Fail-Safe Over-The-Air Programming and Error Recovery in Wireless Networks

Stefan Unterschütz
Hamburg University of Technology
Hamburg, Germany
stefan.unterschuetz@tu-harburg.de

Volker Turau
Hamburg University of Technology
Hamburg, Germany
turau@tu-harburg.de

***Abstract*—Wireless networks are an emerging technology which are employed for a growing number of applications. Maintenance and extensibility necessitate software updates after an initial deployment. Wireless updates are in most cases preferred, e.g. in large-scale networks or inaccessible deployments. Recent research either focused on reliable and fast transmission of a new firmware, or on a modular update of firmware parts. In this paper we additionally address the problem of error-prone software which can permanently disable the update functionality. A comprehensive and fail-safe update system is proposed fulfilling requirements for the usage in industrial environments.**

I. Introduction

A crucial operation in wireless networks is reprogramming of a new firmware using the radio communication. Wireless nodes are often deployed in environments that are difficult to access. A wired firmware update would lead to high costs or may be even infeasable. In these scenarios, the use of radio communication for updating software, which is commonly known as over-the-air programming (OTAP), is recommend.

The subject of wireless software updates can be divided in three independent activities. First, a secure, reliable, and rapid, i.e. scalable, dissemination of a firmware image is required. Once the new software is completely received, a fail-safe reprogramming must be carried out. This is made more difficult, if a simultaneous update of all nodes is required, e.g. protocols of a new firmware are incompatible with old ones. Finally, to cope with erroneous software, a recovery activity becomes necessary. This is crucial since even extensive software tests do not guarantee that a program will work correctly in the target environment. A faulty program rendering the update system unusable causes high maintenance costs. Unfortunately, this issue is only insufficiently analyzed in recent work. Furthermore, to our best knowledge there is no existing work, which takes all three activities into account.

The contribution of this work is a concept of a fail-safe, over-the-air programming system which can automatically recover from error-prone software. The system is implemented for resource-restricted hardware, i.e. an 8-bit AVR microcontroller, and evaluated in a testbed deployment for controlling a heliostat power plant [1].

This paper is structured as follows: Section II gives an overview of recent protocols for OTAP and reveals limitation of thus. Subsequently, the design of the proposed OTAP system is introduced. In Sect. IV the firmware deployment procedure, and in Sect. V the bootloader and recovery process are described in detail. Based on a testbed deployment, we assess in Sect. VI the whole system regarding reliability and scalability. Finally, the paper ends with a short conclusion.

II. Related Work

Deluge [2] is the default firmware propagation protocol for TinyOS. A firmware (called data object) is split into pages. A page itself is further separated into transmittable packets. Each firmware is labeled with a version number. Each node advertises from time to time via broadcast its version number. Neighbors can request specific pages of the firmware, which are then transmitted. Once a complete page is received, this can be forwarded to further nodes. This technique provides pipelining and fast data dissemination. Deluge scales well in large networks and supports secure operation [3].

Deluge coexists with the TinyOS bootloader (TOSBoot) allowing to flash a firmware. In case of a non-working firmware, a user can manually force (by 5 times rebooting) the use of a *golden image*, which provides basic functionality. However, a damaged firmware can not wirelessly be reset. Furthermore, Deluge is completely distributed, which makes a global reboot or validation of the current firmware difficult. However, the last two requirements are mandatory for industrial applications.

Contiki [4], a lightweight and flexible operating system for tiny networked sensors, and Lorien [5], a component-based modular operating environment, support loading and flashing of parts of the firmware at runtime. For this purpose they use a run-time relocation and a linkable format for transmitting the binary data. The advantage is that only parts of the firmware must be transmitted. In Contiki and Lorien all modules and functions share the same address space, thus an invalid software part can render the whole system unusable.

Brown and Sreenan address the problem of error-prone software and suggest a periodic beaconing in order to ensure communication ability [6]. For this purpose they mainly analyzed two broadcasting protocols in terms of detecting connectivity loss. They did not propose an actual technique to recover from arbitrary software faults.

A survey of OTAP protocols can be found in [7]. This also includes approaches to reduce the size of firmware images based on compression or the calculation of differences to previous software. This topic is not addressed in this work.

III. Architecture

In this section the design and general functionality of our OTAP system is introduced including the activities dissemination, flashing, and recovery of firmware images. It was developed in accordance with the requirements of a heliostat power plant: Scalability (important for large-scale networks), fail-safety, reliability, and applicability for resource-restricted hardware. The latter is crucial since the OTAP system had to be implemented for Atmel's ATmega128RFA1 with embedded IEEE 802.15.4 compliant transceiver. The 8-Bit AVR microcontroller is equipped with 16 KiB SRAM, 128 KiB Flash, 16 MHz clock, and 128 KiB external and 4 KiB internal EEPROM. Note that the proposed OTAP system meets the requirements of most wireless networks, e.g. wireless sensor networks, it can thus be easily reused for these, too.

Most approaches for over-the-air programming consider only faults during the transmission, persistent storing or flashing of the firmware. In this context, fail-safety means also to recover from any software fault as well as from any transient hardware fault. An example for the latter is that a node runs out of energy, but restarts later if the hardware is powered up by e.g. energy harvesters. To cope with this, each step of the update procedure should be interruptible, meaning that no invalid state is reached. Dealing with software faults, i.e. programming errors, is more demanding, because this includes, among others, unwanted infinity loops, memory corruption, non-working communication stacks, disabled interrupts.

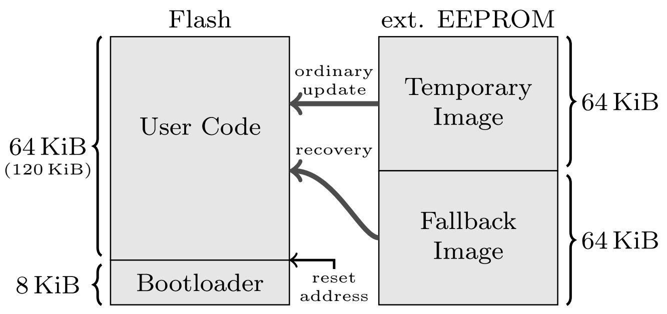


Fig. 1: Persistent Memory Partition

The partition of the persistent memory for the OTAP system is shown in Fig. 1. Machine code can only be executed from the Flash memory. For this, the AVR microcontroller allows a separation in a bootloader section and and an application section. Since the former is limited to 8 KiB, which is insufficient to include all required protocols for OTAP (e.g. routing), the bootloader implements only parts of the flashing and recovery functionality as well as the start-up procedure after reset. The application section of the flash memory contains the user code. The user code must include all needed protocols for dissemination and storing of data. Two firmware images are stored in the external EEPROM: One is used as recovery (fallback) image in case of hardware or software faults. The other memory block holds a temporary image, which is used for flashing new software. For our hardware platform the EEPROM size of 128 KiB limits the maximum firmware image size to at most 64 KiB. If more persistent memory is available, the storage of additional images will be possible. This allows to switch between different firmware versions.

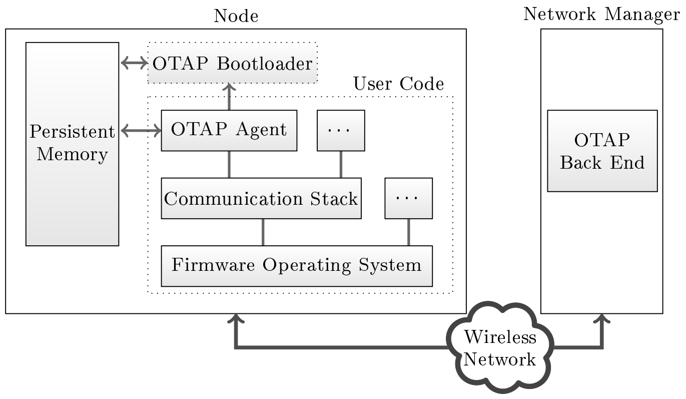


Fig. 2: General Architecture

The design of the OTAP system is depicted in Fig. 2. The update process is centralized controlled by a network manager. At least one gateway node must be connected with the network manager in order to grant access to the whole network. The software, running on each node, is depicted in the left block of Fig. 2. It is mandatory to integrate the OTAP Agent to the user code and to connect it with a working communication stack. Further arrangements are not needed, in particular the compilation and linking of source code can be done as usual. The OTAP Agent is able to write data to the EEPROM and collaborates with the bootloader.

The OTAP Agent demands the following functionality from the user's communication stack: reliable and bidirectional one-to-one communication (network manager to node and vice versa), node-to-many communication (network manager to nodes). The first type is used for sending and receiving control messages, e.g. to restart and flash a new firmware. For scalability reasons, one-to-many communication is needed to disseminate the firmware image to all nodes. High reliability is not mandatory for the one-to-many communication. However, Sect. IV-C describes a technique to increase reliability. Note that the OTAP Agent does not rely on a particular communication stack. For instance, if no multi-hop communication is desired, a (wired) serial connection as well as a CSMA/CA protocol for 1-hop networks can be used. The OTAP Agent uses 16-Bit CRC checksums to validate firmware images, but does not provide security, i.e. confidentiality, authenticity. The latter must be provided by the user's communication stack.

IV. Firmware Deployment

A. *Firmware separation*

A firmware image is equally split into segments. Each segment is further split into m packets. The packet size is in most cases limited by the used wireless technology, e.g. approximately 100 Byte for 802.15.4. However, the use of smaller packets might be reasonable in networks with low link qualities. The used segmentation of the firmware image has several reasons. Firstly, retransmission of missing data takes place on segment level. For this, each nodes stores in a bit vector the number of received segments. This bit vector fits exactly in one packet and is used to request missing segments. Storing the state of individual packets would lead to a higher memory demand. Furthermore, we use forward error correction on packet-level for each segment transmission (see Sect. IV-C). Note that the complete firmware must be given in an executable, binary format. Relocation (linking) of code on the node side is currently not supported.

B. Dissemination

The dissemination process is supervised, i.e. controlled, by the network manager. The user passes a firmware image and, optionally if no network-wide dissemination is desired, a list of node identifiers to the network manager. Before sending any firmware data, the network manager sends a preparation request to the intended receivers. Only these selected node will store received firmware data. Now, the network manager generates the segments and packets and starts transmitting the data via one-to-many communication. The selected nodes are storing all data in the EEPROM section for the temporary image. After the transmission, the network manager inquires each node for missing segments and immediately retransmits outstanding data. The retransmission is done via one-to-many communication, allowing other nodes that miss the same segment to overhear. In case of highly error-prone environments the network manager can also use the reliable one-to-one routing protocol for the dissemination process. Finally, the network manager can force single nodes to flash data at a specific time. To ensure integrity of the firmware, stored in the EEPROM, a CRC-16 is transmitted and used for verification.

In comparison to existing data dissemination protocols, e.g. Deluge, we exploited a centralized approach. This reduces the scalability. However, since one-to-one communication is only used for control traffic, and the bulk of data is transmitted via one-to-many, the system is still efficient. The main reason for the centralized approach is that a feedback of each single node, whether the firmware update succeed, is required by most applications such as heliostat plant control. This approach further benefits from the possibility for an easy modification and adaption of the dissemination procedure on the network manager based on the network demands (e.g., packet rate, use of error correction). Furthermore, the complexity of the OTAP Agent on the nodes is kept low. Note that the pure dissemination of the firmware image can also be realized in a distribute fashion for our system, e.g. by using Deluge.

As mentioned the OTAP system doesn't rely on specific communication protocols. For simplicity, we use flooding for the data dissemination in the testbed deployment. Since pure flooding leads to a high collision probability in dense networks (cf. [8]), we alternatively tested algorithms based on a connected-dominating-set and counter-based flooding. However, the node deployment must be taken into consideration, when selecting an appropriate communication stack. Empirical experiments indicate that especially in sparse networks, pure flooding is the best choice. For the one-to-one communication we use a fault-tolerant tree-source routing algorithm.

We tried to maintain a flexible and modular approach for the whole firmware deployment procedure. Single components of the system can be exchanged against more efficient implementations. For example, the performance of the data dissemination can be improved by applying forward error correction codes. The fundamentals of this are described in the following.

C. Forward Error Correction

One-to-many communication, e.g. flooding, relies on broadcasting and is inherently unreliable. Acknowledgments on segment or even packet level for all nodes would lead to a high overhead. For this reason, we use Reed-Solomon (RS) codes for the segment transmission: In addition to the m packets of each segment, k redundant packets are disseminated. Now, a segment can be completely reconstructed, if at least m packets out of the $m + k$ packets are received. Note that redundancy is spread over packets rather than appended to each single packet. There are several reasons for this: no modification of the physical or MAC layer is necessary, robustness against burst-errors since only one packet is affected, decoding can be done very efficient on packet level, and no error detection is necessary which doubles error correction ability. The latter is due to the fact that corrupt packets are already filtered out by the used MAC layer.

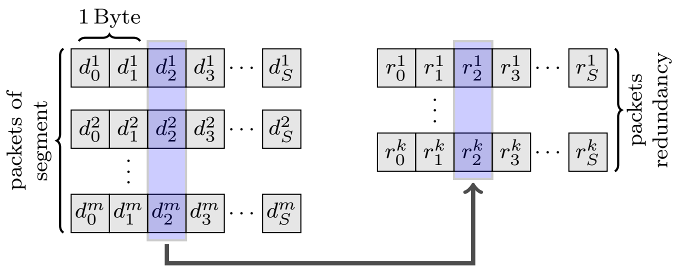

1 Byte
packets of segment
d_0^1 d_1^1 d_2^1 d_3^1 $\cdots$ d_S^1
d_0^2 d_1^2 d_2^2 d_3^2 $\cdots$ d_S^2
d_0^m d_1^m d_2^m d_3^m $\cdots$ d_S^m
r_0^1 r_1^1 r_2^1 r_3^1 $\cdots$ r_S^1
r_0^k r_1^k r_2^k r_3^k $\cdots$ r_S^k
packets redundancy

Fig. 3: Reed Solomon Coding of Packets

Encoding and decoding is done on symbols of length 4-bit. Each packet is subdivided in S nibbles (each 4 bit). For each nibble position of the m packets, redundant nibbles are generated building up k redundant packets. Figure 3 shows this approach. Note that the number of redundant packets can be adapted at runtime, without any modification of the code. The underlying mathematics for one operation are briefly sketched in the following.

The encoding and decoding of symbols can be expressed as a system of linear equations. The structure for encoding is shown in (1). The vector $\mathbf{d}$ (containing m symbols) is multiplied with the matrix $\mathbf{A}$ yielding $\mathbf{c}$. The vector $\mathbf{c}$ consists of the original vector $\mathbf{d}$ and k additional, redundant symbols (vector $\mathbf{r}$). For decoding, any m elements of $\mathbf{c}$ are sufficient, because the corresponding matrix $\bar{\mathbf{A}}$ (containing m lines of $\mathbf{A}$) is regular due to its structure. The reconstruction of the original vector $\mathbf{d}$ is done using the inverse matrix $\bar{\mathbf{A}}^{-1}$. All operations, including Gaussian elimination, have to be done in the Galois field $GF(2^4)$. We used lookup tables to increase the execution time. However, we refrain from a precalculation of the inverse matrices, because for large values of k and m the number of possible inverse matrices is $\binom{m+k}{m}$ and thus not storable in the SRAM. A good summary of RS coding with focus on the implementation is given by James Plank [9].

$$\overbrace{\begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \dots & 1 \\ \\ 1 & 2^0 & \dots & m^0 \\ 1 & 2^1 & \dots & m^1 \\ \vdots & & & \vdots \\ 1 & 2^{k-1} & \dots & m^{k-1} \end{pmatrix}}^{\mathbf{A}} \cdot \overbrace{\begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ d_m \end{pmatrix}}^{\mathbf{d}} = \overbrace{\begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ d_m \\ \\ r_1 \\ \vdots \\ r_k \end{pmatrix}}^{\mathbf{c}} \qquad (1)$$

Superficially, Reed-Solomon coding decreases throughput due to the need of additional packets, but significantly increases the probability of a successful reception of a segment. For example, let $p = 0.80$ be the probability of successfully receiving one packet. If a segment is separated in $m = 4$ data packets and $k = 3$ redundant packets than the probability for receiving a segment is $\sum_{i=4}^{7} \binom{7}{i} 0.8^i (1 - 0.8)^{7-i} \approx 0.97$. Without redundancy the probability to successfully receive a whole segment is $0.8^4 \approx 0.41$.

V. Bootloader and Recovery

In order to run the bootloader and to ensure recovery from arbitrary software faults the following settings are necessary: The hardware watchdog is permanently activated and cannot be deactivated by software, modification of the bootloader section of the Flash is locked, the reset address points to the bootloader section; hence, the actual bootloader can never be exchanged wirelessly once deployed and should be therefore carefully tested. Note that the used AVR microcontroller had a considerable bearing on the design of the bootloader, however, most of the following concepts can be adopted by other platforms, too.

A. Basic Bootloader Operations

On startup, the OTAP system runs a small bootloader. Firstly, this program reads an operation code from the internal EEPROM. For example, the OTAP Agent, which is part of the user code, may have stored the code for flashing a new firmware. Naturally, the programming of the flash memory is verified by the bootloader and will be repeated if needed. The end of an demanded operation is sealed by storing a no operation (NOP) code. With respect to the resilience, in the bootloader's implementation we keep care that operations are interruptible by hardware failures, e.g. energy depletion in solar powered devices, and will not lead to an inconsistent state. This is achieved by storing additional verification codes in the EEPROM in order to detect failed operations.

If the operation code, read by the bootloader, is invalid or contains the NOP code, an unexpected reset of the microcontroller must be assumed. Different policies to cope with this situation are possible. We decided to program the fallback image if not already done. This technique collaborates with the recovery procedure described in Sect. V-B and leads to a high robustness. Since the write cycles of the persistent memory are limited, once the fallback is loaded further unexpected resets do not lead to a modification of the persistent memory.

Communication protocols for the firmware deployment are not part of the bootloader due to the size limitation of the bootloader section. Thus, the OTAP Agent which is connected with a suitable communication stack is responsible for receiving packets. Since these components are part of the user code, they can be updated. Consequently, the data dissemination procedure can be easily optimized.

A replacement of the fallback image is possible, but should be done with caution. Above all, a fallback candidate should already run at a reasonable period in the network without any error. The bootloader can then be forced to override the fallback image with the current image stored in the Flash memory. An unexpected reset, e.g. during overriding the fallback, is nonhazardous, because the bootloader will unyieldingly try to finish this operations after restart.

B. Fault Detection and Recovery

Next to reprogramming, the bootloader further implements an error detection mechanism which allows to recover from any software fault. For this purpose, a part of the bootloader stays active during the execution of the user code. Additionally, the network manager periodically sends beacons. These are received by the OTAP Agent but checked for validity by the bootloader. Missing or invalid beacons lead to a reset and reprogram. Thus, connectivity loss is used as indicator for a *possible* software failure. In any case, the user can force a switch to the fallback by refraining from sending beacons.

In order to run the bootloader code next to the application (which is unaware of this), the bootloader modifies the interrupt vector table of the user code. The bootloader occupies a hardware timer which becomes unavailable for the application.

The use of the timer is twofold. On the one hand the hardware watchdog is reset and on the other hand the last received beacon is checked. For the latter, the required communication with the OTAP Agent is realized via shared memory. The OTAP Agent allocates on startup memory and writes the address into the internal EEPROM (if not already done). This indirection is necessary since the bootloader, once the user application runs, is not allowed to access static memory.

The used hardware watchdog has to fulfill an important task. If interrupts were permanently disabled by the user code, the bootloader would not be invoked any longer. In this case the watchdog resets the node. Even the situation in which an erroneous application manually resets the watchdog and disables the interrupt is prevented. For this, the bootloader modifies the firmware image of the user. The reset of the Watchdog is done with a specific 16 Bit opcode (see AVR Instruction Set). The bootloader exchanges all occurrences of this with the "no operation" opcode, thus the user cannot reset the watchdog even if they try.

The transmission of the beacons, used for ensuring connectivity, is done with the one-to-many communication protocol. A reset of a node must not depend on single beacons, since communication is assumed to be unreliable. However, a consecutively loss of beacons should lead to a recovery. Here, false alarms must be taken into account. Let p_r be the probability of a node to receive a beacon, R the beacon transmission rate, and T the maximum beacon waiting time. The false alarm probability becomes approximately $(1 - p_r)^{R \cdot T}$.

VI. Evaluation

This section evaluates the OTAP system in terms of scalability and reliability for an ATmega128RFA1 microcontroller. The proposed approach was implemented on top of an extensible, tiny operating system [10]. For the latter, a fault-tolerant tree-source routing protocol with end-to-end acknowledgments and a flooding algorithm are already available. Additionally, the forward error correction, as described in Sect. IV-C, was implemented. The required processing time of EEPROM operations and Reed Solomon decoding is summarized in Table I. The size of the firmware image was fixed to 64 KiB and is split in 256 segments, each containing 4 packets of size 64 Byte. The network manager sends beacons each minute. The recovery timeout was set to 10 minutes.

TABLE I: Processing Time for EEPROM and RS Operations

(a) Time to write/read data to/from the external EEPROM

Block Size (Byte)	Read (ms)	Write (ms)
64	4	7
128	13	12
256	24	24
512	48	49

(b) Decoding time dependent on number of needed redundant packets

Redundancy (Packets)	Decoding (ms)
0	0
1	4
2	6
3	8

Figure 6 presents the testbed deployment of about 100 wireless nodes in the heliostat field Jülich, Germany. This testbed pursues the goal to evaluate and validate the feasibility of a wireless mesh network as control technology in a field of self-powered heliostats, thus eliminating the need for cabling [1]. Since the heliostats are working autonomously, the wireless communication is mainly used for monitoring, calibration, and to trigger an emergency shutdown, e.g. in case of storm. The OTAP system is used for remote programming of experimental code. Since an on-the-spot support is expensive, the system must provide a high reliability.

A. Reliability

Up to now, approximately 20 new firmware versions were flashed in a period of nine month. During this time we

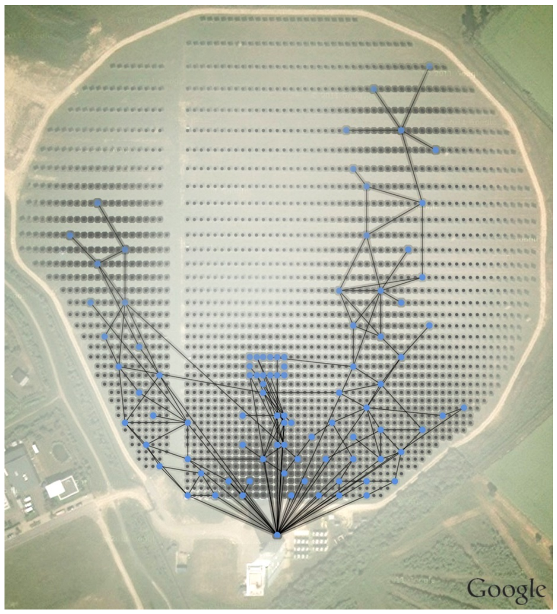


Fig. 4: Testbed Deployment in Jülich, Germany
(Latitude, Longitude: 50.913316, 6.387691)

observed node failures due to energy depletion, transient connectivity loss, and various software bugs. None of these problems lead to a permanent drop out of the OTAP system. However, non-recoverable failures were due to permanent hardware damage, e.g. caused by corrosion.

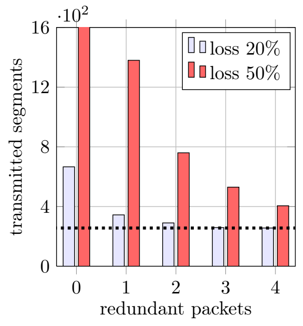
The recovery ability of the OTAP system was further exploited in an unusual way. When code for short-term experiments were deployed, the reset to the previous firmware was done via the recovery mechanism for reasons of convenience. For this, the network manager stopped broadcasting beacons and all nodes switched back to the fallback image.

B. Reed Solom Coding

The benefits of using RS codes were evaluated in a small test setup. In the experiment the network manager transmitted a firmware to a connected node. We use a wired connection which provides a nearly error free transmission. However, the network manager damaged packets with a given rate. The node will discard these packets, if the CRC-16 checksum is invalid. Redundant packets may increase the probability of a successful reception of a segment. In the experiment the total number of transmitted packets (inclusively retransmissions) was counted.

Communication costs in dependency of the packet error rates 20 % and 50 % are shown in Fig. 5. Transmitted segments (left plot) and transmitted packets (right plot) are depicted for different numbers of redundant packets. The dotted black line shows for both cases the optimal (error-free) value for segment and packet transmissions which is 256 and 1024, respectively.

The transmission with RS codes significantly reduces communication costs for lossy channels. For instance, nearly no retransmission were necessary for 3 redundant packets and an error rate of 20 %. However, the number of packet transmissions was 1813 as opposed to 1024 for the optimal case. A remarkable reduction of the communication costs were



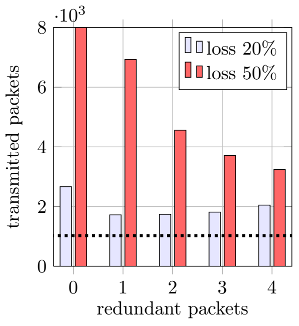
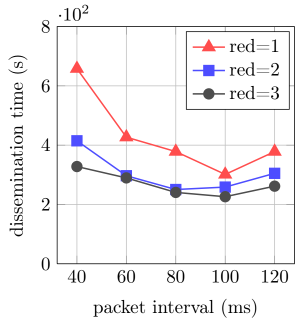


Fig. 5: Reed Solomon Coding



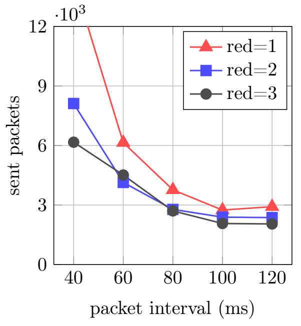


Fig. 6: Reprogramming of 50 nodes

achieved for a loss rate of 50 %. In such an error prone environment a successful reception of a segment without error correction is unlikely ($0.5^4 = 0.0625$). The use of 4 redundant packets reduced the number of segment transmissions to 405. The packet count is approximately 3 times higher than the optimal value of 1024.

An outcome of the experiment is that overrating is less crucial than underrating the number of redundant packets. Furthermore, the design of the network manager permits an adaption of the applied redundancy during runtime. For this, no modification of the implemented RS decoding is necessary.

C. *Performance*

For measuring the performance of the proposed OTAP system, 50 nodes were simultaneously programmed. For the experiment all other network services, e.g. monitoring, were disabled. Packet retransmission took place until all nodes had completely received the firmware image. As in the previous experiment we varied the number of redundant packets. Pure flooding was used for the one-to-many communication. This is reasonable since the density of the deployment was low (cf. Fig. 6) and thus the *Broadcast Storm Problem* is avoided [8]. In order to analyze the influence of overload and congestion, different packet rates for propagating the firmware are applied.

In Fig. 6 the dissemination time (left) and the total number of transmitted packets (right) are shown. For each single setup we averaged the results of 10 runs. In general, the transmission interval of firmware packets has a significant influence. Contrary to the intuition, a low value, which corresponds to a high packet rate, does not decrease the dissemination time. On the one hand, internal packet handling (cf. Table I) inhibits a node from forwarding data. On the other hand, high data rates lead to congestion. The optimal results were achieved for a packet interval of 100 ms and 3 packet in redundancy for each segment. For this setting we end up with a dissemination time of less than 250 s and a transmission count of 2069 packets.

VII. Conclusion

This paper introduced a comprehensive and fail-safe system for over-the-air programming. Its functionality is maintained even in case of critical software failures and transient hardware errors. For this purpose, we propose a recovery system which inherently stays available during the execution of arbitrary user code. For this, periodic beacons are used to ensure connectivity and a valid operational state. Furthermore, we significantly improve the performance of the wireless firmware dissemination by applying Reed Solomon codes.

The proposed OTAP system was evaluated in terms of scalability and reliability in a testbed deployment in a heliostat power plant. Even after extensive use, no failure of the OTAP system occurred. In the testbed, the transmission of a firmware to 50 nodes could be accomplished in less than 250 s using IEEE 802.15.4 compliant hardware.

Due to the modularity of the proposed OTAP, enhancements can be easily integrated, e.g. data compression or more efficient communication protocols. Next to this optimization, we plan to port the system to other wireless platforms.

References

[1] S. Kubisch, M. Randt, R. Buck, A. Pfahl, and S. Unterschütz, “Wireless Heliostat and Control System for Large Self-Powered Heliostat Fields,” in *SolarPACES’11: Proc. 17th International Symposium on Concentrated Solar Power and Chemical Energy Technologies*, Sep. 2011.

[2] J. W. Hui and D. Culler, “The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale,” in *SenSys ’04: Proc. 2nd international conference on Embedded networked sensor systems*, 2004, pp. 81–94.

[3] P. K. Dutta, J. W. Hui, D. C. Chu, and D. E. Culler, “Securing the Deluge Network Programming System,” in *IPSN ’05: Proc. 5th International Conference on Information Processing in Sensor Networks*, Apr. 2006, pp. 326–333.

[4] A. Dunkels, B. Gronvall, and T. Voigt, “Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors,” in *LCN ’04: Proc. 29th Annual IEEE International Conference on Local Computer Networks*, Nov. 2004, pp. 455–462.

[5] B. Porter, U. Roedig, and G. Coulson, “Type-safe Updating for Modular WSN Software,” in *DCOSS ’11: Proc. 7th International Conference and Workshops on Distributed Computing in Sensor Systems*, Jun. 2011, pp. 1–8.

[6] S. Brown and C. J. Sreenan, “Software Update Recovery for Wireless Sensor Networks,” in *SENSAPPEAL ’09: Proc. 1st International Conference on Sensor Networks Applications, Experimentation and Logistics*, Sep. 2009, pp. 107–125.

[7] S. Brown, “Updating Software in Wireless Sensor Networks: A Survey,” Tech. Rep., 2006.

[8] S.-Y. Ni, Y.-C. Tseng, Y.-S. Chen, and J.-P. Sheu, “The Broadcast Storm Problem in a Mobile Ad Hoc Network,” in *MobiCom ’99: Proc. 5th International Conference on Mobile Computing and Networking*, Aug. 1999, pp. 151–162.

[9] J. S. Plank, “A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems,” *Software – Practice & Experience*, vol. 27, pp. 995–1012, 1997.

[10] S. Unterschütz, A. Weigel, and V. Turau, “Cross-Platform Protocol Development Based on OMNeT++,” in *OMNeT++’12:Proc. 5th International Workshop on OMNeT++*, Mar. 2012.