Fast and Reliable Update Protocols in WSNs During Software Development, Testing and Deployment

Tobias Schwindl, Klaus Volbert and Sebastian Bock
Technical University of Applied Sciences Regensburg, Germany

Keywords: WSN, Software Update, Low-Power Devices.

Abstract: A lot of research has been done in the area of Wireless Sensor Networks during the past years. Today, Wireless Sensor Networks are in field in many different ways and applications (e.g. energy management services, heat and water billing, smoke detectors). Nevertheless, research and development is continued in this area. After the network is deployed, software updates are performed very rarely, but during development and testing one typical, high frequented task is to deploy a new firmware to thousands of nodes. In this paper, we consider such a software update for a special, but well-known and frequently used sensor network platform. There exist some interesting research papers about updating sensor nodes, but we have a special focus on the technical update process. In this context, we show the reasons why these existing update processes do not cover our challenges. Our goal is to allow a developer to update thousands of nodes reliably and very fast during development and testing. For this purpose, it is not so important to perform the best update with regard to energy consumption. We do not need a multi hop protocol, because all devices are in range, e.g., in a laboratory. In our work, we present a model of the update process and give very fast protocols to solve it. The results of our extensive simulations show that the developed protocols do a fast, scalable and reliable update.

1 INTRODUCTION

Wireless Sensor Networks are used in various environments. Some basics of such wireless networks are described in (Akyildiz et al., 2002), (Chong and Kumar, 2003), (Schindelhauer et al., 2007), (Lukovszki et al., 2006) and (Meyer auf der Heide et al., 2004). Special restrictions regarding power and time management in these systems are shown in (Sivrikaya and Yener, 2004) and (Sinha and Chandrakasan, 2001) Many example applications and its areas are shown in (Altmann et al., 2016; Kenner et al., 2016; Kenner and Volbert, 2016; Schlegl et al., 2014). Since most of these sensor networks are energy constrained one of the main goal of every application in such an environment is to secure that the lifetime of every node is as long as possible and simultaneously reach a satisfactory performance level. In specific cases, e.g., after production of devices, inital firmware must be programmed 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 programm and/or update all devices one by one, but instead with one wireless update run. In this laboratorylike scenario, the update process must ensure that the firmware is transmitted completely and without errors. All nodes should receive the firmware and this should happen automatically, meaning no user is required to update the nodes. As many wireless sensor nodes are battery driven one goal of most update processes is to reduce power consumption and simultaneously minimize the update time. Most update protocols use different approaches to ensure these points. As we show in the chapter 2 most update mechanisms do 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 introduce a new update protocol optimized for a specific use case. This use case is the programming of software on all used sensor nodes during the development process for a sensor network. At one point in the development process this means that sensor nodes need an update, to test new code or to extend the current code base to more devices. This includes the possibility that the firmware update is needed for a small amount of nodes, but at another point in the development process this can mean that a very large amount of nodes, i.e. thousands or tenthousands of nodes have to get an

Schwindl, T., Volbert, K. and Bock, S.
Fast and Reliable Update Protocols in WSNs During Software Development, Testing and Deployment.
DOI: 10.5220/0006534400190030
In *Proceedings of the 7th International Conference on Sensor Networks (SENSORNETS 2018)*, pages 19-30
ISBN: 978-989-758-284-4

update. Such sensor nodes, if no wireless update protocol is useable, are programmed with a simple hardware tool like the Gang-Programmer (TI, 2017) that can update a small number of nodes simultaneously. Such a device is used, e.g., in Germany to programm 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 program time of each and every node. Every time a new software is added 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 since the update as such 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 minimize the power consumption as much as possible, but there are possibly more sophisticated approaches to reach this goal (done in other protocols). The flawless and complete transmission of the new code are naturally important factors since the update protocol would otherwise not be reliable. As a result either manual intervention or many update runs would be needed to secure that all devices receive the new application. This said, such an update mechanism would not be very useful.

2 BACKGROUND & RELATED WORK

The problem of programming nodes in wireless sensor networks is discussed in many articles. A survey is given by (Brown and Sreenan, 2013) and in (Wagn et al., 2006), a smaller overview of some update mechanism is also shown in (Sternecker, 2012). There are various 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 (Stolikj et al., 2012) and (Rickenbach and Wattenhofer, 2008), try others to be a more generic solution for the process of software updates in WSNs. Many of these protocols, e.g., Deluge (Hui and Culler, 2004), MOAP (Stathopoulos et al., 2003) or Trickle (Levis et al., 2004) use the idea of a four step process (Brown and Sreenan, 2013) to ensure the needed functionality. This process is shown in Figure 1 and consists

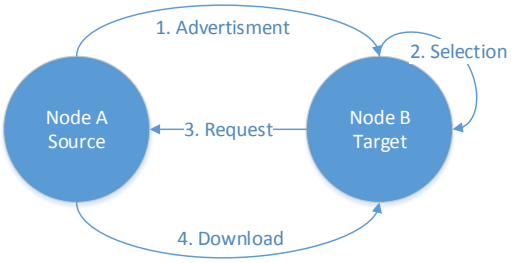

Figure 1: Dissemination idea for update protocols.

of 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 quality of the radio channel to the different possible sources (selection). These two steps include the idea of multiple sources and therefore multiple senders. At the same time these protocols often use the idea of multihop. This approach allows reaching nodes not only within the transmission area of one device, but to reach every node in a wide sensor network. Such multihop 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 optimal solution for some problems with appropriate hard- and software but does not fulfill 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, hence this allows us an easy mechanism to broadcast messages, because there is only one sender and therefore this message can not collide with other messages. There are simply no other messages at the same time in our complete system. The third step (request) initiates communication between source and target. The final step is the actual download of the requested data to update the target node. After all these steps are completed, the new software is executable on every node that received the update. Such sophisticated update models solve crucial problems to their special use case(s). But the additional steps of advertising and necessary following selection would need more time and energy in our scenario. Since we have a very dense network these steps do only include their disadvantages, nonetheless they would work with such networks. Though the advan-

tages of such an approach would be lost and therefore these kinds of methods do 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 do 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 basic ideas of the update protocol and give reasons for their use. The process is designed to update any amount of nodes in reasonable time. The update itself does only support full updates, i.e. there is no possibility to update a part of the firmware while keeping other parts of the software. The complete flash memory is reprogrammed with the new code. Since we designed and implemented the update process for a specific hardware from Texas Instruments (TI) we used existing software and ideas where possible, but nonetheless these ideas can be easily used for other systems as well. The already available TI 1:1 update process was the basis for our further development. 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 specific software on the sensor node, which will be updated, to work. Because of that, the update process is only one-time executeable. If the new loaded software does not support the specific sequence to launch and execute the update process, the complete process is not useable any longer, but must be manually flashed again. This kind of limitations were another reason for the development of a more practical update mechanism. The TI software, after the wireless update is started and initialization process is complete, does use a simple stop and wait method, meaning that after every single data packet an acknowledgement from the device is expected. If validation is positive the next data packet is ready for transmission. 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 currently 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 complete 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 there on all packets are transmitted. However, an error is not recognized immediatley, but only with the next acknowledgement. Since the process is designed to work with any amount of sensor nodes it must be ensured that all nodes know when to send their acknowledgement packet, otherwise some transmissions will fail due to interference with other possible transmissions. This is secured by the used time division multiple access (TDMA) mechanism. Every node has a specific, fixed time interval when to send the acknowledgement. The first easy idea and implementation uses the TDMA mechanism after a fixed amount of data packets independent whether the radio channel is of good or bad quality. 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 not very likely. An update protocol, which uses the information about the radio channel and its quality could save both time and energy since it decides flexible when an ack phase is needed more often. The adaption to the quality of the current radio channel is used in different 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 very rarely, but is increased by a multiplicative factor after an error occurred, originally shown in (Jacobson, 1988). A detailed analysis of the algorithms is presented in, e.g., (Edmonds, 2012) and (Karp et al., 2000). A similar, simplified mechanism is used by our second designed update protocol. The idea was to get a better adaption to the actual needed acknowledgement rate given by the physical quality of the radio channel. Mechanisms, messages and its sequences to initialize the update process are presented and analyzed in 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 every fixed time step and all nodes can synchronize with the system time with every data packet which is received. This ensures that every update device, if 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 saves a lot of energy since update nodes

are only in RX mode when it is absolutely mandatory. Due to 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 secures that RX windows remain narrow. This basic idea is valid for a lot of update scenarios. These ideas 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 specific hardware and software model on which we actually implemented and tested these update designs and protocols.

4 SOFTWARE & HARDWARE ARCHITECTURE

The software architecture and its distribution is based on the hardware devices used. Because the update protocol has several tasks, we operate with different devices, which are suited for their special 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 includes an integrated sub-1GHz wireless radio module based on the CC1101. It stores a bootloader (max. 2KB) which handles the complete update process on watch side. Therefore, the code size of the complete update protocol software must be less than 2KB, including 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 greater than the available bootloader ROM size. The eZ430 comes with 32KB of internal flash memory and 4KB of RAM (TI, 2013b). The microcontroller supports different internal sensors and offers different 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 with a CC1101 radio core (TI, 2015) which is used for communication with the watch. The access point does contain a USB interface for further communication with other USB devices. This device is responsible for dissemination of update messages, e.g., the new code, meaning it regulates the complete sequence of the update. The third 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. Communication between access point and GUI is handled via the USB interface. The complete structure of our update model is shown in Figure 2.

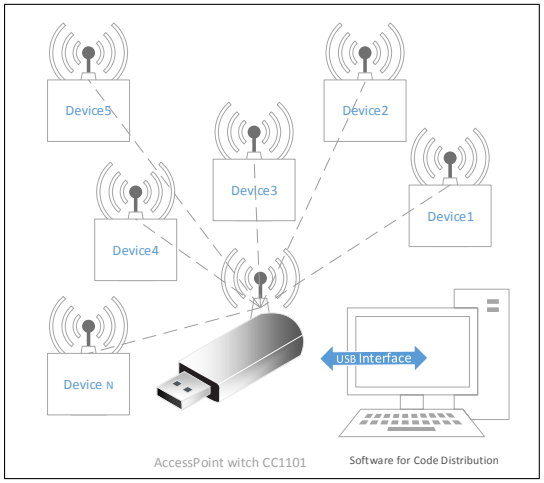

Figure 2: Overview of update model.

All watches are distributed in a small area within radio reach of the access point. But other than that, no additional requirements must be met. The update is started by the application and after that no user intervention is needed. The access point does now communicate with all watches in radio reach to update these nodes. In the design of the complete update protocol, communication between different sensor nodes, i.e. the CC430 devices, is not planned. From the update devices perspective it is a 1:1 communication with the access point. On the contrary the access point distributes code to all devices, hence this is a 1:n communication. The specific timings of messages and its sequences are presented in the next chapter.

5 ANALYSIS

In this chapter we analyze the update protocol and its different 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 possible and used messages in the system are

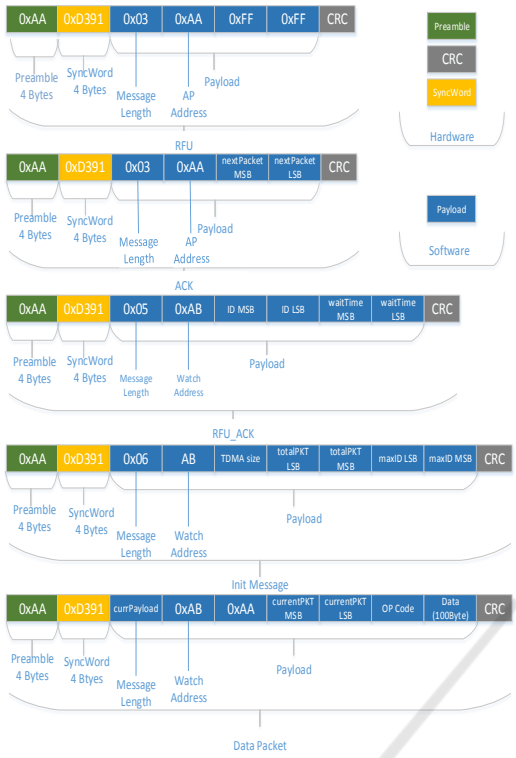

Figure 3: Protocol messages in the updatesystem.

shown in Figure 3. As seen in the protocol messages, the hardware settings of the radio communication include 4 byte preamble and 4 byte sync word. The preamble is an alternating sequence of ones and zeros, i.e. 0xAA is transmitted. The sync word contains application specific data and is used for byte synchronization. Additionally, it allows a differentiation between systems with the same hardware as the radio writes only data to the internal buffer if the correct synchronization word is received (TI, 2013a). In our application it is used with the value of 0xD391, which is used twice to get a 4 byte sync word. The 16 bit checksum calculation is enabled, therefore it is not necessary to check for errors in software as corrupted packets are removed automatically. The data packets data field is flexible, meaning during the update run the access point could decide to increase or decrease the payload length. This mechanism is currently not used, but could be used in future versions. All other messages have fixed length and can not change its size during protocol execution.

The exact sequence of the complete update process is shown in Figure 4. To start the firmware update mechanism all watches must execute its bootloader software. This can be done via reset of the device either manually or by triggering a software reset, if 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 started the bootloader, the update devices now transmit every specific time interval, e.g., 3 seconds, the RFU (ReadyForUpdate) message. If the access point receives such a message (this message comes from a watch which has not an update ID and therefore was not recognized yet) it answers with the RFU_ACK message and sets update ID for this watch and update run. The frequency of the RFU message determines how often the watch can send its beacon during discovery phase of the update, e.g., if an error occurred or some other watch is sending simultaneously and another RFU message is needed. To prevent most of the colliding messages in the discovery phase a hardware mechanism is used. The discovery phase is the only state where collision between different messages is possible. In all other states there is only one message at a time in the complete system. To save energy after this short communication the watch does now go into LPM3. After a specific, by the user configurable amount of time the discovery phase is finished and the update goes into next state, i.e. init state. At this point all watches that need an update should be recognized by the access point. This is ensured by setting an appropriate time value for the watches to be in the discovery update state. All watches wake up at the same time, very shortly before the first actual packet is sent and now listen to the first update package, i.e. init packet. This is possible due to time information the devices have from the previous communication with the access point, which is included in the RFU_ACK message. The init packet does contain some information about update size, number of update devices and some information about the acknowledgement phase and its length. If during this process a watch does not get an answer for the RFU message or does not get the init message correctly this device does not participate in the following update process, rather resets itself. After this init mechanism for every node is successfully completed, the actual update starts. In this case an adjustable, but during the update process fixed, number of data packages is transmitted before the first acknowledgement phase is started. All packets come in specific time areas, so all watches can go into RX mode very shortly before the actual packet transmission starts.

This ensures that a very small amount of power is needed, since RX time is not longer than absolutely needed given by physical parameters plus calibration time. The time between data packets is 100ms and 5 packets are sent in a row before starting with the acknowledgement phase. That said, the data packet

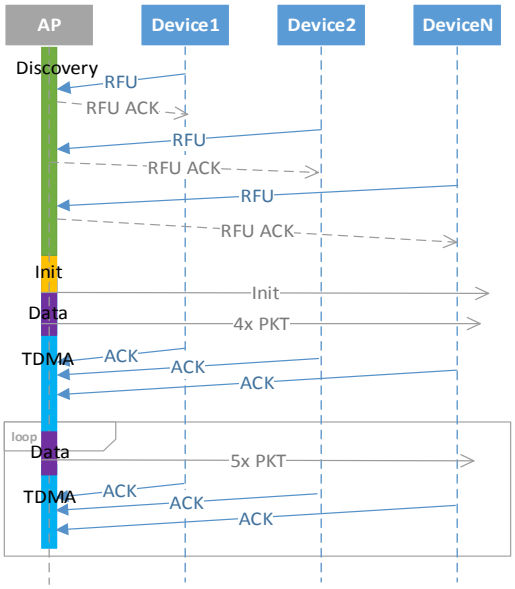

Figure 4: Protocol schedule during the update.

round is 500ms long. After the data packets are received, meaning 500ms packet round with 5 sent packets is over, the update devices, whether they received packets successfully or not, trigger the next and last state, the TDMA phase. If an error occurred at the beginning of the data phase all following packets are lost as well. This is due to the design of the protocol, an update device is only able 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 future versions to improve the update. The complete TDMA window is divided into small time windows for each and every node. The length of this phase is dependent on the number of sensor nodes, which participate in the update run. Every update node has the chance to send an acknowledgement packet which includes the packet number that is expected next, therefore, this information contains the last successfully received packet number. After this TDMA mechanism the access point has all needed data to decide which packet should be transmitted next, i.e. the smallest packet number that was received by the access point. If an ack from a node is lost, for whatever reason, there is no error handling. Next time the ack phase starts there is another chance for this node to send its ack successfully. After every TDMA, the data phase is started again. This process is repeated until no more acks with a packet number below the highest possible packet number is received, therefore all nodes have the complete new code. Every node that has received all needed 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 start with sending the RFU message. This process can be aborted by special input to easily start the application. The acknowledgement phase is the most complex phase of the complete update protocol. The more devices are participating in one update run, the longer this phase must be. The ack phase is always determined by a specific integer factor of this 100ms, dependent on how many devices need an update. This factor does increase by 1 every 10 nodes. The decision to set these specific timing intervals were made after evaluating the USB communication between the access point and the pc application.

5.2 Calculations – Power Consumption

Since 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. Power consumption of the update process is split 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 different power levels of these states in mA. To ensure low power consumption, the completion time as well as radio transmissions should be minimized and update time should be spent in LPM3 whenever possible. The latter can be reached by an exact time based protocol as we have designed. 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.

Table 1: Power consumption @12MHz (TI, 2013b).

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

Power Consumption 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 neglectable since its not battery driven, but 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 useable 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 250'000 bits per sec-

Table 2: Power consumption of different messages.

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

ond transmission rate. Since the radio module needs calibration every time a communication is initialized a value of 721μs with a power consumption of 9.5mA must be added to every receive or transmit operation (TI, 2013b). This results in additional $1.9 \cdot 10^{-9}$Ah per radio event. Now we know all needed values to compute the complete power consumption of the CPU plus radio module during one update run. Following formula shows the power consumption for the radio module for all transmissions:

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

where a is the number of RFU messages the watch sends, b the number of data packets were received by the watch and c the number of acks the device is sending during the update run. Since 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 and after a data packet was received, because this data must be written to internal flash memory. So 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 100ms 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 10ms and this would be the time value 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

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

During an update run the complete main flash memory of the device is erased and reprogrammed. These memory operations also need time and energy. Timings and power consumptions of all flash operations are shown in (TI, 2013b). We calculate the power consumption during erase with the given typical value of 2mA and during programing with 3mA. While full erasing does need maximal 32ms, the complete programing time of the CC430 flash memory takes about 800ms of active write operation. All flash instructions summorized 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 and erase operations on the flash memory give now the complete power consumption needed 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 27KB, a data packet payload length of 100Bytes and a TDMA window after every 5 data packets, can now be calculated. The minimum amount of packets needed 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 27s, i.e. the 270 packets with one packet in 100ms, plus discovery phase length which is calculated with 7s – 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. As a result a total amount of $9.33 \cdot 10^{-6}$ Ah is used by one update run, if exactly one device is involved. If more devices are used power consumption increases, but only because the time in LPM3 goes up. 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 (CR2,). This results in about 23'500 possible best case update runs before the battery is empty. This should be sufficient 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 goal during 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, that an update run needs can be calculated as follows:

$$\text{completionTime} = \text{Time}_{\text{discoveryPhase}} + \text{Time}_{\text{dataPackets}} + \text{Time}_{\text{TDMA/ACK phase}}$$

The user currently sets 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 is dependent on the update size. The useable flash memory of the device is 32KB, which results currently, since the data payload of one packet is 100 bytes, in maximal 32 seconds. The TDMA/ACK phase of the update is calculated differently and dependent on number of sensor nodes which need an update. The first 10 devices can use the normal time between 2 data packets to send their acknowledgements, but after that every 10 devices get an additional 100ms time slot to send 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 number of update devices and becomes the most important factor that determines how long an update run needs to finish. With, e.g., 1000 devices and an update size of 27KB time spent in the TDMA phase is about 535 seconds, while the actual sending of the packets is 27 seconds. Thus, in later improvements of the update protocol the time spent in the TDMA windows should be reduced to get faster update results. The time to finish the update depends huge 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 entirely different. If an error occurs directly after the ACK phase, the complete packet round is useless 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. With the example of 1000 Devices, 3 more TDMA phases would mean: $99 \cdot 100ms \cdot 3 = 29,7s$ more time to finish the update for all devices. Thus, the exact time/place of the errors has a huge impact on the completion time of the update. Furthermore, with more devices an error becomes more and more impactful 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 specifically want to show our experiments we made. For all tests of the update process we considered all available nodes which need an update are in very short distance to the access point, i.e. max distance was 1m 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 completion time. Another top priority goal is reliability. All devices must be updated completely and without errors. The energy consumption in this case has a lower priority, but nonetheless is always important in wireless sensor networks. Otherwise the battery must be replaced often and therefore most benefits of an update process, e.g., fast and simple testing of new code would be lost.
Calculated best case results for updates are shown in Figure 5. The x-achsis (bottom) shows the number of devices used whereas on the y-achsis (left side) the time to update this number of devices is shown. The diagram is valid for an update size of 27KB.

The fixed parameters update process has an acknowledgement rate of 5. This means that every 5

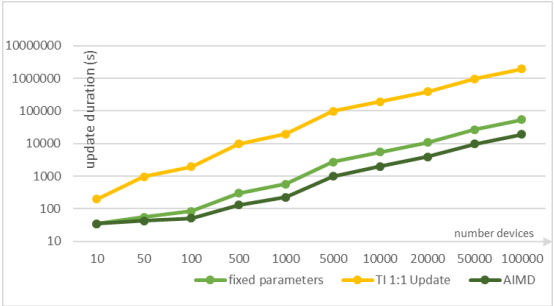

Figure 5: Update duration with different number of devices.

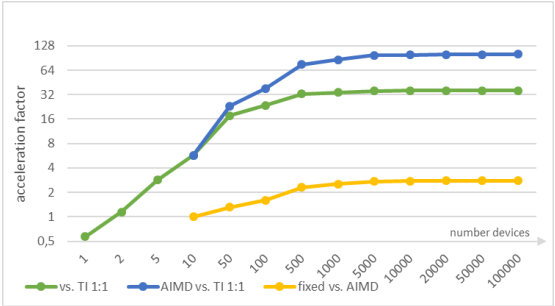

Figure 6: Acceleration between the different update modes.

data packets there is a time window where all watches can send its ack packet. Since the number of ack phases can be reduced, if the radio channel is good and an error is not very likely we used a different approach for the ack phase. The AIMD and AIAD variant introduce easy mechanisms to react flexible to the current radio channel and adapt the ack rate. AIMD starts at the same ack rate and from there on the algorithm decides whether to increase or decrease the ack rate. Discovery phase was set to 7 seconds during all experiments. This was sufficient for our number of available sensor nodes, but must be adjusted accordingly if more nodes are updated simultaneously. The acceleration factor between different update modes (TI 1:1, fixed, AIMD) is shown in Figure 6. Since 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. This is due to the acknowledgement phase of the update protocol. As shown before in 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 in a row. If 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 of time. These diagrams always show the best case, which means that no error occurs during the complete 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 reach these values.

6.1 Results – Static Acknowledgement Rate

Table 3 shows some actual results from experiments we made. These first results were executed for all nodes within a small area and equipped with the fixed parameters software. Onehundred of each test runs, i.e. in total 500 update runs were executed with different amount of sensor nodes. The table shows the number of devices we used for one update run (up to 25), the time the experimental worst case run was longer than the best case calculation and the average and worst case PER (packet error rate). The complete column, i.e. the rate of successful updates gives an overview of how many of these firmware updates were done completly with no errors, meaning a runnable new software is stored in flash memory. PER is the packet error rate which indicates the channelquality between access point and watches during the software update. These experimental results show,

Table 3: Measurements – static variant.

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

that completion rate does not correlate with number 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 current implementation an error can occur in the last data round, the acknowledgement gets lost and therefore the update device can not finish 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 1cm, 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 pack-

ets. The power consumption does increase over time since more CPU and radio activity is needed. The calculation results shown in 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 time to complete 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 how long the update needs, as shown in the previous section in 5.3. Time 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. Such a 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 200ms with 25 devices instead of 100ms with 20 devices. This also means, that for a huge amount of nodes an error is much more impactful than for a small amount of devices.
Another 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 increase the amount of packets needed 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 detected 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 & 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 adapt are the AIMD and AIAD mechanisms. The initialization process and the message types are the same as before. Only difference to the previous mechanism is the rate at which the watches transmit their acknowledgement information. The acknowledgement rate starts again at 5, which means after 5 data packets the first TDMA is initialized. After this first ack phase the access point adjusts 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 and AIAD mechanism is 38 seconds. This means that every single packet is transmitted without an error.

Table 4: Measurements – AIMD/AIAD variant.

	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

This includes both data packets and ack packets. A non received positive ack, meaning an ack that would confirm that no error occurred is handled 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 clearly higher. The average case still shows that the quality of the radio channel is sufficient to reach better completion times than the normal static mechanism. The best case of the first approach lies with 45 seconds 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 seconds and with the AIAD 57.9 seconds. The worst case of the static method is with 52.2 seconds 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 needed. These algorithms have a delay to react to both error cases and because of that it 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 section 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 overall. 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 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 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 & FURTHER WORK

In this paper, we present a fast, reliable but still simple update mechanism plus first improvements and their evaluation. Such an update protocol shows an easy way to update a large amount of sensor nodes simultaneously while keeping its effort manageable. This approach could be used with a variety of different hardware. During the development of software for the sensor network, the co-development of a wireless updater should be top priority since the benefit from such an update process is huge. The easy possibility to update all currently needed nodes at the same time significantly decreases the amount of time needed to reprogram all sensor nodes and therefore, it is easy to test new code very fast. An exactly shared time based protocol is much less complex than update mechanisms that come with multiple senders and/or multihop concepts. Calculations of this protocol shows that the update can be executed multiple times, the power consumptions is not problematic. The execution time is much smaller than in a 1:1 update/flash scenario and its scalability means it is useable with any amount of nodes, whether a small or a large number is used. Some improvements to the update protocol could be done by analyzing the radio channel so the ack phase frequency is reduced. Another possibility to improve the update is not only to make the ACK phase flexible, but also the size, i.e. the payload, of the packets. With a good radio channel and a 1.5 times as large packet size, the TDMA frequence would be reduced by this factor. This would result in higher throughput which leads to an even smaller execution time for the update. It would be interesting to compare some existing update protocols with our developed solution. Especially those mechanisms, which did not provide a real implementation. We plan to implement these already developed update ideas on our hardware platform to get experiment results from these implementations.

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

DataSheet Lithium Manganese Dioxide Battery CR2032. accessed 15.03.2017.

Akyildiz, I., Su, W., Sankarasubramaniam, Y., and Cayirci, E. (2002). A survey on sensor networks. *IEEE Communications Magazine*.

Altmann, M., Schlegl, P., and Volbert, K. (2016). A low-power wireless system for energy consumption analysis at mains sockets. *EURASIP - Embedded Systems*.

Brown, S. and Sreenan, C. J. (2013). Software updating in wireless sensor networks: A asurvey and lacunae. *Journal of Sensor and Actuator Networks, ISSN 2224-2708*. www.mdpi.com/journal/jsan/.

Chong, C.-Y. and Kumar, S. (2003). Sensor networks: evolution, opportunities, and challenges. *Proceedings of the IEEE*.

Edmonds, J. (2012). On the competitiveness of aimd-tcp within a general network. *Theoretical Computer Science*.

Hui, J. W. and Culler, D. (2004). The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at scale.

Jacobson, V. (1988). Congestion avoidance and control. *SIGCOMM Comput. Commun. Rev.*, 18(4):314–329.

Karp, R., Koutsoupias, E., Papadimitriou, C., and Shenker, S. (2000). Optimization problems in congestion control. *Proceedings of FOCS00, IEEE Computer Society*.

Kenner, S., Thaler, R., Kucera, M., Volbert, K., and Waas, T. (2016). Comparison of Smart Grid architectures for monitoring and analyzing power grid data via Modbus and REST. *EURASIP - Embedded Systems*.

Kenner, S. and Volbert, K. (2016). A low-power, tricky and very easy to use sensor network gateway architecture with application example. *10th International Conference on Sensor Technologies and Applications (SENSORCOMM'16)*.

Levis, P., Patel, N., Culler, D., and Shenker, S. (2004). Trickle: A Self-Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks.

Lukovszki, T., Schindelhauer, C., and Volbert, K. (2006). Resource Efficient Maintenance of Wireless Network Topologies. *Journal of Universal Computer Science (J.UCS'06)*. 12(9):1292–1311.

Meyer auf der Heide, F., Schindelhauer, C., Volbert, K., and Grünewald, M. (2004). Congestion, dilation, and energy in radio networks. *Theory of Computing Systems (TOCS'04)*.

Rickenbach, P. and Wattenhofer, R. (2008). Decoding Code on a Sensor Node. *4th International Conference on Distributed Computing in Sensor Systems (DCOSS)*.

Schindelhauer, C., Volbert, K., and Ziegler, M. (2007). Geometric Spanners with Applications in Wireless Networks. *Computational Geometry: Theory and Applications (CGTA'07)*. 36(3):197–214.

Schlegl, P., Robatzek, M., Kucera, M., Volbert, K., and Waas, T. (2014). Performance Analysis of Mobile Radio for Automatic Control in Smart Grids. *Second International Conference on Advances in Computing, Communication and Information Technology (CCIT'14)*.

Sinha, A. and Chandrakasan, A. (2001). Dynamic power management in wireless sensor networks. *IEEE Design & Test of Computers*.

Sivrikaya, F. and Yener, B. (2004). Time synchronization in sensor networks: a survey. *IEEE Network*.

Stathopoulos, T., Heidemann, J., and Estrin, D. (2003). A Remote Code Update Mechanism for Wireless Sensor Networks. *Center for Embedded Networked Sensing*.

Sternecker, C. (2012). Reprogrammierungstechniken fuer drahtlose Sensornetzwerke. *Seminar Sensorknoten - Betrieb, Netze und Anwendungen*.

Stolikj, M., Cuijpers, P. J. L., and Lukkien, J. J. (2012). Efficient reprogramming of wireless sensor networks using incremental updates and data compression. *Department of Mathematics and Computer Science System Architeture and Networking Group*.

TI (2013a). *CC430 Family - User's Guide*. accessed 15.03.2017.

TI (2013b). *MSP430 SoC With RF Core*. accessed 15.03.2017.

TI (2015). *MSP430F5510, MSP430F550x Mixed-Signal Microcontrollers*. accessed 15.03.2017.

TI (2017). *MSP Gang Programmer (MSP-GANG)*. accessed 20.06.2017.

Wagn, Q., Zhu, Y., and Cheng, L. (2006). Reprogramming Wireless Sensor Networks: Challenges and Approaches. *IEEE Network*.