not include the idea of multiple senders. This allows us an easy mechanism to broadcast messages, due to the fact that there is only one sender and therefore this message can not collide with other messages. There are no other concurrent messages at all at the same time in the entire system. The third step (request) establishes a communication channel between source and target. The last step is the actual download of the inquired data for updating the target node. After the execution of all these steps, the new software is executable on each node that received the update.

Such sophisticated update models solve crucial problems to their specific use case(s). But the additional steps of advertising and necessary following selection would need more time and energy in our scenario. Because we have a very dense network these steps only bring their disadvantages, nonetheless they would work with such networks. Though the advantages of such an approach would be lost and therefore these kinds of methods add unnecessary overhead. Analysis of existing software update mechanisms is the reason for our different approach to the problem of updating sensor nodes. All these protocols are not satisfactorily for our challenges, admitting they cover their specific problems very well. The next chapter does introduce our protocol ideas to reach a fast and reliable update while developing new software for sensor networks.

3 Our Approach

In this chapter we describe the underlying ideas of the update protocol and give some reasons for their use. The process is designed to update an arbitrary amount of nodes in reasonable time. The update itself does only support full updates, i.e. it is not possible to update a part of the firmware while keeping other parts of the software. The complete flash memory is reprogrammed with the new program. Since we developed and implemented the update process for a specific hardware from Texas Instruments (TI) we used existing software and ideas where possible. However, these ideas can be easily adopted and used for other systems as well. The already available TI 1:1 update process was the cornerstone for the further development of our update process.

In the existing update mechanism a code distributor, described in the next chapter, communicates over USB with a pc application to collect the source code and send this new code to the update device, the sensor node. The TI update process needs a particular software on the sensor node, which will be updated, running. Due to this fact, the update process is only one-time executable. If the new loaded software does not support the specific sequence to launch and execute the update process, the complete process is not usable any longer, and must be manually flashed again. These limitations were another reason for designing a more practical update mechanism which fits better for our use case.

The TI software, after the wireless update is started and initialization process is complete, does use a simple stop and wait method. This means that after every single data packet an acknowledgement from the device is expected. If the validation is successful the next data packet is ready for transmission to the device. Otherwise, the current data packet will be sent again. This idea was extended and adapted to get it working with more than one device. The new approach does work similar, although at the moment not every packet is validated, but after a specific number of data packets the update device transmits an acknowledgement packet to advert the current position in the entire update process. This is also known as a go-back-N protocol. While the source code distributor receives good acknowledgement packets, i.e. no error occurred, it continues with the next valid data packet. A bad packet indicates an error and this data packet will be sent again as long as all update devices do not correctly receive it. The bad data packet is now the new position in the update and from this position all packets are transmitted. However, an error is not recognized immediately, but only with the next acknowledgement.

Since the process is designed to work with any amount of sensor nodes it must be guaranteed that all nodes know when to send their acknowledgement packet, otherwise some transmissions will collide because of interference with other possible transmissions. This is guaranteed by the used time division multiple access (TDMA) mechanism. Every node has a particular, fixed time slot when to send the acknowledgement. The initial easy idea and implementation uses the TDMA mechanism after a fixed amount of data packets regardless of the quality of the radio channel. A bad transmission channel could result then in slow error recognition. On the contrary when the radio channel is quite good

the acknowledgement phase is kind a waste of time since an error is unlikely. An update protocol, which uses the information about the radio channel and its quality could save time as well as energy because it decides flexible when an acknowledgement phase is needed more often. The adaptation to the quality of the current radio channel is used in several environments. In some TCP/IP implementations there is a mechanism called AIMD (Additive Increase Multiplicative Decrease).

When the error rate is low the ack phase is used infrequently, but is increased by a multiplicative factor after an error appear, originally shown in [29]. A detailed analysis of the algorithms is presented in, e.g., [10] and [19]. A similar, simplified mechanism is used by our second developed update protocol. The idea was to get a better adaptation to the actual needed acknowledgement rate given by incorporating the physical quality of the radio channel. Mechanisms, messages and its sequences to initialize the update process are presented and analysed in Sect. 5.1. The complete update protocol, not only the acknowledgement phase of the process, is a time based, shared protocol, meaning all sensor nodes share the same system time. Data packets with new code come each fixed time step and all nodes are able to synchronize with the system time with every data packet which is received. This guarantees that every update device, which is correctly synchronized with the code distributor (master clock), can switch to receive mode very shortly before the packet is transmitted by the update distributor. This reduces energy consumption since update nodes are only in RX mode when it is absolutely necessary.

Because of the natural clock drift every update node must synchronize itself with the master clock, which in this case is the clock from the source code distributor. This guarantees that RX windows remain narrow. This basic idea is valid for many update scenarios. These concepts show that practical relevance of the update mechanism is of real interest here, as the theoretical background of our update model is not extremely difficult. In the next chapter we describe our particular hardware and software model on which we actually implemented and tested these update designs and protocols.

4 Software and Hardware Architecture

The software architecture and its distribution is based on the hardware devices used. Due to the fact that the update protocol has several tasks, we operate with different devices, which are suited for their specific function. The participants, which are involved in our update model are the sensor node (CC430), the distributor of the update code (access point) and the user interface represented as a desktop application.

The CC430 is a programmable watch delivered within the eZ430 Chronos development kit. This device (CC430F6137) is the sensor node in our environment and has an built-in sub-1 GHz wireless radio module based on the CC1101. It stores a bootloader (max. 2 KB) which handles the complete update process on watch side. Therefore, the code size of the complete update protocol software

must be less than 2 KB, together with device drivers for the flash, ports and radio modules. This limitation allows only a small implementation as a more complex protocol could lead easily to a code size which will not suit the available bootloader ROM size. The eZ430 is equipped with 32 KB of internal flash memory and 4 KB of RAM [26]. The microcontroller supports different internal sensors and offers several power modes to save energy. While not in active mode (AM), but waiting for external/internal events, e.g., expired timers or incoming radio packets, it can switch to different low power modes (LPM0 – LPM4). In our scenario the most update time is spent in LPM3. The lowest power mode LPM4 disables all clocks, which are needed to provide a stable time base and hence, this mode cannot be used by our protocol.

The access point is a MSP430f5509 equipped with a CC1101 radio core [27] which is used for communication with the watch. The access point contains an USB interface for further communication with other USB devices like a pc. This device is responsible for dissemination of update messages, e.g., the new code, meaning it co-ordinates the complete sequence of the update. The last involved party is the user interface. This application handles the firmware file and partition into small update packages so the access point does not need to perform further actions with the packages, but only broadcasts them to the watches. The communication between the access point and GUI is handled via the USB interface. The complete structure of our update model is shown in Fig. 2. All watches

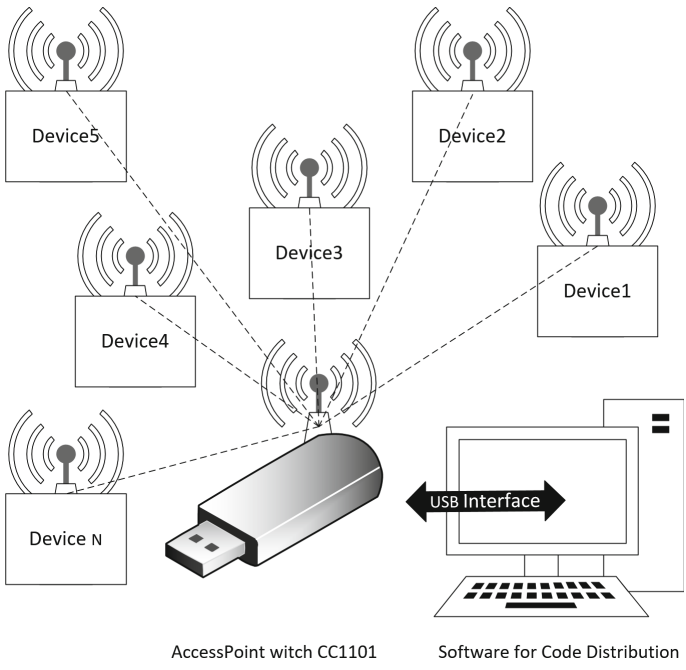

Fig. 2. Overview of update model [23].

are placed in a small area within radio reach of the access point. Besides that, no other additional requirements must be met. The update is initiated by the application. After starting the update process, no user intervention is required. The access point does now communicate with all watches within radio reach to update these nodes. In the design of the entire update protocol, communication among the different sensor nodes, i.e. the CC430 devices, is not planned. From the perspective of the update devices it is a 1:1 communication with the access point. On the other side the access point distributes code to all devices, hence this is a 1:n communication. The particular timings of messages and its sequences are presented in the next chapter.

5 Analysis

In this chapter we analyse the update protocol itself and its several states. We show how we calculated the power consumption as well as the computation of the execution time of the update process.

5.1 Protocol Overview

All sort of messages within the system are summarised in Fig. 3. As seen in the protocol messages, the hardware settings of the radio communication incorporate a 4 byte preamble as well as a 4 byte sync word. The preamble is an alternating series of ones and zeros, i.e. 0xAA is transmitted. The sync word consists of application specific data which is used for byte synchronization. Moreover, it allows a distinction among systems with the same hardware as the radio writes only data to the internal buffer if the correct synchronization word is received [25]. In our application it is used twice with the value of 0XD391 to get a 4 byte sync word. Because of the enabled 16 bit checksum calculation, there is no need to check for errors in software as erroneous packets are removed automatically. The data packets data field is adaptable, meaning during the update run the access point could decide to increase or decrease the length of the payload. This mechanism is currently not in use, but could be used in future versions of the update process. All other messages have a fixed length and can not change its size during protocol execution. The exact sequence of the complete update process is shown in Fig. 4. At the beginning of the firmware update mechanism all watches must execute its bootloader software. This can be done via reset of the device manually as well as by triggering a software reset if this is supported by the current firmware. Then the bootloader is executed where it is possible to start the update (automatically if in a specific time interval no user input was performed) or execute the application. After the device has initiated the execution of the bootloader, the update device transmits every specific time interval, e.g., 3 s, the RFU (ReadyForUpdate) message. When the access point receives such a message (this message comes from a watch which has not an update ID yet and therefore was not recognized yet) it replies with the RFU_ACK message and sets the update ID for this particular watch and update run. The frequency of

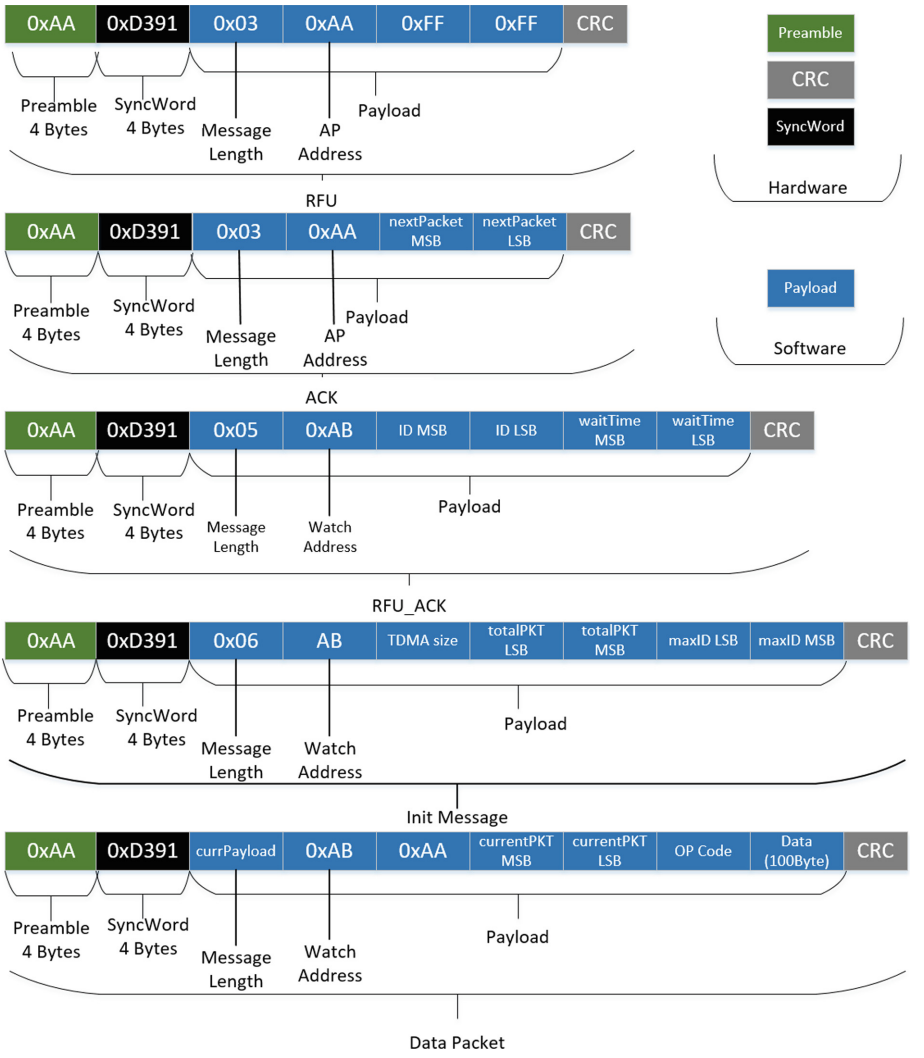

Fig. 3. Protocol messages in the update system [23].

the RFU message determines how often the watch can send its beacon in the context of the discovery phase of the update, e.g., if an error occurred or some other watch is sending simultaneously and another RFU message is needed.

There is a hardware mechanism used during the discovery phase to prevent most of the colliding messages. This stage is the only state where collision between different messages can occur. In all the other states is only one message at any time in the complete system. The watch switches to LPM3 after this short communication to save energy. After a particular and by the user configurable amount of time the discovery phase is done and the update goes into the next

state, i.e. init state. At this stage all watches that need an update should be known by the access point. This is guaranteed by setting an appropriate amount of time for the watches to remain in the discovery update state. All watches wake up at the same time, very close before the first actual packet is send and now listen to the first update package, i.e. init packet. This is achieved by time information the devices gain from the previous communication with the access point, which is included in the RFU_ACK message.

The init packet does contain informations such as the update size, number of update devices and some information about the acknowledgement phase and its length. In the case a watch does not get an answer for the RFU message during this process or does not get the init message correctly this device does not participate in the following update process, rather resets itself. After the init mechanism for each node is successfully completed, the actual update starts. In this case an adjustable, but during the update process fixed amount of data packages is transmitted before the first acknowledgement phase is run. All packets come in specific time slots, so all watches can go into RX mode very close before the actual packet transmission starts. This guarantees that a very small amount of power is consumed, since RX time is not longer than absolutely necessary given by physical parameters and the calibration time. The time among data packets is 100 ms and 5 packets are sent one after the other before starting with the acknowledgement phase. This means the data packet duration is 500 ms. Afterwards, meaning 500 ms packet round with 5 sent packets is done, the update devices, whether they received packets successfully or not, trigger the next and final state, the TDMA phase. In the case of an error at the beginning of the data phase all further packets are lost as well. Because of the design of the protocol, an update device is only capable to save the current state in the complete update, but not single packets which are needed to complete the update. This behaviour could be changed in further versions to amend the update.

The entire TDMA windows is divided into small time slots for every single node. The length of this phase depends on the amount of sensor nodes, which participate in the update process. Each update node has the chance to send an acknowledgement packet which contains the packet number that is expected next, hence, this information covers the last successfully received packet number. The access point has all needed data, after this TDMA mechanism, to decide which packet should be transmitted next, i.e. the smallest packet number that was received by the access point. In the case an ack from a node is lost, for whatever reason, there will be no error handling performed. When the next ack phase starts there is a new chance for this node to send its ack successfully. The data phase is started again after every TDMA phase. This process is repeated as long as there are no more acks with a packet number below the highest possible packet number is received, therefore all nodes have the complete new code. Each node that has received all necessary packets successfully, resets itself automatically and starts the new application if there was user input, otherwise the bootloader waits for another update run, meaning the device starts with sending the RFU message. This process can be cancelled by particular input to easily start the application.

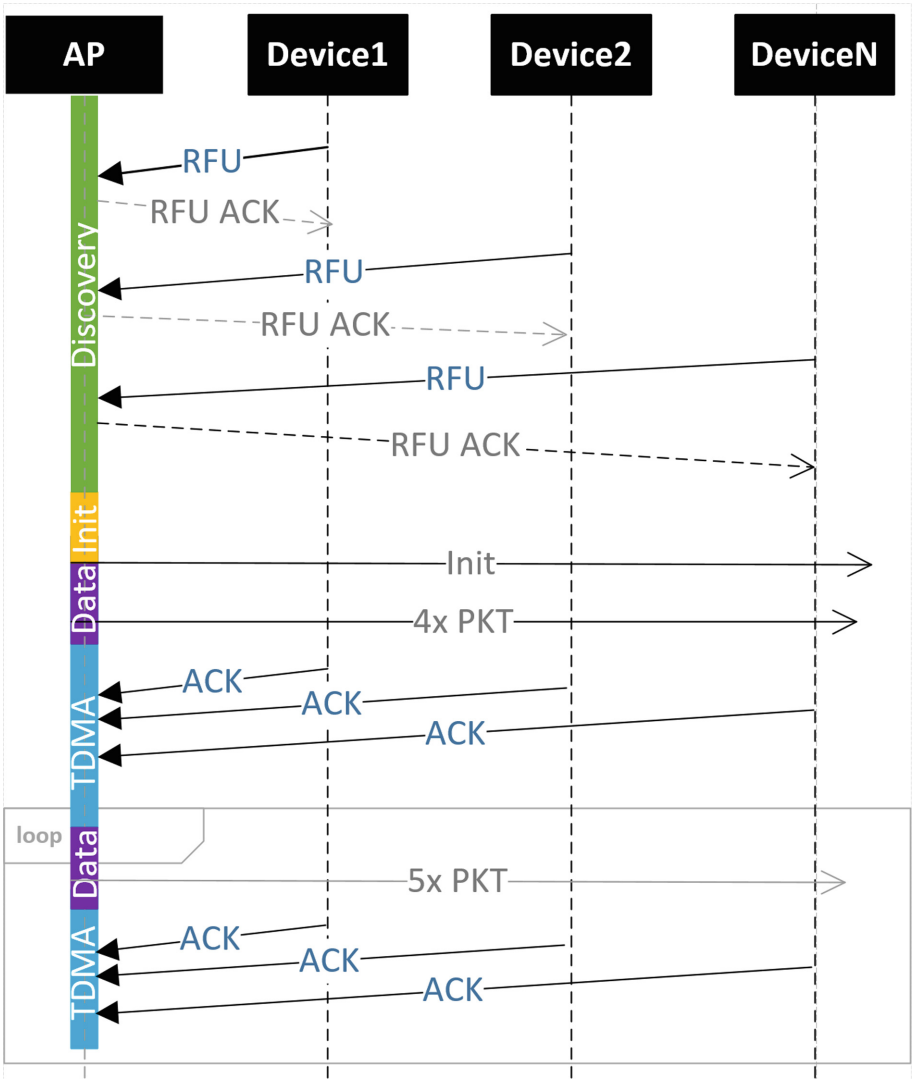

Fig. 4. Protocol schedule during the update [23].

The acknowledgement phase is the most complex phase of the complete update protocol. The more devices are participating in an update run, the longer this phase must be. The ack phase is always determined by a specific integer multiplicator of this 100 ms, dependent on how many devices need an update. This multiplicator does increase by 1 every 10 nodes. The decision to set these particular timing intervals were made after evaluating the USB communication between the access point and the pc application.

5.2 Calculations - Power Consumption

Because power consumption of an update protocol is an important requirement in the design of such a process the exact energy consumption of all messages are presented in this chapter. The power consumption of the update process is divided into four states of the watch: the active mode and LPM3 of the watch and its CPU, the receiving (RX) and the transmitting (TX) mode of the radio module. Table 1 shows several power levels of these states in mA. To guarantee a low power consumption, the completion time as well as radio transmissions should be minimized and update time should be remain in LPM3 whenever possible. The latter can be reached by an exact time based protocol as we have developed. The timings of each update state is known to every single device, hence the active time of any device is minimized to the actually needed active time. The rest of the update time is waiting for an event to get triggered or to let other devices finish their transmissions. The Power Consumption given in Ah, length of the messages in bytes and the timings (RX and TX) in ms of our messages are shown in Table 2 (from watches perspective, the access point power consumption is kind of negligible because its not battery driven and always on a secure power connection). The bytes added and removed automatically by hardware, i.e. the preamble, sync word and the checksum are already included, meaning the actual usable data is always 10 bytes less. This overhead is mandatory for all messages transmitted by the radio module. The shown power consumption is in 10^{-9} Ah. Timings and consequently power consumptions for these messages are valid for a transmission rate of 250,000 bits per second. Because the radio module needs calibration each time a communication is initilized a value of 721 µs with a power consumption of 9.5 mA must be added to each receive or transmit operation [26]. This results in additional $1.9 \cdot 10^{-9}$ Ah per radio event. As we know all the necessary values to calculate the complete power consumption of the CPU and the radio module during one update run the following formula shows the power consumption for the radio module for all transmission:

Table 1. Power consumption @12 MHz [23,26].

Voltage	IDLE + CPU active	TX	RX	LPM3
3.0 V	1.7 + 2.75	33	16	0.0022

Table 2. Power consumption of different messages [23].

Type	Length	TX time	RX time	Power
RFU	14	0.45	0.00	4.10
RFU_ACK	16	0.00	0.51	2.27
InitPacket	17	0.00	0.54	2.41
DataPacket	116	0.00	3.20	14.22
ACK	14	0.45	0.00	4.10

$$\text{radio\_power}_{\text{active}} = a\cdot \text{RFU} + a\cdot \text{RFU\_ACK} + \text{InitPacket} + b\cdot \text{DataPacket} + c\cdot \text{ACK} + (2a+1+b+c)\cdot 1.9\cdot 10^{-9}\,\text{Ah}$$

where a stands for the number of RFU messages the watch transmits, b the number of data packets were received by the watch and c the number of acks the device is transmitting during the update run. Because the exact radio idle time is hard to determine we calculate the radio as always idle, knowing this is not correct but sound. This formula represents the radio power consumption for the idle state and must be added to the complete power consumption:

$$\text{radio\_power}_{\text{idle}} = 1.7\,\text{mA}\cdot \text{updateTime}$$

The CPU is at least 90% of the update time in LPM3. The only CPU activity while updating the firmware, is before transmission and after or during receiving (when the FIFO hardware buffer is full) of a radio packet, since these events are interrupt-driven and wake up the CPU from all low power modes. Additional CPU active time is needed before the update starts as well as after a data packet was received, because this data must be written to internal flash memory. This is why we can calculate the worst case active CPU time and its power consumption with 10% of the update time, knowing the exact CPU active time is dependent on how many packets were received/transmitted. However the 10% calculated time is surely higher than in the actual implementation. Every 100 ms the radio waits for a data packet, which needs time for receiving (copy values, sync mechanism) and when successfully received the time for writing it to the flash memory. After some evaluation of these operations we can safely assume these operations do not need 10 ms and this would be the time amount to reach the 10% active CPU time. To get a wrong but safe bound we add, as the exact CPU active time is also hard to determine, for the complete run

$$\text{CPU\_power}_{\text{LPM}} = 0.90\cdot \text{updateTime}\cdot 2.2\,\mu\text{A}$$

and for the active state of the CPU

$$\text{CPU\_power}_{\text{active}} = 0.10\cdot \text{updateTime}\cdot 2.75\,\text{mA}.$$

The complete main flash memory of the device is erased and reprogrammed during an update run. These write operations also consume time and energy. Timings and power consumptions of all flash operations are shown in [26]. We calculate the power consumption during erase with the given typical value of 2 mA and during programming with 3 mA. While full erasing does need maximal 32 ms, the complete programming duration of the CC430 flash memory takes about 800 ms of active write operation. All flash instructions summarized result in $6.9\cdot 10^{-7}$ Ah, which must be added to the power consumption of one update run.

The sum of $radio_{active}$, $radio_{idle}$, CPU_{LPM}, CPU_{active} and write/erase operations on the flash memory give now the complete power consumption necessary by one sensor device during one firmware update. The best case (one device, minimum amount of packets are sent) of an update with a size of 27 KB, a data

packet payload length of 100 bytes and a TDMA window after each 5 data packets, can now be calculated. The minimum amount of packets necessary are 1 RFU message and its 1 ACK, the 1 init package which is followed by 270 data packets and in between of the data packets there are 54 acknowledgement packets needed. This results in 327 radio transitions. The minimum update time is 27 s, i.e. the 270 packets with one packet in 100 ms, plus discovery phase length which is calculated with 7 s – this is the value we used for our later shown experiments. This results in $6.01 \cdot 10^{-6}$ Ah for the active radio, $1.27 \cdot 10^{-8}$ Ah for the idle radio, $1.87 \cdot 10^{-8}$ Ah for the LPM of the CPU and $2.60 \cdot 10^{-6}$ Ah for the active CPU time. If exactly one device is involved in the update run, a total amount of $9.33 \cdot 10^{-6}$ Ah is used. The more devices are used the more the power consumption increases, but only because the time in LPM3 increases. With 1000 devices the additional power consumption for LMP3 is about $2.94 \cdot 10^{-7}$ Ah. This is much less than 1/10 of the complete update process of one device, therefore does not significantly decrease the number of possible update runs for these devices. The number of radio transmissions and the CPU active times do not increase, but stay the same.

The eZ430 chronos watch contains a standard CR2032 lithium battery with a nominal capacity of 220 mAh [1]. This means the battery lasts for about 23,500 possible best case update runs. This should be acceptable for most development processes. Although these results show only the best case with 1 involved device and hence errors would decrease the number of possible updates accordingly. An error rate of 1% would increase the needed amount of packets at least to 273, more likely an even higher number of packets is needed due to the design of the error detection, but also increases the CPU active time and the LPM time. The maximum amount of packets (worst case) for a packet error rate of 1% is 285, every error occurred directly after the ack phase and therefore all other packets in this data round are lost too. This results in additional 3 more acks and 15 more packets, i.e. a total amount of $2.26 \cdot 10^{-7}$ Ah must be added to the power consumption.

5.3 Calculations - Time

The main objective during the development of the update protocols was to achieve a fast and reliable process and hence time evaluation of the update process is important. The time an update run needs can be calculated as follows:

$$\begin{aligned} \text{completionTime} &= \text{Time}_{\text{discoveryPhase}} \\ &+ \text{Time}_{\text{dataPackets}} + \text{Time}_{\text{TDMA/ACKphase}} \end{aligned}$$

At present the user sets the time of the discovery phase. In this time slot each node has to complete its RFU communication and hence the more devices are updated the longer this time should be. Time needed for transmitting all the data packets depends on the size of the update. The usable flash memory of the device is 32 KB, which results currently, since the data payload of one packet

is 100 bytes, in maximum 32 s. The TDMA/ACK phase of the update is calculated differently and dependent on the amount of sensor nodes which require an update. The first 10 devices can use the normal time between 2 data packets to transmit their acknowledgements. After that every 10 devices get an additional 100 ms time slot to transmit the ack packet. Thus, the TDMA time can be computed as follows:

$$\text{Time}_{\text{TDMA/ACK}} = (\#\text{devices} - 1) \;/\; 10 \cdot 100\,\text{ms} \cdot \#\text{acks}$$

This time increases with growing amount of update devices and becomes the most important factor that determines how long an update run needs to be done. With, e.g., 1000 devices and an update size of 27 KB time spent in the TDMA phase is about 535 s, while the actual sending of the packets is 27 s. In later improvements of the update protocol the time spent in the TDMA windows should be minimized to get faster update results. The time to finish the update depends heavily on the error rate. The more errors, the more packets have to be sent. But with growing number of sensor devices the TDMA phase has the most impact on the finishing time. If we use again a small error rate of 1% different update times could be observed. The best case with regard of completion time are errors that are recognized immediately. With 3 additional sent packets, 273 in total, the TDMA/ACK phase length would be the same – still under the best case assumption. Since the last round is incomplete and if all 3 packets are received without errors the last TDMA phase is not started, leading to no additional time cost for the update process.

The worst case on the other hand is completely different. If an error occurs right after the ACK phase, the whole packet round is unusable and therefore, all packets are lost. As a result only 3 errors would cost another 3 additional TDMA rounds. While only one device is involved, only the additional 15 packets sent are relevant for the execution time of the update. Given the example of 1000 Devices, 3 more TDMA phases would mean: $99 \cdot 100\,\text{ms} \cdot 3 = 29{,}7\,\text{s}$ more time to complete the update for all devices. Thus, the exact time/place of the errors has a huge impact on the completion time of the process. Furthermore, with more devices an error becomes more and more important regarding the completion time.

6 Experiments

This section contains experiments we made with our available hardware. Not all introduced update protocols contain an evaluation of their methods on a real hardware environment, therefore we particularly want to describe our experiments we made. For all tests of the update process we considered all available nodes which need an update are in very close distance to the access point, i.e. max distance was 1 m in our test environment. The most important metric to measure quality of an update process in our scenario, i. e. the development of new software for sensor networks, is the update duration. Another top priority achievement is reliability. All devices must be updated completely and without

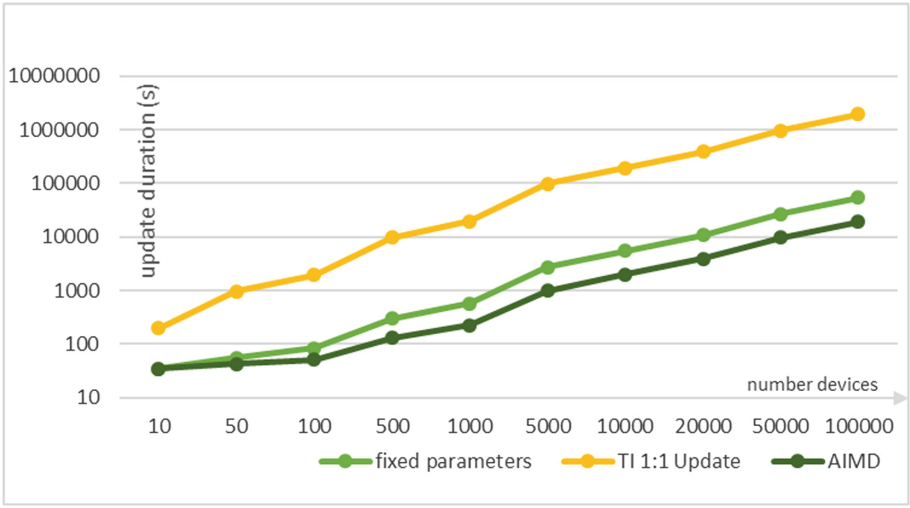

Fig. 5. Update duration with different number of devices [23].

errors. Although energy consumption is important in wireless sensor networks, it has a lower priority in this case. Otherwise the battery must be replaced very frequently and most benefits of an update process like fast and simple testing of new code would be lost.

Calculated best case results for updates are shown in Fig. 5. The x-axis (bottom) shows the amount of devices involved whereas on the y-axis (left side) the time to update this amount of devices is shown. The diagram is valid for an update size of 27 KB. The fixed parameters update process has an acknowledgement rate of 5. This means that after each 5 data packets there is a time window where all watches can reply with an ack packet. The number of ack phases can be reduced, if the radio channel is good and an error is unlikely. In this case we used a different approach for the ack phase. The AIMD and AIAD variant introduce easy mechanisms to respond flexible to the current radio channel and adapt the ack rate. AIMD start at the same ack rate and from there on the algorithm decides whether to increase or decrease the ack rate. The discovery phase was set to 7 s during all experiments. This was sufficient for our amount of sensor nodes, but must be adjusted if more nodes are updated simultaneously.

The acceleration factor between different update modes (TI 1:1, fixed, AIMD) is shown in Fig. 6. Because the TI update does only allow a 1:1 update, the 1:n update process is obviously faster. The factor does increase with growing number of sensor nodes but is kind of limited due to the acknowledgement phase of the update protocol. As shown before in Sect. 5.3 the ack phase becomes the bottleneck of the update process since every ack round needs significantly more time than sending the 5 data packets one after the other. In case the AIMD and/or AIAD implementation is used and the quality of the radio channel is good, the ACK phase does not trigger as often as before, thus saves a lot

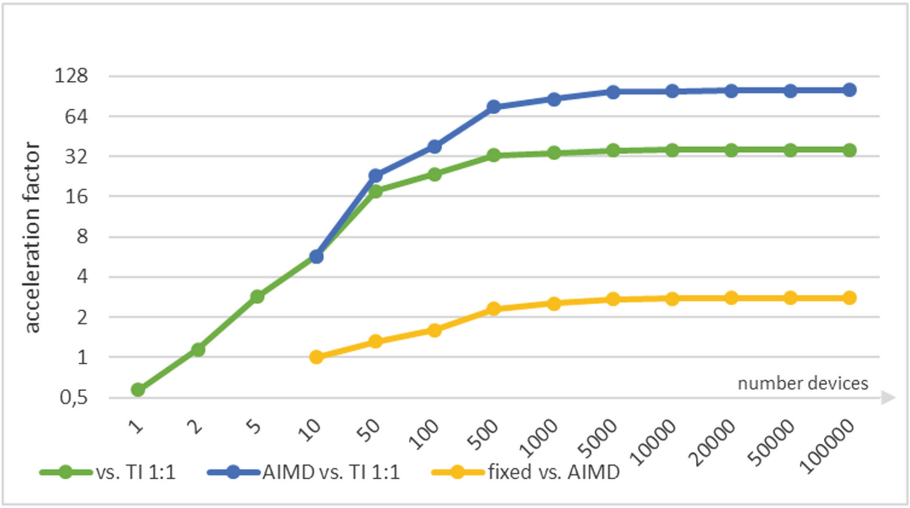

Fig. 6. Acceleration between the different update modes [23].

of time. These diagrams always show the best case, which means that no error occurs during the entire update time, hence are at this time only theoretical values. The experimental results, which will be presented in the next section will indicate how likely it is to achieve these values.

6.1 Results - Static Acknowledgement

Table 3 shows some actual results from experiments we made. These initial results were executed for all nodes within a small area and equipped with the fixed parameters software. One hundred of each test runs, i.e. in total 500 update runs were executed with several amount of sensor nodes. The table shows the amount of devices we used for one update run (up to 25), the time the experimental worst case (WC) run was longer than the best case (BC) calculation and the average and worst case PER (packet error rate). The entire column, i.e. the rate of successful updates gives an overview of how many of these firmware

Table 3. Measurements – static variant [23].

#devices	WC-BC	complete	avg.PER	max.PER
5	0.6 s	99.2%	0.12%	1.09%
10	2.5 s	99.8%	0.45%	3.75%
15	3.0 s	99.1%	0.66%	4.69%
20	4.2 s	99.8%	1.23%	7.76%
25	7.7 s	99.6%	2.92%	9.72%

updates were done totally without errors, meaning a runnable new software is stored in flash memory. PER is the packet error rate which indicates the channel quality between access point and watches during the software update.

These experimental results show, that completion rate does not correlate with amount of nodes which are updated. Through all test runs the completion rate was approximately the same with no spike in either one direction. In other words the mechanism does not become unreliable with more sensor nodes and hence, is able to scale with the problem size. The simple update protocol has a dependability over 99%. In the present implementation an error can occur in the final data round, the acknowledgement gets lost and therefore the update device can not complete the run.

A more sophisticated, explicit error handling like a two-way communication between the update device and the access point during or at the end of the update process could increase the percentage further. When both devices expect a validation packet the error is recognized for sure, therefore an explicit handling for this case could be created. Since in all experiments nodes where in a very dense formation, i.e. the average distance between the different nodes was about 1 cm, the PER does correlate with the number of nodes. This leads to the assumption that there is possible interference between the individual sensor nodes, which leads to corrupted packets.

The power consumption does increase over time since more CPU and radio activity is necessary. The calculation results shown in Fig. 5 for more devices will most likely not be reached since average PER is not near 0, but does vary between 0% and 3% during different update runs. The duration to finish the update on all watches goes up with the PER. As long as the number of devices is 10 or lower the TDMA has no influence on the update duration, as shown in the previous section in Sect. 5.3. The duration increases only due to packets which are sent again. In all other cases time increases not only by sending data packets again, but also with the additional time spent in the TDMA window. This behaviour was noticed in our experiments between 20 and 25 update nodes. In both specific worst cases the PER was high, but the actual completion time and its difference to the best case was significantly higher with 25 devices compared to 20 devices. The explanation for this is the acknowledgement phase. For every error that results in an additional ack window the update time increases by 200 ms with 25 devices instead of 100 ms with 20 devices. This also means, that for a large number of nodes an error is much more impactful than for a small amount of devices.

A further characteristic behaviour of the update protocol is based on the statically fixed acknowledgement state (in this specific case after every 5 data packets). A packet error can enlarge the number of packets necessary to be sent significantly and therefore the completion time for the update. This happens if an error occurred directly after the acknowledgement phase and is only recognized in the next phase. For this easy protocol it is not possible to detect this error otherwise than during the acknowledgement phase.

6.2 Results - AIMD and AIAD

The more sophisticated approach to get acknowledgement packets is to react to the current quality of the radio channel. Techniques used by our protocol to adjust are the AIMD and AIAD mechanisms. The initialization process and the message types are the same as before. The only difference to the previous mechanism is the rate at which the watches transmit their acknowledgement information. The acknowledgement rate starts again at 5. This means after 5 data packets the first TDMA is initialized. After this first ack phase the access point adapts the time for the next ack round. All other messages, timings and states are valid again. The results of these 150 (75 for each variant) update test runs with 25 sensor devices are shown in Table 4. The best case time calculation for both the AIMD as well as the AIAD mechanism is 38 s. This means that each single packet is transmitted without an error. This includes both data and ack packets. A non received positive ack, meaning an ack that would confirm that no error occurred is treated as an error. This minimal time can be reached by the update process as shown in Table 4, but the experimental evaluated average case is significantly higher. The average case still shows that the quality of the radio channel is sufficient to reach shorter completion times compared to the normal static mechanism. The best case of the first approach lies with 45 s higher than the average case of both the AIMD and AIAD protocols. However, with experiments evaluated we can show that the quality of the radio channel can go low enough that the update execution time is longer than with the static mechanism. The worst case execution time of the AIMD protocol is 55.9 s and with the AIAD 57.9 s. The worst case of the static method is with 52.2 s lower than both the AIMD and AIAD protocols. If a short, random error occurred than acks are not needed often, but if an error occurred and is detected rather late or the error is due to some radio interference and stays in the system a more often TDMA phase would be necessary.

Table 4. Measurements – AIMD/AIAD variant [23].

	avg.PER	avg.T	min.T	avg.#dataPackets
AIMD	1.49	43.6	38.4	287.75
AIAD	1.32	41.9	38.1	295.57

These algorithms have a delay to respond to both error cases. This fact can slow down the complete update process. The comparison between AIMD and AIAD is similar. Since the multiplicative decrease mechanism can react faster to errors which remain in the system, the worst case execution time with a high PER is shorter than the additive decrease mechanism. But if one random error occurred, the AIMD protocol needs a long time to recover before it is at the same ack rate as before the error, meaning it loses time compared to an additive decrease mechanism. The AIAD can react to such random errors much better.

As shown in the results, most of the time the quality of the radio channel is good enough to use the AIAD algorithm to get faster updates than AIMD. But as seen in Table 4 the AIAD needs in average more packets until the update is complete. However, as explained in Sect. 5.3 the TDMA/ACK phase is the update state where most of the time is spent. The number of data packets is not the crucial part of the completion time.

Since both the flexible ack mechanisms are faster than the previous one and the AIAD is faster than the AIMD, we learned that in general and over a lot of update runs the AIAD method gives the best results in terms of completion time and as previous explained this is the most important metric for the presented update scenario. The power consumption between the different protocols can be compared based on the average results shown in Table 4. The interesting part about the results is that the AIAD needs more radio transmission in general. Since more packets are sent during an average AIAD update run, the devices must be more often in RX mode. This results in higher power consumption for each sensor node. The LPM and radio-idle power consumption is lower than with AIMD as a result of the lower execution time of the update. The formula shown in Sect. 5.2 is now not exact enough to calculate the difference between the individual update mechanisms regarding the active CPU time. The active time with AIAD is still higher than in the AIMD process (in Sect. 5.2 this would be differently computed). The more packets are received the more the CPU is active. To calculate the exact differences between each update protocol a more specific CPU active formula would be needed. Since the power consumption was not the most important goal in the development, the power consumption is not significantly higher than in the static variant and the number of update runs should be sufficient for the software development, such exact computations were not done.

7 Conclusion and Further Work

We presented a fast and reliable but still simple update mechanism plus first improvements and their evaluation in this paper. This kind of an update protocol shows an easy way to update a large number of sensor nodes in parallel while keeping its effort manageable. Our approach could be used with a variety of different hardware. The co-development of a wireless updater during the development of software for the sensor network should be top priority since the benefit from such an update process is huge. The easy possibility to update all the nodes at the same time decreases the amount of time necessary to reprogram all sensor nodes and therefore, it is easy to test new code very quickly. An exactly shared time based protocol is not as hard as update mechanisms that come with multiple senders and/or multi-hop concepts. The power consumptions of this protocol is not problematic as calculations show that the update can be executed multiple times. The execution time is much less than in a 1:1 update/flash scenario and its scalability means it is usable with any amount of nodes. Improvements for the update protocol could be done by analysing the

radio channel to reduce the ack phase frequency. A further improvement for the protocol is not only to make the ack phase adaptable, but also the size, i.e. the payload, of the packets. A good radio channel and 1.5 times larger packet size, the TDMA frequency would be reduced by the same factor. This would lead to an smaller execution time for the update due to higher throughput.

Acknowledgements. This work was supported by the research cluster for Robotics, Algorithms, Communication and Smart Grid (RAKS) of the OTH Regensburg. Further information under www.raks-oth.de. This work was also supported by the Regensburg Center of Energy and Resources (RCER) and the Technology- and Science Network Oberpfalz (TWO). Further information under www.rcer.de.

References

1. Datasheet lithium manganese dioxide battery cr2032 (2018). https://www.mouser.com/ds/2/315/panasonic-lithium-cr2032-datasheet-837927.pdf
2. Sinha, A., Chandrakasan, A.: Dynamic power management in wireless sensor networks. IEEE Des. Test Comput. **18**, 62–74 (2001)
3. Altmann, M., Schlegl, P., Volbert, K.: A low-power wireless system for energy consumption analysis at mains sockets. EURASIP J. Embed. Syst. **2017**(1), 18 (2017)
4. Schindelhauer, C., Volbert, K., Ziegler, M.: Geometric spanners with applications in wireless networks. In: Computational Geometry: Theory and Applications (CGTA 2007) (2007)
5. Sternecker, C.: Reprogrammierungstechniken fuer drahtlose sensornetzwerke. Seminar Sensorknoten - Betrieb, Netze und Anwendungen (2012)
6. Chong, C.-Y., Kumar, S.P.: Sensor networks: evolution, opportunities, and challenges. Proc. IEEE **91**, 1247–1256 (2003)
7. Sivrikaya, F., Yener, B.: Time synchronization in sensor networks: a survey. IEEE Netw. **18**, 45–50 (2004)
8. Akyildiz, I.F., Su, W., Sankarasubramaniam, Y., Cayirci, E.: A survey on sensor networks. IEEE Commun. Mag. **40**, 102–114 (2002)
9. Beutel, J., Dyer, M., Meier, L., Ringwald, M., Thiele, L.: Next-generation deployment support for sensor networks. Computer Engineering and Networks Lab; Swiss Federal Institute of Technology (ETH) Zurich
10. Edmonds, J.: On the competitiveness of AIMD-TCP within a general network. Theor. Comput. Sci. **462**, 12–22 (2012)
11. Hui, J.W., Culler, D.: The dynamic behavior of a data dissemination protocol for network programming at scale (2004)
12. Kenner, S., Thaler, R., Kucera, M., Volbert, K., Waas, T.: Comparison of smart grid architectures for monitoring and analyzing power grid data via modbus and rest. EURASIP J. Embed. Syst. **2017**(1), 5 (2017)
13. Stolikj, M., Cuijpers, P.J.L., Lukkien, J.J.: Efficient reprogramming of wireless sensor networks using incremental updates and data compression. Department of Mathematics and Computer Science System Architecture and Networking Group (2012)
14. Meyer-auf-der-Heide, F., Schindelhauer, C., Volbert, K., Grünewald, M.: Congestion, dilation, and energy in radio networks. Theory Comput. Syst. (TOCS) **37**, 343–370 (2004)

15. Levis, P., Patel, N., Culler, D., Shenker, S.: Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor networks (2004)
16. Rickenbach, P., Wattenhofer, R.: Decoding code on a sensor node. In: 4th International Conference on Distributed Computing in Sensor Systems (DCOSS) (2008)
17. Schlegl, P., Robatzek, M., Kucera, M., Volbert, K., Waas, T.: Performance analysis of mobile radio for automatic control in smart grids. In: Second International Conference on Advances in Computing, Communication and Information Technology (CCIT 2014) (2014)
18. Wagn, Q., Zhu, Y., Cheng, L.: Reprogramming wireless sensor networks: challenges and approaches. IEEE Netw. **20**, 48–55 (2006)
19. Karp, R., Koutsoupias, E., Papadimitriou, C., Shenker, S.: Optimization problems in congestion control. In: Proceedings of FOCS 2000. IEEE Computer Society (2000)
20. Brown, S., Sreenan, C.J.: Software updating in wireless sensor networks: a survey and lacunae. J. Sens. Actuator Netw. **2**, 717–760 (2013). ISSN 2224–2708
21. Kenner, S., Volbert, K.: A low-power, tricky and very easy to use sensor network gateway architecture with application example. In: 10th International Conference on Sensor Technologies and Applications (SENSORCOMM 2016) (2016)
22. Lukovszki, T., Schindelhauer, C., Volbert, K.: Resource efficient maintenance of wireless network topologies. J. Univers. Comput. Sci. (J. UCS) **12**, 1292–1311 (2006)
23. Schwindl, T., Volbert, K., Bock, S.: Fast and reliable update protocols in WSNs during software development, testing and deployment. In: 7th International Conference on Sensor Networks (Sensornets 2018) (2018)
24. Stathopoulos, T., Heidemann, J., Estrin, D.: A remote code update mechanism for wireless sensor networks. Center for Embedded Networked Sensing (2003)
25. TI: CC430 family - user's guide (2013)
26. TI: MSP430^{TM} SoC with RF core (2013)
27. TI: MSP430F5510, MSP430F550X mixed-signal microcontrollers (2015)
28. TI: MSP gang programmer (MSP-GANG) (2017)
29. Jacobson, V.: Congestion avoidance and control. SIGCOMM Comput. Commun. Rev. **18**(4), 314–329 (1988)