Remote Reprogramming of a Wireless Sensor Network / Απομακρυσμένος επαναπρογραμματισμός ασύρματου δικτύου αισθητήρων

Diploma thesis for the Department of Computer and Communications Engineering, University of Thessaly

Διπλωματική εργασία για το Τμήμα Μηχανικών Η/Υ, Τηλεπικοινωνιών και Δικτύων, Πανεπιστήμιο Θεσσαλίας

Author:
Asvestopoulos Alexandros - Vasileios

Συγγραφέας:
Ασβεστόπουλος Αλέξανδρος - Βασίλειος

Thesis supervisor: Spyros Lalis, Associate Professor, CCED

Επιβλέπων καθηγητής: Σπύρος Λάλης, Αναπληρωτής Καθηγητής, ΤΜΗΥΤΔ

Thesis examination committee / Εξεταστική επιτροπή διπλωματικής εργασίας:

Spyros Lalis / Σπύρος Λάλης,
Associate Professor, CCED / Αναπληρωτής Καθηγητής, ΤΜΗΥΤΔ

Dimitrios Katsaros / Δημήτριος Κατσαρός,
Lecturer, CCED / Λέκτορας, ΤΜΗΥΤΔ

Acknowledgements / Ευχαριστίες

I deeply thank my professor and thesis supervisor Dr. Spyros Lalis for providing me with the opportunity to work on the present thesis subject, as well as for his undivided guidance and support throughout my work. I would also like to thank my professor and member of the thesis examination committee Dr. Dimitrios Katsaros, for approving my thesis.

I am grateful to the people of the R&D department of Prisma Electronics, for giving me the opportunity to implement my thesis on their platform and for their help, and to the Ph.D. candidate Mr. Emmanouil Koutsoumbelias, for his help and for sharing his experiences on embedded systems with me. Moreover, I thank the Ph.D. candidate Mr. Athanasios Fevgas for his help in the beginning of my thesis.

Many thanks to my family and friends, whose encouragement and support gave me the strength to accomplish my thesis, as well as my studies.

Ευχαριστώ θερμά τον καθηγητή μου και επιβλέποντα της διπλωματικής μου εργασίας Δρ. Σπύρο Λάλη για την ευκαιρία που μου έδωσε να ασχοληθώ με το παρόν θέμα, καθώς και για την αμέριστη καθοδήγηση και υποστήριξή του καθ' όλη την διάρκεια της ενασχόλησής μου. Επίσης, θα ήθελα να ευχαριστήσω τον καθηγητή μου και μέλος της εξεταστικής επιτροπής της διπλωματικής μου εργασίας Δρ. Δημήτριο Κατσαρό που ενέκρινε την πορεία της εργασίας μου.

Είμαι ευγνώμων στους ανθρώπους του τμήματος E&A της Πρίσμα Ηλεκτρονικά, για την ευκαιρία που μου έδωσαν να υλοποιήσω την εργασία μου στην πλατφόρμα τους και για την βοήθειά τους, και στον υποψήφιο διδάκτορα του ΤΜΗΥΤΔ κ. Εμμανουήλ Κουτσουμπέλια, για την βοήθειά του και που μοιράστηκε τις εμπειρίες του πάνω στα ενσωματωμένα συστήματα μαζί μου. Επιπλέον, ευχαριστώ τον υποψήφιο διδάκτορα του ΤΜΗΥΤΔ κ. Αθανάσιο Φεύγα για την βοήθειά του στο ξεκίνημα της διπλωματικής μου.

Τέλος, θέλω να δώσω πολλά ευχαριστώ στην οικογένειά μου και τους φίλους μου, η συμπαράσταση και υποστήριξη των οποίων μου έδωσε την δύναμη να φέρω εις πέρας διπλωματική μου εργασία, αλλά και τις σπουδές μου.

Abstract

As the ever growing field of wireless sensor networks has a wide application range, the ability to dynamically deploy applications to such networks, without physically accessing each node (reprogramming ability of a WSN), proves to be a matter of great importance. This ability eases both the maintenance and re-tasking of the system, which in some cases are otherwise difficult or even harmful to the purpose of the deployment. Nevertheless, supporting the reprogramming of a WSN is rather challenging as there are quite a few aspects to consider, emerging from the resource-constrained sensor nodes and the unreliable nature of wireless communications.

This thesis presents the design and implementation of a utility enabling the remote reprogramming of a WSN (known as “reprogramming of a WSN” in literature), focusing on the constraints of the implementation platform (PrismaSense WSN [1]). Nevertheless, the designed reprogramming procedure is generally applicable on any WSN platform.

The solution was progressively developed in three versions, each one refining and improving the functionality offered. Version 1 updates the entire code image of the node via a stop-and-wait and point-to-point procedure, in a non-intrusive manner to the currently running code image and with the ability to resume a previous update. This way, the entire WSN is reprogrammed node-by-node. In version 2, the key improvement is that only the application-specific part of the image is updated, while the real-time operating system (RTOS) part is left intact. Of course, any inconsistency between the RTOS image part that the update was compiled for and the RTOS image part present on the node is detected by the procedure, to avoid unfortunate results. The last version enables the one-to-many updates, utilizing the native radio broadcasts, in order to update all the nodes at once. For this reason, the stop-and-wait scheme used in propagating the update is replaced by a non-ARQ, NACK based one. This modification speeds up the point-to-point update procedure as well.

As regards the evaluation of the three solution versions, a series of tests validated the implemented functionality and asserted the improvement obtained by each version. The tests results also indicate the network sizes for which the one-to-many update procedure outperforms the point-to-point one in reprogramming the entire WSN.

Περίληψη

Το διαρκώς εξελισσόμενο πεδίο των ασύρματων δικτύων αισθητήρων (WSNs) βρίσκει εφαρμογή σε πολλούς τομείς, όπως στον στρατό, την βιομηχανία, την υγεία, την παρακολούθηση περιβάλλοντος ή τα έξυπνα σπίτια. Ως εκ τούτου, η δυνατότητα απομακρυσμένου επαναπρογραμματισμού των δικτύων αυτών, δηλαδή η δυνατότητα δυναμικής εγκατάστασης μιας εφαρμογής στο δίκτυο χωρίς να χρειάζεται φυσική πρόσβαση σε κάθε κόμβο, θεωρείται σημαντικό ζήτημα. Μια τέτοια δυνατότητα διευκολύνει σε μεγάλο βαθμό εργασίες τόσο για τη συντήρηση, όσο και για τον επαναπροσδιορισμό των στόχων ενός WSN (πχ από μέτρηση θερμοκρασίας σε μέτρηση κραδασμών). Αυτές οι εργασίες, χωρίς την δυνατότητα επαναπρογραμματισμού, είναι εξαιρετικά δύσκολες ή ακόμη και ανέφικτο να γίνουν χωρίς να βλάψουν τον σκοπό της εγκατάστασης (πχ παρακολούθηση συνθηκών σε φωλιά πουλιών).

Ωστόσο, η υποστήριξη της δυνατότητας επαναπρογραμματισμού ενός WSN αντιμετωπίζει αρκετά ζητήματα, που πηγάζουν από τους ούτως ή άλλως περιορισμένους σε πόρους κόμβους και την αναξιόπιστη φύση της ασύρματης επικοινωνίας. Τέτοια ζητήματα είναι τα κόστη σε επικοινωνία, υπολογισμούς, ισχύ και μνήμη, όπως επίσης και η αξιοπιστία, ευρωστία και ασφάλεια της διαδικασίας επαναπρογραμματισμού.

Η παρούσα διπλωματική εργασία παρουσιάζει τον σχεδιασμό και την υλοποίηση του μηχανισμού απομακρυσμένου επαναπρογραμματισμού ασύρματου δικτύου αισθητήρων (γνωστός ως επαναπρογραμματισμός ασύρματου δικτύου αισθητήρων στη βιβλιογραφία), επικεντρώνοντας στους περιορισμούς που εισάγει η πλατφόρμα υλοποίησης (PrismaSense [1] WSN). Πριν, αλλά και κατά τη διάρκεια ανάπτυξης της λύσης, μελετήθηκε λεπτομερώς η πλατφόρμα υλοποίησης, και έγιναν οι απαραίτητες αλλαγές σε επίπεδο λογισμικού, ώστε να ενσωματωθεί σ' αυτή ο υλοποιημένος μηχανισμός. Εντούτοις, ο βασικός σχεδιασμός της λύσης μπορεί να εφαρμοστεί σε οποιαδήποτε πλατφόρμα.

Η λύση αναπτύχθηκε σταδιακά σε τρεις εκδόσεις, με κάθε μία να βελτιώνει και να επαυξάνει την παρεχόμενη λειτουργικότητα. Και στις τρεις εκδόσεις, η διαδικασία επαναπρογραμματισμού/ενημέρωσης κώδικα ενός κόμβου μπορεί να χωριστεί σε δύο κύρια μέρη. Στο πρώτο μέρος, ο κόμβος λαμβάνει το νέο εκτελέσιμο κώδικα σε κομμάτια (chunks) από τον σταθμό βάσης (H/Y) και τα αποθηκεύει προσωρινά σε αποθήκη, στην μη-πτητική μνήμη του. Στο δεύτερο μέρος, ο κόμβος αντικαθιστά τον τρέχοντα κώδικα με το νέο, αντιγράφοντας αδιάλειπτα τον δεύτερο από τη προσωρινή αποθήκη στη θέση του πρώτου στην μνήμη προγράμματος (μη-πτητική μνήμη), και επανεκκινεί για να εκτελέσει τον νέο κώδικα. Επίσης, κάθε υλοποιημένη έκδοση αποτελείται από δύο κύρια κομμάτια λογισμικού, ένα για τον σταθμό βάσης (H/Y) κι ένα σε κάθε ασύρματο κόμβο αισθητήρων. Το μεν πρώτο αναπτύχθηκε στη γλώσσα C# και είναι μια εφαρμογή κονσόλας, το δε δεύτερο στην έκδοση της C που υλοποιεί ο μικροελεγκτής των κόμβων.

Η πρώτη έκδοση της λύσης ενημερώνει ολόκληρο τον εκτελέσιμο κώδικα ενός κόμβου, μέσω μιας παύσης-και-αναμονής και σημείο-προς-σημείο διαδικασίας, όπου η λήψη κάθε κομματιού κώδικα επιβεβαιώνεται ρητά προς τον σταθμό βάσης πριν σταλεί το επόμενο. Έτσι, ολόκληρο το δίκτυο αισθητήρων επαναπρογραμματίζεται κόμβο προς κόμβο. Αξίζει να αναφερθεί ότι η τρέχουσα εφαρμογή δεν διακόπτεται ώσπου να τελειώσει η λήψη του νέου κώδικα, κι ότι είναι δυνατό να συνεχιστεί μια παλιότερη λήψη από το σημείο που είχε σταματήσει. Ωστόσο, ο τρέχον εκτελέσιμος κώδικας κι ο νέος πρέπει να χωράνε και οι δύο στην διαθέσιμη μη-πτητική μνήμη του κόμβου, πράγμα το οποίο περιορίζει σημαντικά την λειτουργικότητα της κάθε εφαρμογής.

Στην δεύτερη έκδοση η σημαντικότερη βελτίωση είναι ότι πλέον ενημερώνεται μόνο ο κώδικας εφαρμογής, αφήνοντας τον κώδικα του λειτουργικού συστήματος πραγματικού χρόνου (RTOS) άθικτο. Βεβαίως, τυχόν ασυμβατότητες ανάμεσα στον κώδικα RTOS για τον οποίο αναπτύχθηκε η νέα εφαρμογή και τον κώδικα RTOS που υπάρχει στον κόμβο, οδηγεί σε ματαίωση της διαδικασίας ώστε να αποφευχθούν ανεπιθύμητα αποτελέσματα. Επίσης, η διαδικασία αντικατάστασης του τρέχοντος κώδικα με το νέο προσαρμόστηκε ώστε να μπορεί να συνεχίσει αν διακοπεί από σφάλμα επανεκκίνησης, αυξάνοντας την αξιοπιστία της λύσης. Η υπόλοιπη λειτουργικότητα διατηρήθηκε από την πρώτη έκδοση.

Η τρίτη και τελευταία έκδοση καθιστά εφικτή την ένας-προς-πολλούς διαδικασία ενημέρωσης, ώστε να επαναπρογραμματίζονται όλοι οι κόμβοι του δικτύου μονομιάς. Για τον λόγο αυτό, το παύσης-και-αναμονής σχήμα της διαδικασίας προώθησης του νέου κώδικα αντικαταστάθηκε από ένα σχήμα μη αυτόματης επανάληψης αίτησης (non-ARQ) βασισμένο σε αρνητικές επιβεβαιώσεις λήψης (NACK). Έτσι, ο νέος κώδικας προωθείται σε ομάδες κομματιών κώδικα (chunks), σε διαδοχικές φάσεις διάδοσης-επιδιόρθωσης για κάθε ομάδα. Στη φάση διάδοσης τα κομμάτια μιας ομάδας εκπέμπονται χωρίς να επιβεβαιώνονται ξεχωριστά. Κατόπιν, στη φάση επιδιόρθωσης συλλέγονται οι όποιες συγκεντρωτικές αρνητικές επιβεβαιώσεις λήψης και όσα κομμάτια χάθηκαν επανεκπέμπονται. Επίσης, στην ένας-προς-πολλούς διαδικασία ενημέρωσης, οι εκπομπές από το σταθμό βάσης προς τους κόμβους είναι πανεκπομπές και οι κόμβοι

απαντάνε με μονοεκπομπές, ενώ στην σημείο-προς-σημείο διαδικασία οι εκπομπές και των δύο μεριών είναι μονοεκπομπές (όπως και στις προηγούμενες εκδόσεις). Η υπόλοιπη λειτουργικότητα διατηρήθηκε από την δεύτερη έκδοση. Στα αξιοσημείωτα, το νέο αυτό σχήμα της διαδικασίας προώθησης του νέου κώδικα, εκτός του ότι επιτρέπει στην ένας-προς-πολλούς διαδικασία ενημέρωσης να κλιμακώνει ικανοποιητικά, επιταχύνει σημαντικά και την σημείο-προς-σημείο διαδικασία ενημέρωσης.

Σχετικά με την αξιολόγηση των τριών εκδόσεων της λύσης, η υλοποιημένη λειτουργικότητα κάθε έκδοσης επικυρώθηκε μέσω δοκιμών. Επίσης, τα αποτελέσματα των μετρήσεων στην καθυστέρηση της διαδικασίας επαναπρογραμματισμού για διάφορες τοπολογίες του δικτύου αισθητήρων, επιβεβαίωσαν την σταδιακή βελτίωση από έκδοση σε έκδοση. Τέλος, οι μετρήσεις έδειξαν το μέγεθος που χρειάζεται να έχει το δίκτυο (πλήθος κόμβων), ώστε η ένας-προς-πολλούς διαδικασία ενημέρωσης κι ο επαναπρογραμματισμός όλων των κόμβων μονομιάς να υπερτερούν της σημείο-προς-σημείο διαδικασίας ενημέρωσης και του κόμβο-προς-κόμβο επαναπρογραμματισμού του δικτύου.

Αξίζει επίσης να σημειωθεί ότι κατά τη διάρκεια ανάπτυξης της λύσης μελετήθηκαν σχετικές δουλειές πάνω στον επαναπρογραμματισμό των ασύρματων δικτύων αισθητήρων, από διάφορες δημοσιεύσεις, για την καλύτερη κατανόηση του προβλήματος. Οι μηχανισμοί που προτείνονται στις δημοσιεύσεις αυτές αναλύθηκαν και βοήθησαν σε συγκεκριμένες σχεδιαστικές αποφάσεις της τρίτης έκδοσης της λύσης, σχετικά με την διάδοση του νέου εκτελέσιμου κώδικα σε ομάδες τμημάτων κώδικα.

Contents

1. Introduction

Reprogramming of a wireless sensor network (also known as reconfiguration or re-tasking in literature) is the ability to dynamically deploy applications to the nodes of the WSN that have been placed in field, without physically accessing each node. As WSNs are used in a wide application range (e.g. military, industry, environment monitoring, healthcare, smart homes), their reprogramming proves to be a matter of great importance.

The ability to reprogram a WSN eases the maintenance and re-tasking of the system, which in some cases are otherwise difficult or even harmful to the purpose of the deployed WSN. End users need an easy way to re-task the deployed network, without the intervention of a technician. Application developers and system administrators need to update the WSN software in order to fix bugs, introduce new functionality or change the operation parameters. In several cases, physical access of wireless nodes may be difficult or impossible (e.g. a node on a mountain peak), or damaging the deployment's task (e.g. a node in a bird nest for environmental monitoring).Thus, the reprogramming ability increases the flexibility and reliability of a deployment.

Any approach enabling the reprogramming of a WSN needs to consider quite a few aspects. First of all, as the sensor systems are fairly constrained in resources, the developed solution needs to meet performance requirements such as communication and computational overhead, energy and memory storage cost and latency. Next, several reliability, scalability and security issues emerge from the unreliable and open nature of wireless communications. Specifically, the entire new program code (or a part of it) needs to be received by all targeted nodes correctly, despite any temporary communication problems. Also, the reprogramming procedure needs to be robust enough to deal with large scale networks, as well as adaptive to dynamic changes in the size of the network. Regarding security, the procedure could make the WSN vulnerable to a number of attacks (e.g. reprogramming could be used as a workaround to the systems security measures). Finally, the approach should be non-intrusive to the WSNs operation and require as less human intervention as possible.

This thesis presents the design and implementation of a utility enabling the remote reprogramming of a WSN[1], focusing on the constraints of the implementation platform (PrismaSense [1] WSN). The solution was progressively developed in three versions, each one refining and improving the functionality offered. Since the main objective was for the solution to work under the target platform's software and hardware constraints, several aspects were left as future work (e.g. security). Nevertheless, the designed reprogramming procedure is generally applicable on any WSN platform.

The rest of this thesis is structured as follows. In chapter 2, related work in the field of reprogramming and code dissemination in WSNs is discussed in short. Chapter 3 offers insight into the implementation platform. Chapter 4 describes the general design of solution, while chapters 5, 6 and 7 focus on each of the three versions implemented. Chapter 8 presents the tests made and their results, regarding the evaluation of the three versions. Finally, chapter 9 concludes the work done and discusses possible feature work. The three versions of the custom protocol designed for the reprogramming procedure, and implemented by each of the three solution versions, as well as the validation tests and their results can be found in the appendix.

[1] Actually the concepts "reprogramming" and "remote reprogramming" of a WSN are identical. The adjective "remote" is consciously added to emphasize the absence of physical access to the nodes. Therefore, it is used both in the subject and when referring to the solution designed and implemented in the present thesis.

2. Related work

Multi-hop Over-the-Air Programming (MOAP) [2] is a code distribution mechanism targeted and implemented for Mica-2 motes, trading energy and memory efficiency for increased latency. It uses a store-and-forward approach and a publish-subscribe scheme to distribute the code ("Ripple" dissemination protocol). The base station initiates the procedure by broadcasting publish messages, with each node that received the code further disseminating it. Once a node receives no subscribe messages and is finished transmitting, it transfers the new image from EEPROM to program memory and reboots to it. A sliding window for the retransmission tracking scheme and unicast retransmissions are adopted, reducing power consumption (EEPROM access) at the cost of reduced out-of-order message tolerance. Furthermore, a "late joiner" mechanism addresses temporary node and network failures via each node periodically advertising its version.

Although MOAP uses a link-statistics mechanism to avoid unreliable links, there is no other support for rate control or suppressing multiple senders. Moreover, the latency is increased due to receiver-source role decoupling, the absence of pipelining and the fact that MOAP code is transferred over the air too. Nevertheless, emulation and testbed experiments showed 60-90% less communication overhead than flooding.

Borrowing techniques from the epidemic/gossip, scalable multicast, and wireless broadcast literature, Trickle [3] uses a "polite gossip" policy, where motes periodically broadcast a code summary to local neighbors, but stay quiet if they have recently heard a summary identical to theirs from a certain number of neighbors. When a mote hears an older summary than its own, it brings everyone nearby up to date by broadcasting an update. Trickle dynamically regulates the per-node, Trickle-related traffic to a particular rate, thus adjusting automatically to the local network density. This scales well, even with packet loss taken into account. Moreover, dynamic changes of the gossip interval enable rapid propagation of the new code, while using less network bandwidth when there are no known changes. However, Trickle assumes that motes (and their radio transceivers) are always on. Finally, a policy for the actual propagation, once update need detected, is largely ignored in the original paper. Notably, Trickle has been implemented in Mate [4].

Deluge [5] is built on prior work on density-aware, epidemic maintenance protocols (Trickle) with several optimizations and enhancements. Specifically, a three-way handshaking protocol ensures that a bi-directional link exists before transferring data (ADV, REQ, DATA), addressing the asymmetric link problem. Additionally, the new image is represented as a set of fixed-size pages (further split to fixed-size packets), providing a manageable unit of transfer and allowing for spatial multiplexing. Deluge advertises the availability of received pages even before all pages of the image are received by a node, enabling pipelining. It also supports incremental updates, through the use of advertisements which indicate which pages in an image have changed since the previous version.

Deluge's main disadvantage is that it disseminates the image of the protocol together with the program to be transferred, causing up to twenty times more overhead for the transmission of a program consisting of a single page. Its implementation on Mica2-dot nodes can push nearly 90 bytes per second, reaching 11% of the maximum transmission rate of the radio supported under TinyOS. Control messages are limited to 18% of all messages, while a node receives about 3.35 times the minimum number of required data packets, due to the single-channel, broadcast network.

In [6], Infuse is presented. It is a TDMA-based protocol that disseminates bulk data in location-aware sensor networks. A selective listening policy on TDMA slots by the receiver reduces energy consumption (no congestion in contrast to CSMA), while the nodes periodically select predecessors and successors to prevent broadcast storms.

First, the base station broadcasts a start-download message containing a version ID and specifying the number of messages/capsules in the new data sequence, and then sends the capsules in subsequent TDMA slots. The receivers forward the start-download message and capsules, also saving the latter in flash. When the last capsule is received, the application is signaled that the download is complete. Two loss recovery mechanisms are discussed: Go-Back-N, based on implicit acknowledgements, and Selective-Retransmission, based on explicit, piggybacked acknowledgements. The algorithm parameters regarding the loss recovery mechanism and the use of selective predecessors (or not) are defined at compile time, resulting in four possible versions of Infuse. The authors claim that Infuse achieves a more effective pipelining and smaller dissemination latency than Deluge, by avoiding the CSMA contention at the middle of the network.

Stream [7][8] builds on Deluge and optimizes what is actually sent over the channel (main disadvantage of Deluge), by using the facility of having multiple code images on a node and switching between them. This is done by partitioning the flash memory and pre-installing the reprogramming protocol image in a different segment than the application program image. The application program is equipped with the ability to listen for updates and switching to the reprogramming image. Additionally, the authors designed an opportunistic sleeping scheme whereby nodes can sleep during the period when reprogramming has been initiated, but has not yet reached the neighborhood of the node. The savings become significant for large networks and for frequent reprogramming.

Simulation and testbed experiments on Mica2 motes demonstrate up to 98% reduction in reprogramming time and up to 132% reduction in the number of bytes transferred compared to Deluge. DStream [9] is a newer version with single and multi-hop reprogramming capabilities to choose from, depending on factors like network size, node density and most importantly, link reliability.

MNP, a multi-hop network reprogramming protocol, was proposed in [10][11], and provides a reliable service to propagate new program code to all sensor nodes in the network over wireless radio. To reduce the problems of collision and hidden terminal, MNP incorporates a sender selection algorithm that attempts to allow only one active sender in a particular neighborhood at a time, also greedily selecting the sender which sender will have the most impact (most number of requests). MNP also uses pipelining to enable fast data propagation, while it reduces the active radio time and energy consumption of a node by putting the node into a “sleep” state when its neighbors are transmitting a segment that is not of interest (contention sleep). To further reduce the energy consumption, another “noreq” sleep was added, where a node goes to sleep if none of its neighbors is interested in receiving the segment it is advertising. An optional init sleep was also introduced in the initial phase of reprogramming. In particular, the MNP service can be tuned so that the probability that a node is assigned to transmit the code is proportional to its remaining battery life. MNP operates in 4 phases (figure 1), using three-way handshaking as Deluge.

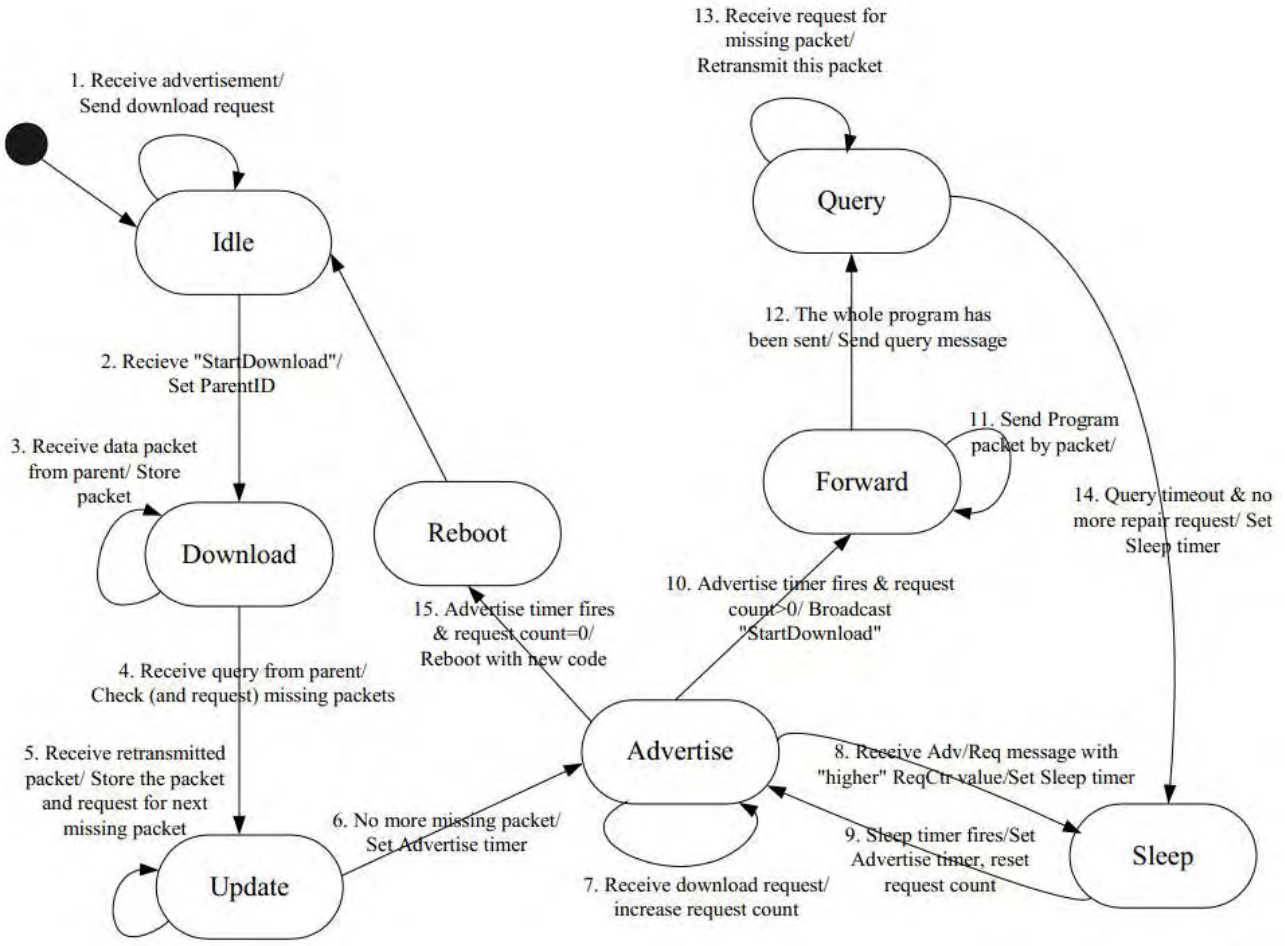


figure 1 MNP state diagram

Aqueduct [12] expands Deluge to support reprogramming of heterogeneous WSNs and multiple applications. Specifically, a FORWARD node state is added to those defined in Deluge to establish “aqueducts” of intermediate nodes between source nodes and target nodes. These aqueducts are built only upon shortest path with symmetric links to avoid paths that may be shorter in terms of hops, but include unreliable and asymmetric links. Aqueduct nodes use a FIFO cache to temporarily store more than one pages of the new image.

Unlike other approaches, SYNAPSE [13] focuses on extremely efficient solutions for the local delivery of the data (efficient error recovery phase), as well as their proper integration with previous techniques. It implements a Hybrid ARQ solution, where data are encoded prior to transmission via Fountain Codes and uses incremental redundancy to recover from losses, thus considerably reducing the transmission overhead. Nevertheless, it adopts many of the techniques in Deluge, MNP and Stream.

SYNAPSE uses three-way handshaking (ADV-REQ-CODE) with randomization when sending advertisements. The code dissemination is performed hop-by-hop, where data blocks are sent during dissemination rounds, where only one node at a time is allowed to send. Also, broadcast transmissions are used for the code and the nodes request missing data via NACKs. Finally, the reprogramming module is preinstalled on the nodes. Experimental results show that SYNAPSE outperforms Deluge in data traffic, dissemination time, and control traffic, with the difference augmenting with packet loss probability.

In [14], the authors proposed a protocol called Freshet, optimizing the energy for code upload and speeding up the dissemination if multiple sources of code are available. The protocol, to handle multiple sources, provides a loose coupling of nodes to a source and disseminates code in waves each originating at a source with a mechanism to handle collisions when the waves meet. The suppression of redundant transmissions of the data and the metadata uses the shared nature of the wireless medium and the capacity of a node to overhear its neighbors’ communication. Pipelining of the different pages in a binary image is used to expedite the code upload. Freshet can also speed up the process when multiple sources of code are available, by allowing nodes to receive pages out of sequence for streams from different sources. The energy optimization is achieved by equipping each node with limited nonlocal topology information which it uses to determine the time when it can go to sleep since code is not being distributed in its vicinity. That is, to reduce the energy consumption due to code upload, nodes can be put to sleep by making the advertisement-request-data handshake happen only at certain points in time.

Freshet functions in three phases for each new code image: blitzkrieg, distribution, and quiescent. When new code is introduced into the network, Freshet has an initial phase, the blitzkrieg phase, when information about the code propagates through the network rapidly along with some topology information. The topology information is used by each node to estimate when the code will arrive in its vicinity and the three-way handshake (distribution phase) will be initiated. Each node can go to sleep in between the blitzkrieg phase, and the distribution phase, thereby saving energy. Freshet also optimizes the energy consumption by exponentially reducing the metadata rate during conditions of stability in the network (the quiescent phase) when no new code is being introduced. Simulations proved Freshet to be 20–45% more energy efficient than Deluge, while requiring about 10% more time for code propagation

The authors of [15] present Melete, a system that supports concurrent applications with efficiency, reliability, flexibility, programmability, and scalability. It is based on the Mate virtual machine and Trickle with significant modifications and enhancements. Melete enables reliable storage and execution of concurrent applications on a single sensor node. Dynamic grouping is used for flexible, on-the-fly deployment of applications based on contemporary status of the nodes. Melete uses a group-keyed code dissemination mechanism, developed for reliable and efficient code distribution among sensor nodes.

In more detail, Melete uses a passive code dissemination policy (dissemination initiated by the requester) with active advertisements (all nodes advertise all applications), to minimize network traffic overhead while keeping all sensor nodes up to date without large delay. One more state is added to those of Trickle, whereas the distribution is done in a selective and reactive fashion, instead of the proactive and unbiased one of Trickle. To prevent neighbors of a requester from blindly expanding the forwarding region (intermediate nodes between requester and source) to the entire network even though the request can be resolved by other neighbors locally, lazy forwarding and progressive flooding are adopted. Simulation results indicate satisfactory scalability of the system to both application code size and node density.

MCP [16] is a stateful Multicast based Code redistribution Protocol for achieving energy efficiency, supporting the case of different applications in the same WSN. It is similar to Melete, but uses multicast instead of broadcast to reduce collisions and communication overhead. A gossip-based source node discovery is incorporated with each node maintaining a small table to record the information on known applications (source, next hop to source, # of pages, version). The table enables the forwarding of multicast-

based code dissemination requests, in which only a subset of neighboring node contributes to code dissemination. Furthermore, this table is periodically sent out by each node.

Initially, the sink floods a dissemination command with the subset of nodes to be updated. Afterwards, each node chooses between three roles (source, requester or forwarder) and the multicast dissemination scheme takes place. On average, MCP requires 45% and 25% less time to finish over Deluge and Melete respectively, and about 20% less message overhead than Melete.

In [17] a dependable data dissemination protocol is introduced for time-efficient and secure code updates in large-scale wireless sensor networks. A multi-hop propagation scheme is used, based on security-enhanced fountain codes and means from fuzzy control theory. To address the packet collisions and hidden terminal problems, means from fuzzy control were used to dynamically adapt the send rates. In short, a fuzzy controller takes as input the local radio channel congestion and the demand of neighbor-nodes for missing packets, and its output is used to define a time interval during which the next packet will be sent randomly. The local radio channel congestion level is deduced by overheard encoded packets, while the demand of neighbor-nodes for missing data packets is characterized by NACK packets. This protocol performs considerably faster than Deluge, at the expense of a larger, but less substantial, data overhead.

Another reprogramming protocol is ReXOR [18], which is based on Deluge. The emphasis in this approach is placed on the lightweight implementation and on the adaptivity to the network density (adaptive inter-page waiting time). As in Deluge, ReXOR employs three phases for Advertisement, Request and Data transmission. However, it uses XOR encoding for the retransmissions to reduce the communication cost (number of packets). In a nutshell, after the sender has collected the requests for retransmissions, it runs a coloring algorithm on the requested packets, encodes each color-group of packets to an XOR-ed packet and starts transmitting the XOR-ed packets (figure 2). The receivers decode the packets, if possible, by XOR-ing them with the ones they already have. If necessary, the receivers request any missing packets again. Once a node receives an entire page, it can serve as sender (pipelining) or request a new page. Notably, a page with a lower page number takes higher priority (sequential propagation). The evaluation results show that ReXOR achieves good network-level performance in both dense and sparse networks, compared with Deluge and a typical coding-based reprogramming protocol, Rateless Deluge (Deluge enhanced with Fountain codes).

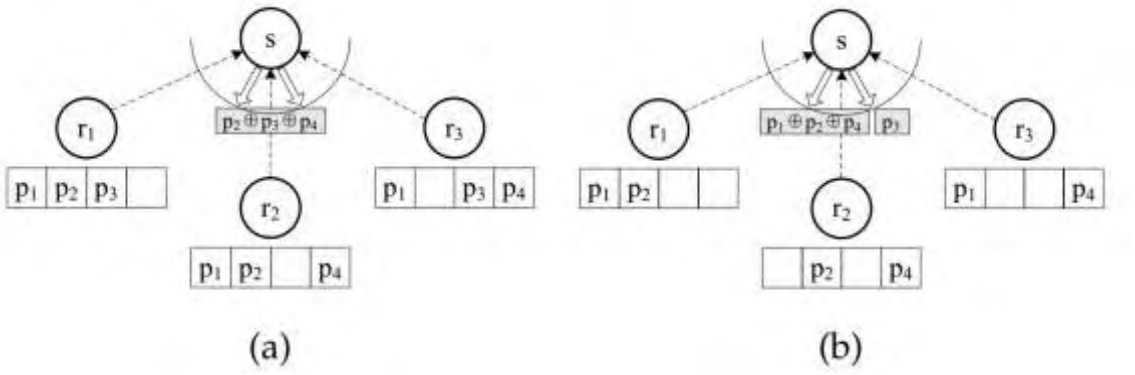


figure 2 ReXOR retransmission scheme

In [19], a lightweight and reliable Program Image Dissemination Protocol (PIDP) for autonomous ad hoc multi-hop WSNs is introduced. PIDP requires no external memory storage, is independent of the WSN stack, offers low overhead for transferring program images, and is designed to be platform-independent. It can reprogram the entire main image (including the WSN stack) with resistance to node failure and incompatibilities of main images. By design, it trades applicability on extremely resource-constrained platforms for intrusiveness and latency. Also, the propagation latency may further increase, due to the node-by-node reprogramming scheme.

PIDP consists of three parts. The first is a software version handshake, which happens between neighbors on power up, when a node exchanges routing information or synchronizes with its neighbors. This way, information is exchanged hop-by-hop, without flooding and relatively seldom, which may limit the propagation of the image. The second is periodic advertisement, acting mainly as a failsafe, in case of a problem during reprogramming, image transfer disruption, or image incompatibilities (e.g. in the network stacks of different nodes). The last is the actual image transfer protocol, whereby the sender acts as a server and selects the image transfer radio channel (channel info in advertisement) and the receiver acts as a

client, requesting the image header and contents in blocks. Each block is immediately written to the program memory, invalidating the old image. A validation procedure follows, which on reboot triggers scanning for advertisements and re-transfer if the validation failed.

3. Implementation platform

3.1. Overview

The platform consists of a Real-time Operating System for embedded devices (Intelligent Sensors Operating System / ISOS), several PrismaSense Quax nodes, a PrismaSense Quax Gateway node, a PC communicating with the Quax nodes via the Quax Gateway node and a communication software layer residing on the PC.

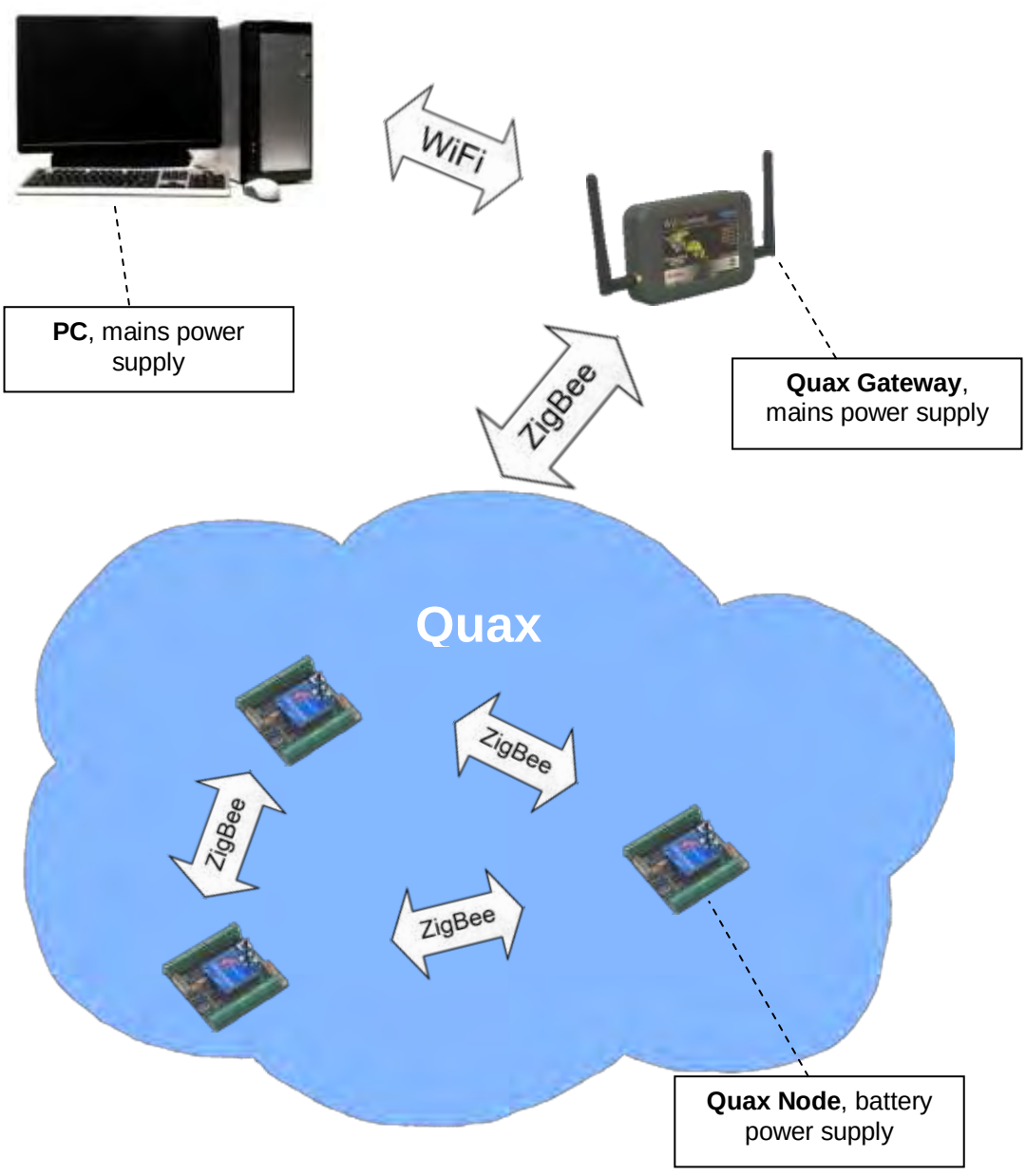


figure 1 Platform overview

3.2. Intelligent Sensors Operating System (ISOS)

ISOS is an operating system for intelligent sensors written in C. It was developed by Prisma Electronics S.A. to support the QUAX Motes and emphasized on controlling the consumed energy, increasing the reliability of the wireless network and easily developing new applications based on wireless smart sensors networks.

Some characteristics of ISOS are:

- Task Scheduler
- Event handler (software and hardware events/interrupts)
- Frequency Agility
- Packaging Buffer
- Measurement Simulation
- Safe Power Down Modes

3.3. PrismaSense Quax node

There are two types of Quax nodes, MS and DT. Both types have an MSP430F1611 microcontroller unit, several sensors, and a ZigBee module (using the XBee-Pro ZNet 2.5 firmware), and run the Intelligent Sensors Operating System (ISOS). The microcontroller unit has a 48 Kbyte program memory, 10 Kbyte RAM, 256 bytes of info memory and the ability of in-system programming its flash memory (figure 2). The Quax MS nodes also have an external memory of 512 Kbytes.

		MSP430F1611
Memory Main: interrupt vector Main: code memory	Size Flash Flash	48KB 0FFFFh – 0FFE0h 0FFFFh – 04000h
RAM (Total) Extended Mirrored	Size Size Size	10KB 038FFh – 01100h 8KB 038FFh – 01900h 2KB 018FFh – 01100h
Information memory	Size Flash	256 Byte 010FFh – 01000h
Boot memory	Size ROM	1KB 0FFFh – 0C00h
RAM (mirrored at 018FFh - 01100h)	Size	2KB 09FFh – 0200h
Peripherals	16-bit 8-bit 8-bit SFR	01FFh – 0100h 0FFh – 010h 0Fh – 00h

figure 2 MSP430F1611 memory organization

3.4. PrismaSense Quax Gateway node

This node has an MSP430F1611 microcontroller unit and a ZigBee module (as the PrismaSense Quax DT node), but instead of sensors, it has a Wi-Fi module. It runs a modified version of ISOS, as its only purpose is to act as a “dummy” communication bridge between the PC and the PrismaSense Quax nodes. Specifically, it is always the coordinator of the ZigBee network formed by itself and the Quax nodes, and forwards messages received from the Quax nodes (via ZigBee) to the PC (via Wi-Fi) and vice versa.

3.5. Communication software layer on the PC

This software layer is responsible for bilateral tunneling of data between the ad-hoc Wi-Fi network connecting the PC with the Quax Gateway and a virtual serial port. This way, any application on the PC handles the communication with the Quax Gateway as serial port communication.

3.6. ZigBee communications (XBee-Pro ZNet 2.5)

A brief description of the Quax node's communication capabilities is necessary in order to specify the design options available.

XBee module interface with microcontroller: The XBee module is configured by ISOS to operate in a specific mode (AP 2), which requires several special byte values to be escaped (flagged) when exchanged over the serial interface connecting the module with the MCU. This means that each of these byte values requires the transmission of two bytes over the serial interface, the special flag and the byte value XOR'ed with 0x20.

MAC: The XBee-Pro ZNet 2.5 implementation of ZigBee follows a CSMA/CA protocol for media access control.

Parent-child packet caching: Each ZigBee parent (ZigBee router/coordinator) holds a cache with FIFO eviction policy to store the last received packet(s) destined for any of its children (ZigBee end-devices), as the latter might be in a sleep mode. When a child wakes up and sends its beacon to the parent, any packets cached for the child are handed over to it. A parent's unicast packet cache can hold up to two packets or up to 120 bytes (whichever is less), while its broadcast packet cache can hold only one packet.

Broadcast packets: Anytime a router receives a broadcast packet it does two things to determine if it must retransmit that packet. It checks the source address (sender) and the sequence number of the broadcast packet, both of which are contained in the message's header. If these values don't match with any entry of an internal broadcast transaction table (of eight slots) that the router keeps, it retransmits the packet three times proactively.

The minimum coverage of broadcast packets is two hops. That is, all nodes within the range of the initial sender, as well as within the range of the routers which are one-hop neighbors of the initial sender will receive the packet, since these one-hop neighbor routers will rebroadcast the packet. For example, consider the network topology below (figure 3) with three routers, where each router is in the range of exactly one of the other two. If R1 initiates a broadcast with the minimum coverage selected, R3, as well as all end-device children of R1 and R2 will receive the broadcasted packet. Consequently, an actual one-hop broadcast reaching only the 1-hop neighbors of the broadcast initiator is impossible.

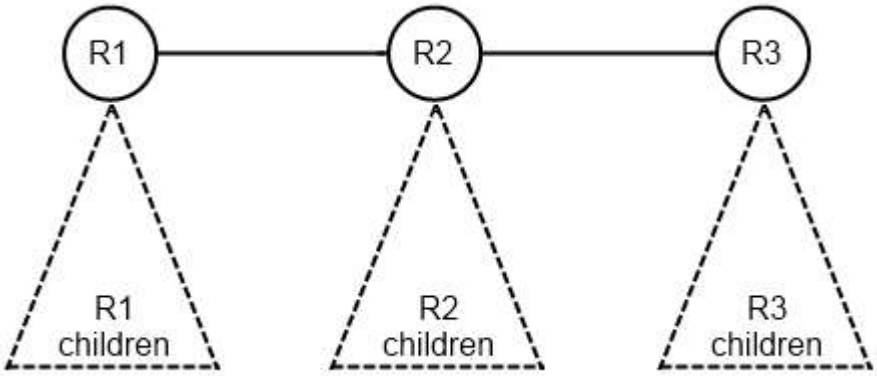


figure 3 minimum broadcast coverage example topology

Packet addressing: The packet addressing in this implementation of ZigBee is done only via the extended 64bit ZigBee address of the destination node.

Topology awareness: Each node knows its extended, 64bit ZigBee address and its 16bit network address. End-devices also know their parent's 16bit network address.

ND AT command: Through this command a node can:

- receive descriptions from each node containing information about its 64bit and 16bit addresses, parent 16bit address (only available on end-device nodes), node type (ZR/ZED), specified identifier (string), etc.
- receive descriptions only from the nodes that have a specified identifier (string) or 16bit address.

4. Solution overview

The solution was designed and implemented incrementally in three versions. For this purpose, a custom "Remote reprogramming protocol" was designed and adjusted to the requirements of each version of the solution. This chapter offers a general description of the update procedure[2], followed by an overview of major design options taken into account by all three versions, to address the implementation platform constraints. Any version-specific design choices are discussed in each of the respective chapters. Finally, a brief presentation of the improvement achieved step by step with this incremental development method prepares the reader for the following chapters, which focus on each version.

4.1. Update procedure, general description

The update procedure is initiated by the client program running on a PC, able to communicate with the wireless sensor nodes through the Quax Gateway. The Quax nodes are re-programmed by exchanging messages of the respective version of the custom "Remote reprogramming protocol" with the PC client program and overwriting the running code image with the new one.

This procedure is mainly separated in two parts:

- Code image download: The node receives the new code image in chunks, storing them temporarily in non-volatile memory.
- Image overwrite and reboot: After the new code image is successfully downloaded, the node copies it non-interruptively over the running image and reboots (given the node's ability of in-system programming).

4.2. Important design options

As mentioned in the respective chapter, there are two types of target nodes (Quax MS and DT), so the solution is bound by the type that introduces the most constraints. Regarding the available memory, the Quax DT node is the most constrained, since it does have any external memory in contrast with Quax MS. Consequently, the non-volatile memory that can be used is 48 Kbytes (program memory) plus the 256 bytes of the info memory.

The code residing on the Quax nodes is divided in two parts: ISOS (i.e. hardware drivers, interrupt handler etc) and application-specific. The first is present in every code image, whereas the latter can be different depending on the purpose of each code image (e.g. temperature, humidity, light measurement etc). Therefore, both full (entire image) and partial[3] (application-specific part only) code update are feasible. As a first approach, the full code update was chosen, leaving the partial code update as an improvement for later versions of the solution. This improvement is also motivated by the fact that the ISOS code part may rarely (if at all) change, after the initial deployment.

Another important issue is whether to halt the application during the update procedure. The decision made for this matter is to allow application code execution, in order for the code update procedure to be as non-intrusive to the application as possible. To achieve this, the code chunks must be stored temporarily until the complete new image is downloaded, before it is written over the old one. An important constraint on this approach is that the new image can only be as big as the free program memory available at runtime on the node. Moreover, the application may seriously delay the overall reprogramming procedure. On the plus side, the code update can be aborted at any time by either side, without affecting the application.

Since the downloaded image is first buffered in non-volatile memory and then written over the old one, it only makes sense to support the continuation of a previously interrupted code download. After all, the "Remote reprogramming protocol" was designed from the beginning with this aspect in mind. To clarify, the state about the amount of the new image part already downloaded and buffered in non-volatile memory must be stored in permanent memory too. This way, the update procedure can handle communication loss or even a node reboot (e.g. due to an unexpected runtime error).

---

[2] The terms “update procedure” and “remote reprogramming procedure” are considered identical and will be used interchangeably.

[3] A “partial update” is also known as “incremental update” in literature.

4.3. Incremental development method overview

The solution was designed and implemented in an incremental fashion to gradually refine and improve the functionality offered. During this process, three versions were produced, each one adding new or improving the existent features of the solution.

The first approach (version 1) updates the entire code image of the Quax node via a stop-and-wait protocol. This way, the entire WSN is reprogrammed node-by-node. Additionally, the application code is not halted until the entire new image is downloaded, while it is possible to resume a previous download. Nevertheless, both the currently running and the new image must fit in the program memory, limiting the application's functionality.

In version 2, the key improvement is that only the application-specific part of the image is updated, while the ISOS part is left intact. Of course, any inconsistency between the ISOS image part that the update was compiled for and the ISOS image part present on the Quax node is handled by not allowing the update to proceed and cause unfortunate results. Additionally, the overwrite procedure was modified to be reboot-resistant and the integrity of the information kept in non-volatile memory (regarding both the current and the downloaded images) is checked via CRC, resulting in a more reliable solution. The rest of the first version's functionality is retained.

The last version inherits the features of version 2, except that the update protocol is no longer stop-and-wait. Specifically, the update procedure is performed in successive propagate-repair phases, where the client transmits a number of code chunks comprising a code unit of 512 bytes without any replies sent by the Quax node. A subsequent report collection round is initiated by the client and any lost chunk packets are identified and retransmitted. This non-ARQ, NACK based propagation scheme enables one-to-many updates, which utilize the native ZigBee broadcasts and reprogram all the nodes at once. Moreover, the aforementioned scheme improves the point-to-point update latency as well.

5. Solution, version 1

5.1. Design

Regarding the design options discussed in subchapter 4.2, this version updates the entire code running on the Quax node in a non-intrusive fashion to the application, while supporting the continuation of a previously interrupted code update. The update procedure can be aborted by both sides (PC client program or Quax node), in case of communication problems (timeout expiration) or explicit user input (on the client).

5.2. Update procedure

The update procedure is initiated by the client program running on a PC, able to communicate with the wireless sensor nodes through the Quax Gateway. The Quax nodes are reprogrammed in a node-by-node fashion, by exchanging messages of the stop-and-wait and point-to-point "Remote Reprogramming Protocol for Prisma WSN – v10" (see respective section in Appendix) with the PC client program and overwriting the currently running code image with the new one (figure 1)[4].

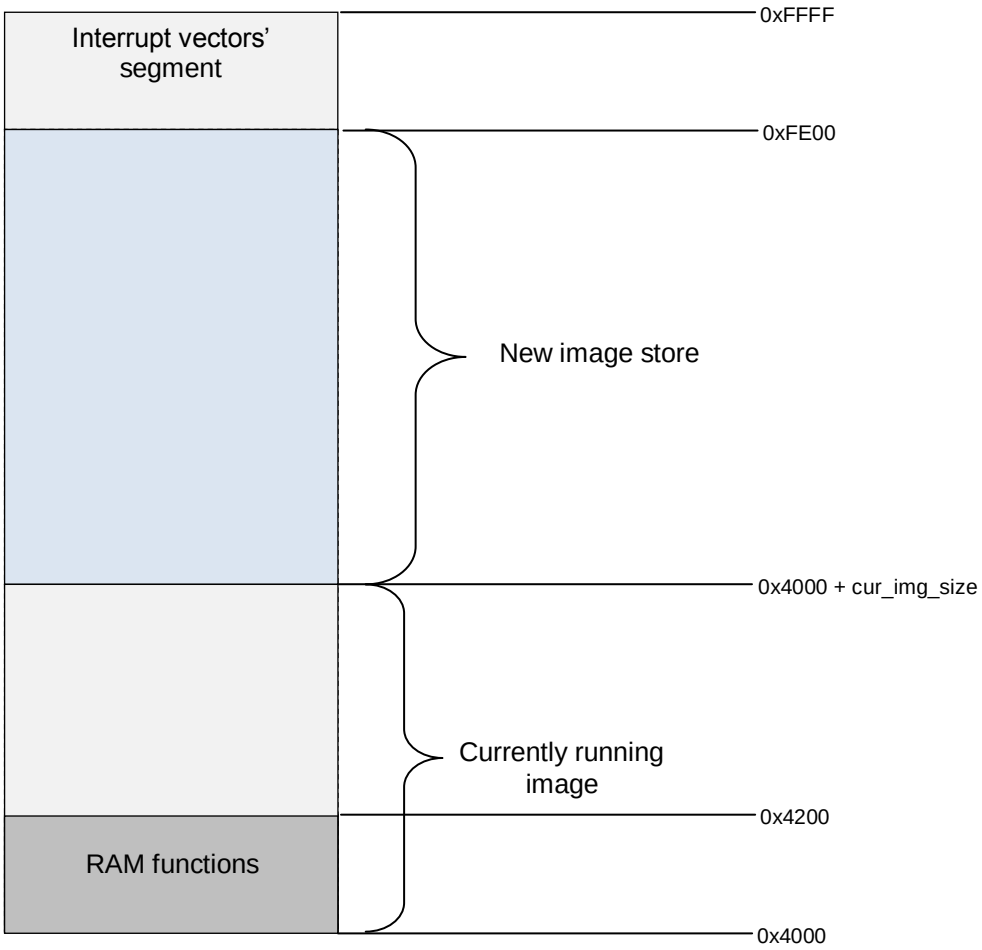


figure 1 version 1, program memory map

[4] The "RAM functions" are permanently stored in flash memory, but copied to and ran from RAM in order to perform flash write operations.

First, the client synchronizes the communication with the node, to find out when the node's radio is ready to receive messages (more details on synchronization in subchapter 5.3 and in Appendix on "Remote Reprogramming Protocol for Prisma WSN – v10"). Then, the client sends a "Get Version" request to retrieve the version number of the image running on the node. The node replies with the version number of the currently running image and the client can decide whether an update is necessary by comparing the version number in the reply with the update's version number.

If the node is not running the latest image, the client sends a "Start Session" request to establish a new code update session. The node replies indicating whether it is ready to proceed or the image is too big to be buffered in the flash memory together with the current image. If the node is ready, it sends the number of the last code chunk already received (this is to support a continuation of a previous update).

Afterwards, the client sends the new image in chunks and waits for an explicit acknowledgement from the node for each chunk before sending the next one, until the node receives all the chunks. The downloaded part of the new code image and information about the download status are saved in non-volatile memory (flash). Therefore, it is possible to continue a previously interrupted code update session.

Once the image is successfully uploaded on the node, the client requests the node to commit and start running it. The node verifies that the CRC digest of the downloaded image is correct (the client sends the digest in the "Start Session" request). If the check succeeds, the node sends a positive reply, application execution is stopped, all interrupts are disabled, and the old image is overwritten with the new, downloaded image, resulting in a full code update. The node reboots when the overwrite operation completes, and starts executing the new image. The current image version and size values are stored in non-volatile info memory and updated before rebooting to the new image. If the digest check fails, the node sends a negative reply, and the client re-synchronizes the communication, and initiates a new download session.

The client also initiates a new session whenever communication with the node is lost (client timed out waiting for message), or if the node has silently aborted the current session (node timed out waiting for message) and replies with a "session aborted" message. Again, the client must first synchronize the communication. Notably, a new session always overrides an old one.

Diagrams

The update procedure is illustrated in the following state-transition diagrams, one for each side (figure 2 for the PC client side, figure 3 for the Quax node side).

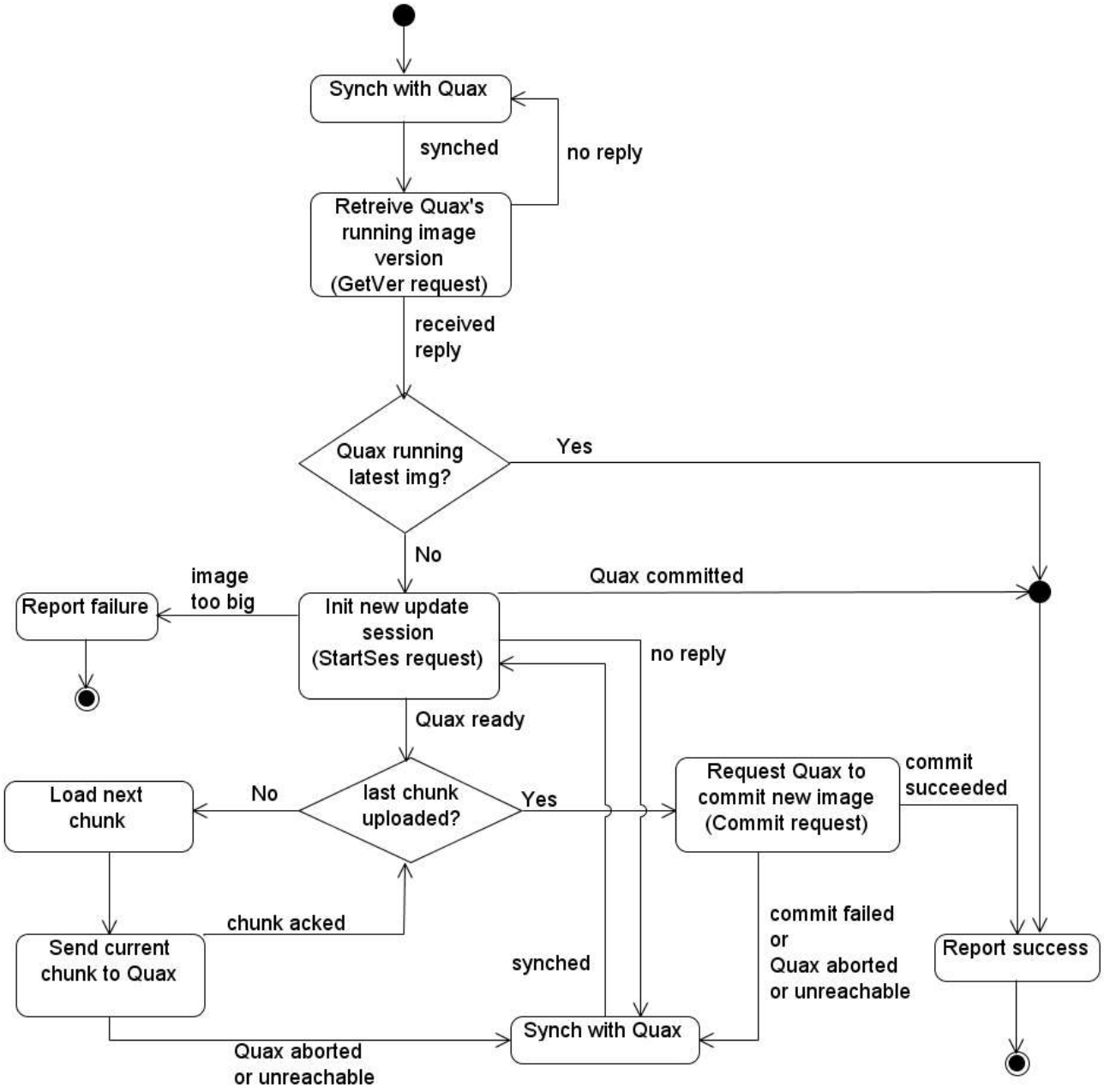


figure 2 version 1, PC client state-transition diagram

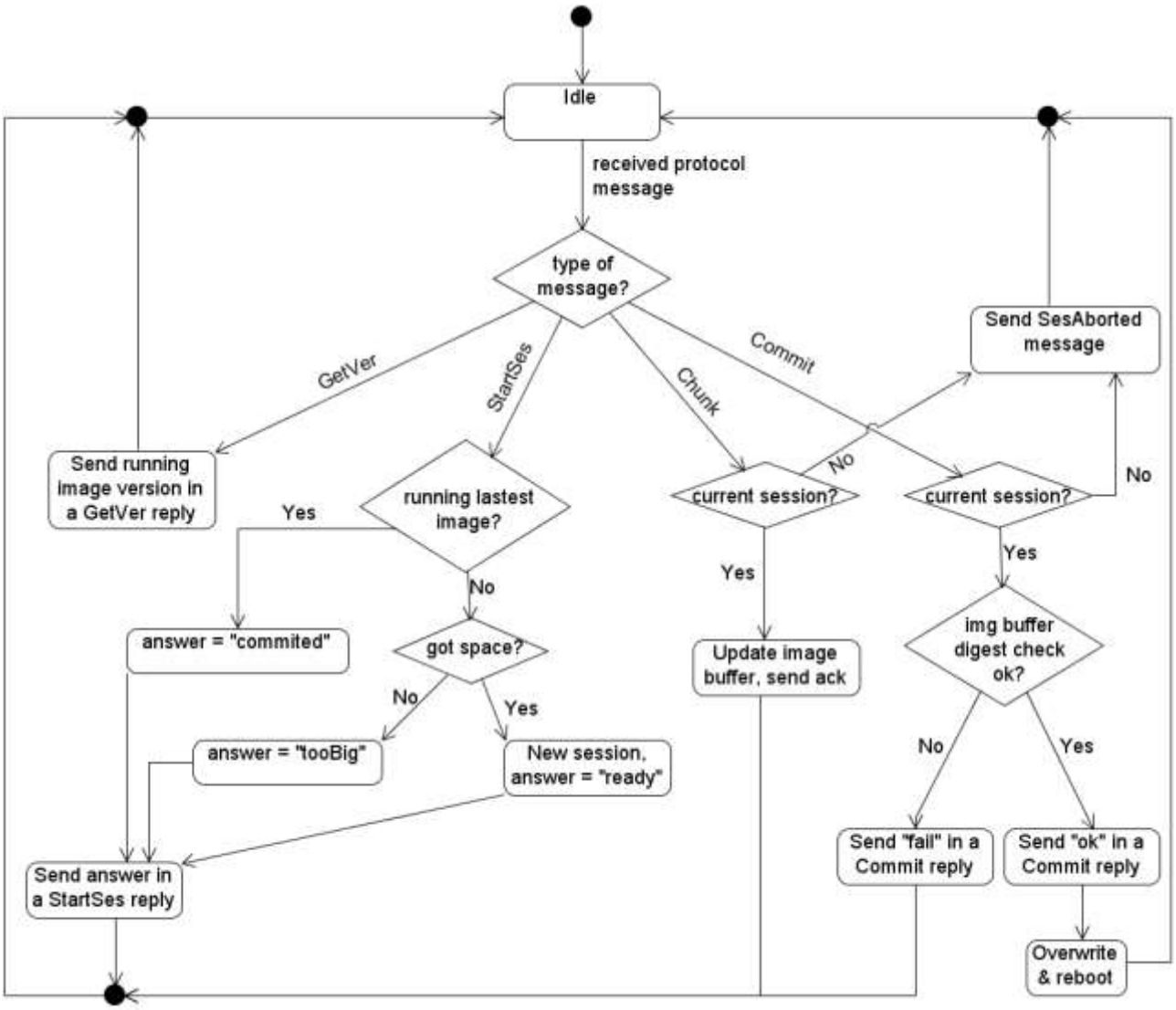


figure 3 version 1, Quax node state-transition diagram

Additionally, several indicative sequence diagrams are given below with a brief description under each diagram (figures 4-7).

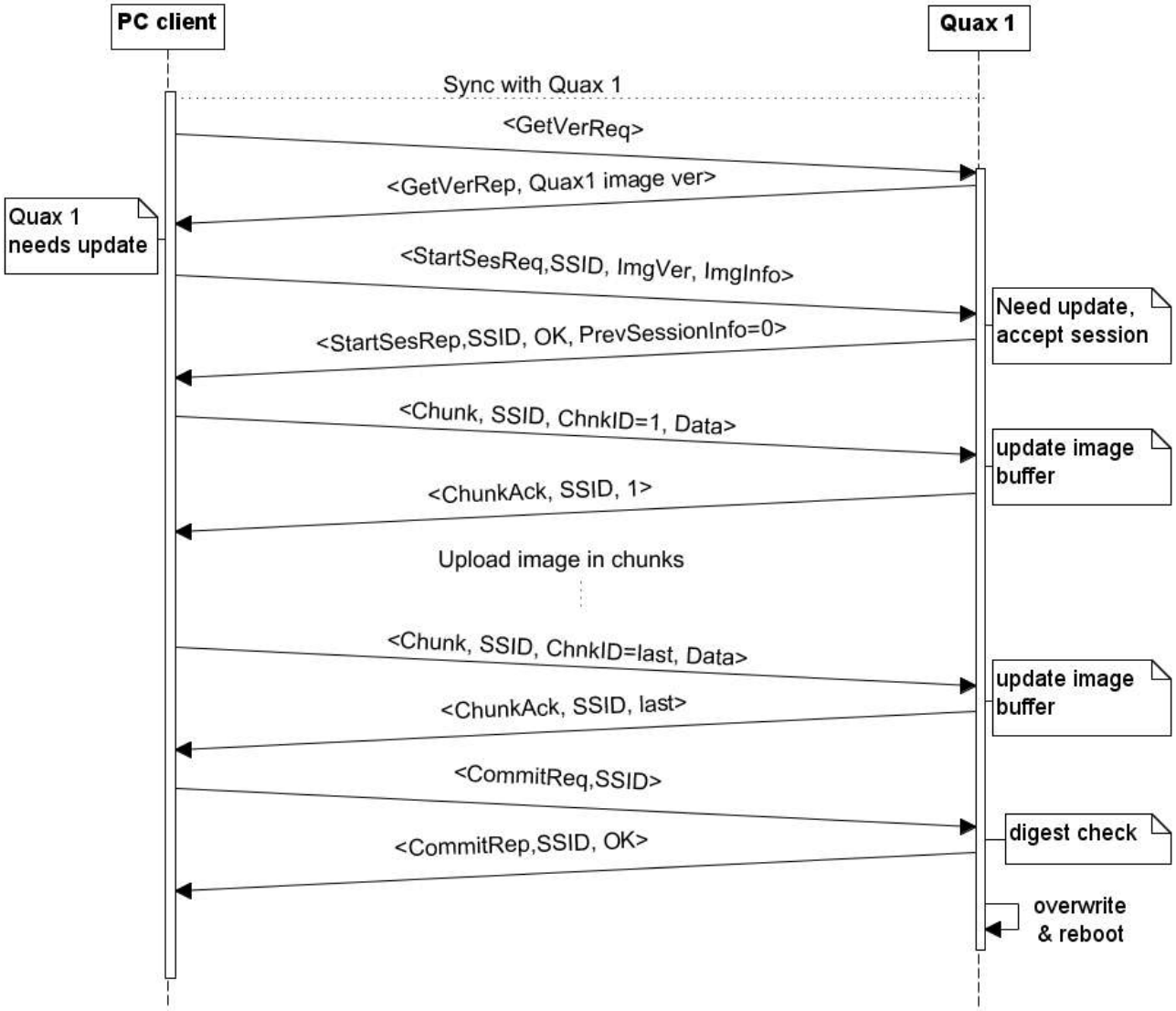


figure 4 version 1, regular flow of update procedure

Figure 4 shows the regular flow of the update procedure in version 1, where the image currently running on Quax 1 is of older version than the image update. The client first synchronizes the communication with Quax 1 (omitted) and retrieves the version of the image currently running on the Quax. Then, since Quax 1 needs an update, the client initiates an update session via a “Start Session” request-reply and uploads the new image via subsequent chunk messages, each one acknowledged separately. After the last chunk is uploaded and acknowledged, the client requests from Quax 1 to commit to the new image. Finally, Quax 1 confirms the integrity of the downloaded image (digest check), sends a positive Commit reply, overwrites the currently running image with the new one and reboots to the latter.

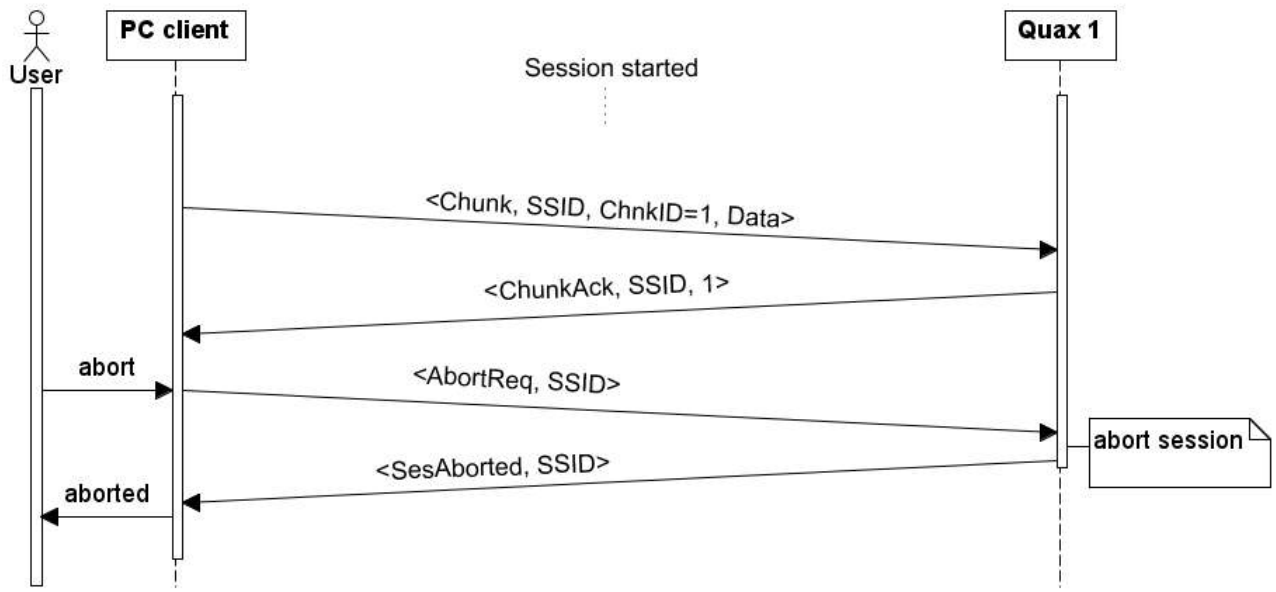


figure 5 version 1, session abortion by PC client

Figure 5 depicts the case of a user-initiated abortion of an update session, omitting the synchronization, version retrieval and session initialization parts. The user commands the PC client to abort the undergoing update session via the user interface and the client sends an Abort request to Quax 1. Then, Quax 1 aborts the current session and sends a “Session aborted” reply back to the client.

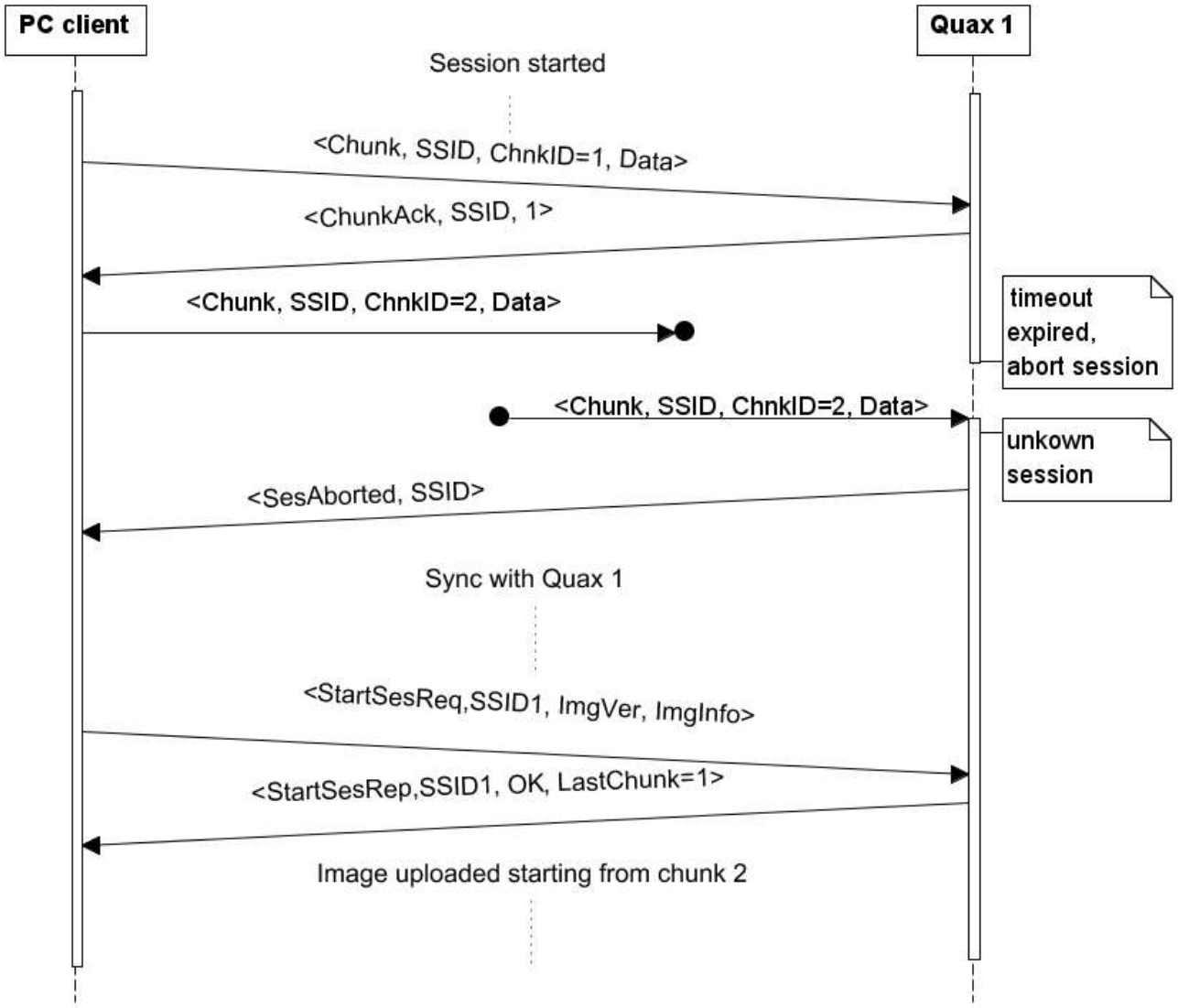


figure 6 version 1, session abortion by Quax

In figure 6, the update procedure goes on normally, until temporary communication problems cause a chunk message transmission to fail. Although (through network-level retransmissions) the chunk message finally reaches the Quax, the latter has already silently aborted the update session, as the timeout for protocol message reception has expired. Thus, Quax 1 replies properly and notifies the client about the abortion. Then, the client re-synchronizes communication with Quax 1, initializes a new update session and resumes the code upload from the point it was interrupted (chunk #2).

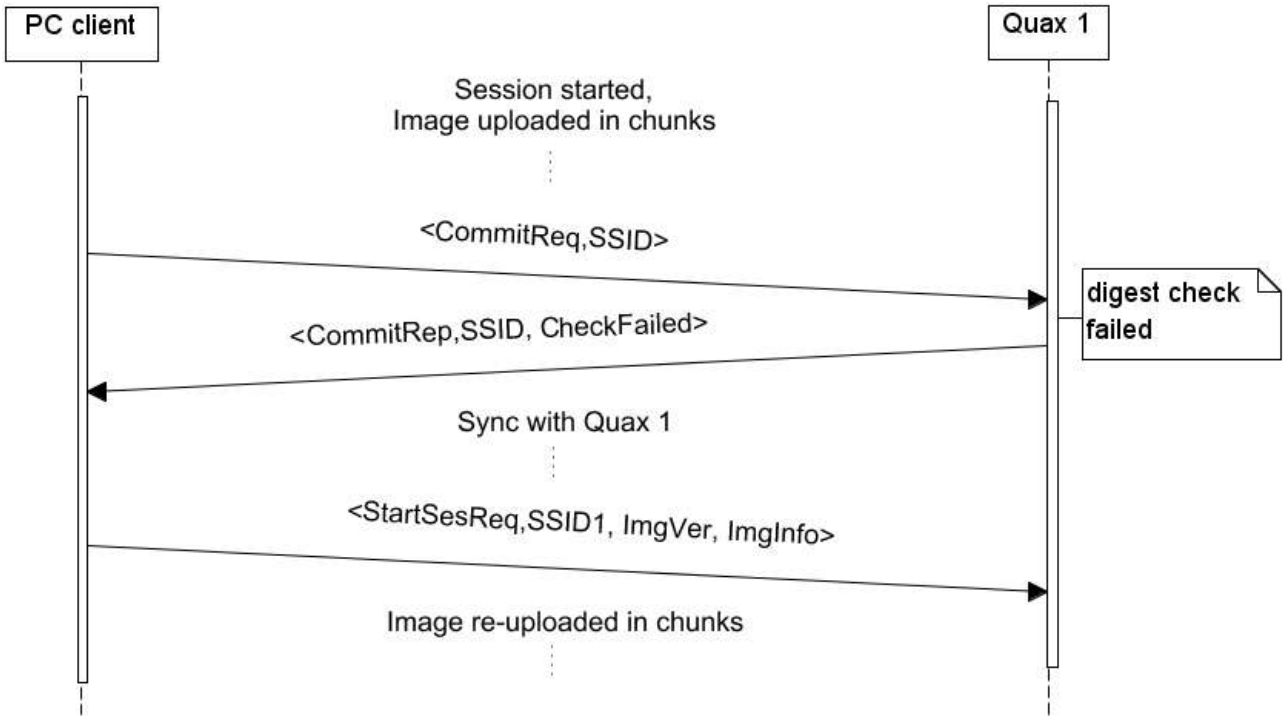


figure 7 version 1, digest check failure

In figure 7, the update procedure goes on normally (as in figure 2) up until the digest check failure on Quax 1. Quax 1 informs the client about the failure via a proper Commit reply ("CheckFailed") and causes the entire procedure to repeat. That is, the client synchronizes communication with Quax 1 anew, starts a new update session and uploads the entire new image to the Quax again.

5.3. Implementation

The PC client is a console application, written in C#, implementing the client side of the "Remote reprogramming protocol" (see Appendix on “Remote Reprogramming Protocol for Prisma WSN – v10”). It is configured via two configuration files, one containing information about the nodes (ZigBee addresses) and one containing information about the code image file (version and path to the code image file). Moreover, the code image file must comply with a specific format. The client allows the user to take control of simple tasks such as starting/resuming the update of a list of Quax nodes, selecting a Quax to update, or aborting an update session.

The code residing on the Quax is written in the C implementation of the MSP430 microcontroller unit, implements the node side of the "Remote reprogramming protocol" (see Appendix on “Remote Reprogramming Protocol for Prisma WSN – v10”) and runs as a part of the ISOS. The ISOS code handling message reception was modified in order to distinguish between application messages and remote reprogramming messages, handing them to the application and the remote reprogramming component respectively.

Regarding the communication synchronization schema and the online-offline strategy of a ZigBee end-device node, two simple rules are followed. The first rule guarantees that the node must turn its radio on at least once during a specified offline period and that every time it does so, the offline period is reset and a special synch message is sent to the PC. In case the radio was turned on by the application to send an application-level message, the latter serves as a synch message as well. The second rule guarantees that whenever the radio is turned on, either by the application or by the remote reprogramming component (first rule), it will stay on for at least a specified online period. Finally, the online period is set to an extended value whenever the node receives a remote reprogramming message.

The communication synchronization schema and the online-offline strategy of a ZigBee router node is much simpler, since such a node keeps its radio on all the time. Thus, a router node only keeps track of one period (the offline period), resets the period whenever the application sends a message or whenever the period expires (also sending an explicit synch message).

The described online-offline strategy guarantees that whenever the client wishes to update a Quax, it will not wait for more than an offline period to start the update session.

Pseudocode

A high-level pseudocode for each side is presented below (more detailed pseudocodes, also handling periods/timeouts, in the Appendix on “Remote Reprogramming Protocol for Prisma WSN – v10”).

PC client pseudocode:

```
updateQuax(Quax) {
    synch communication;
    do {
        send GETVER request to Quax;
    }while ( ! received GETVER reply );
    if ( Quax runs the latest image ) { return success; }
    state = NEWSES;
    while(1) {
        switch( state ) {
            case NEWSES: {
                do {
                    send NEWSES request to Quax;
                } while( ! received NEWSES reply from Quax );
                if ( answer == TOO_BIG ) { return failure; }
                if ( answer == COMMITED ) { return success; }
                if ( Quax has the entire image ) state = COMMIT;
                else state = NXTCHUNK;
            }
            case NXTCHUNK: {
                state = COMMIT;
                foreach chunk of the image {
                    do {
                        send chunk in a NXTCHUNK request to Quax;
                    } while( ! received ABORTED reply && ! received NXTCHUNK reply );
                    if ( Quax aborted ) {synch communication;state = NEW_SES;}
                }
            }
            case COMMIT: {
                do {
                    send COMMIT request to Quax;
                } while(! received ABORTED reply && ! received COMMIT reply);
                if ( Quax aborted ) {synch communication;state = NEW_SES;}
                else if ( check failed on Quax) {synch communication;state = NEW_SES;}
                else return success;
            }
        }
    }
}
```

Quax pseudocode:

```
receive_handler( msg ) {
    switch( msg ) {
        case GETVER request: {
            send running image version in a GETVER reply;
        }
        case NEWSES request: {
            if( not enough space )
                answer = TOO_BIG;
            else if ( already running this image)
                answer = COMMITED;
            else {
                if ( this is a new update )
                {
                    initialize download data;
                }
                start new session;
                answer = <READY, last_chunk_recvd_id>;
            }
            send answer in a NEWSES reply;
        }
        case NXTCHUNK request: {
            if( request belongs to unknown session)
                send ABORTED reply;
            else if( chunk is in order){
                save chunk;
                update last_chunk_recvd_id;
            }
            send last_chunk_recvd_id in a NXTCHUNK reply;
        }
        case COMMIT request: {
            if( request belongs to unknown session)
                send ABORTED reply;
            else if ( digest check fails) {
                erase download data;
                send CHECK_FAILED in a COMMIT reply;
            }
            else {
                send OK in a COMMIT reply;
                overwrite old image with new and reboot;
            }
        }
        case ABORT request: {
            if( request belongs to current session)
                send ABORTED reply;
        }
    }
}
```

6. Solution, version 2

6.1. Design

This version retains the functionality of version 1 regarding the non-intrusive fashion of the update procedure, continuation of a previously interrupted code update and abortion of the update procedure by both sides. However, only the application-specific part of the image[5] running in the Quax is updated (partial code update[6]), while the ISOS part is left intact. Thus, the constraints on the application's size and functionality are far looser than those of version 1. Additionally, to avoid human-generated mistakes in the process of versioning, the version numbers of the two parts (core and application-specific) are computed by software as the CRC digest of the corresponding binary images.

6.2. Update procedure

The update procedure is initiated by the PC client, as in version 1, which reprograms the Quax nodes in a one-by-one fashion. The two sides exchange messages of the stop-and-wait and point-to-point "Remote Reprogramming Protocol for Prisma WSN – v12" (thorough description, also addressing compatibility issues with version 1, in the respective section of the Appendix) and when the download completes, the currently running application-specific image is overwritten with the new one (figure 1).

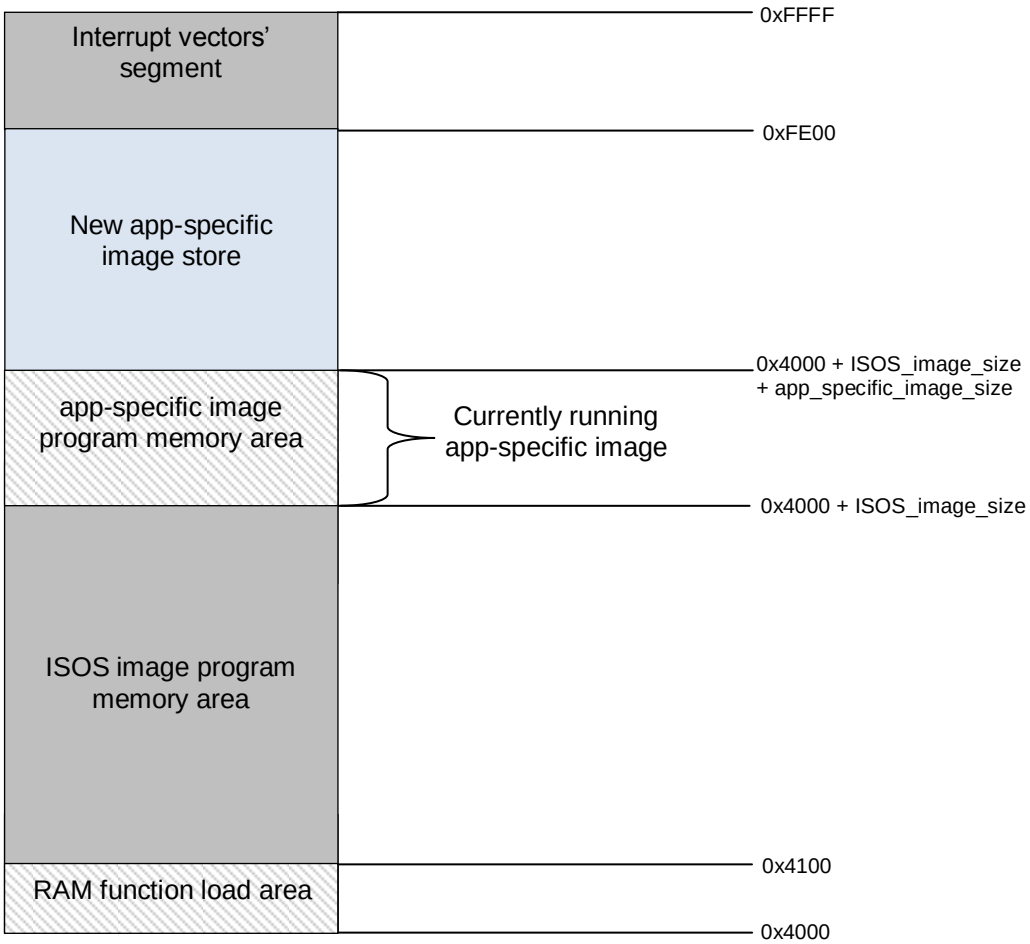


figure 1 version 2, program memory map

[5] The terms "application-specific part of the image"/ "application-specific image" will be used interchangably, as will the terms "ISOS part of the image" / "ISOS image".

[6] Known as incremental update in literature.

Initially, the client synchronizes the communication with the node. Then, it sends a “Get Version” request to retrieve the ISOS and application-specific version numbers of the images running on the node. The node replies with the version numbers of the ISOS image and the currently running application-specific image, and the client can decide whether the update is feasible and necessary by comparing the version numbers in the reply with the update’s version numbers. If the ISOS code running on the node is not the same as the one required by the application (not the ISOS version the application was compiled for), the client does not proceed to a code update session with the node.

If the node has the required ISOS code and is not running the latest application-specific image, the client proceeds to a new code update session and uploads the code to the node as usual (explicit acknowledgements for each chunk). Again, flash is used as a buffer for the download data and state.

Once the download completes, the client requests the node to commit and start running it. In contrast to version 1, the node first verifies that the CRC digest of the downloaded image (in the buffer) is correct, performs the overwrite procedure and then verifies that the CRC digest of the overwritten application-specific image (in the application-specific program memory area) is correct too. If any of the two digest checks fails, the node sends a negative reply. Additionally, if the second check failed, all jumps to the application-specific program memory area are prevented and only the core image continues running. This way the client can detect the failure and initiate a new download session to fix it. If both checks succeed, the node sends a positive reply, updates the current image info held in flash (versions and size) and reboots to the new image.

After the node has successfully committed to reboot to the new image, the client synchronizes the communication with the node again, and initiates a new “Get Version” message exchange to verify the success of the reboot as well.

In case of communication problems with the node or session abortion by the node, a new session is initiated by the client, as in version 1.

The update procedure is illustrated in the following state-transition diagrams, one for each side (figure 2 for the PC client side, figure 3 for the Quax node side).

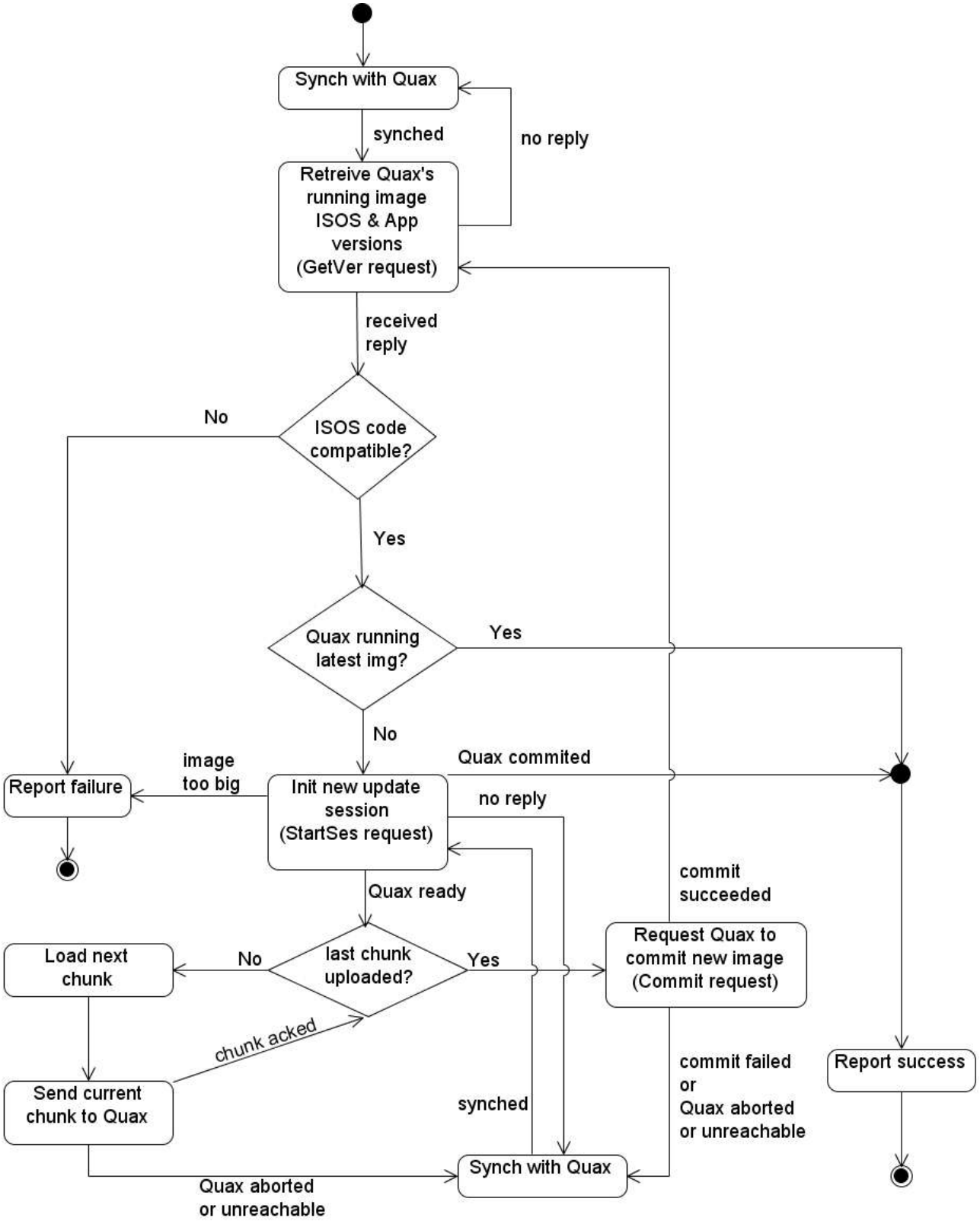


figure 2 version 2, PC client state-transition diagram

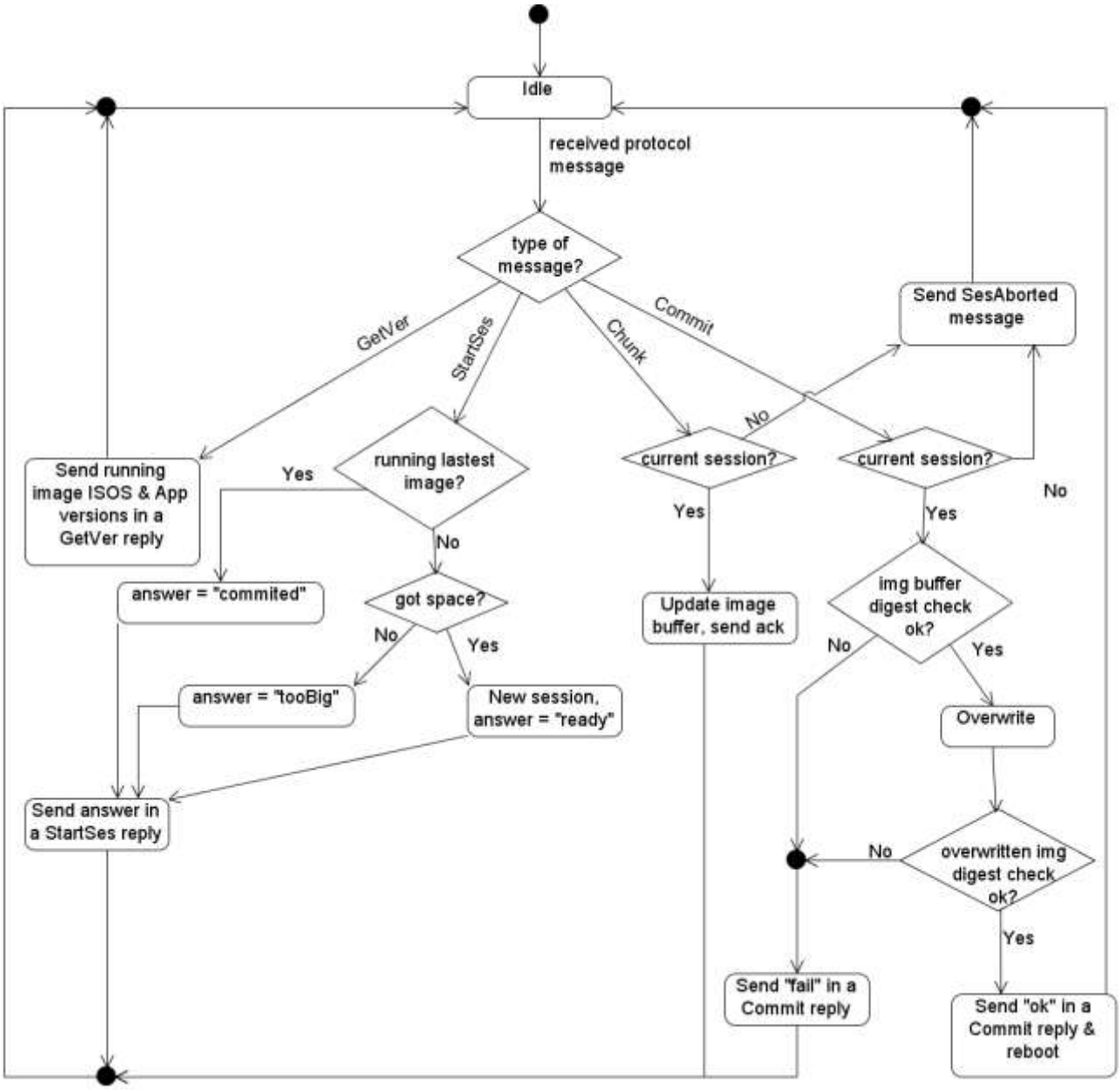


figure 3 version 2, Quax node state-transition diagram

Moreover, several indicative sequence diagrams are given below, with a brief description under each diagram (figures 4-5).

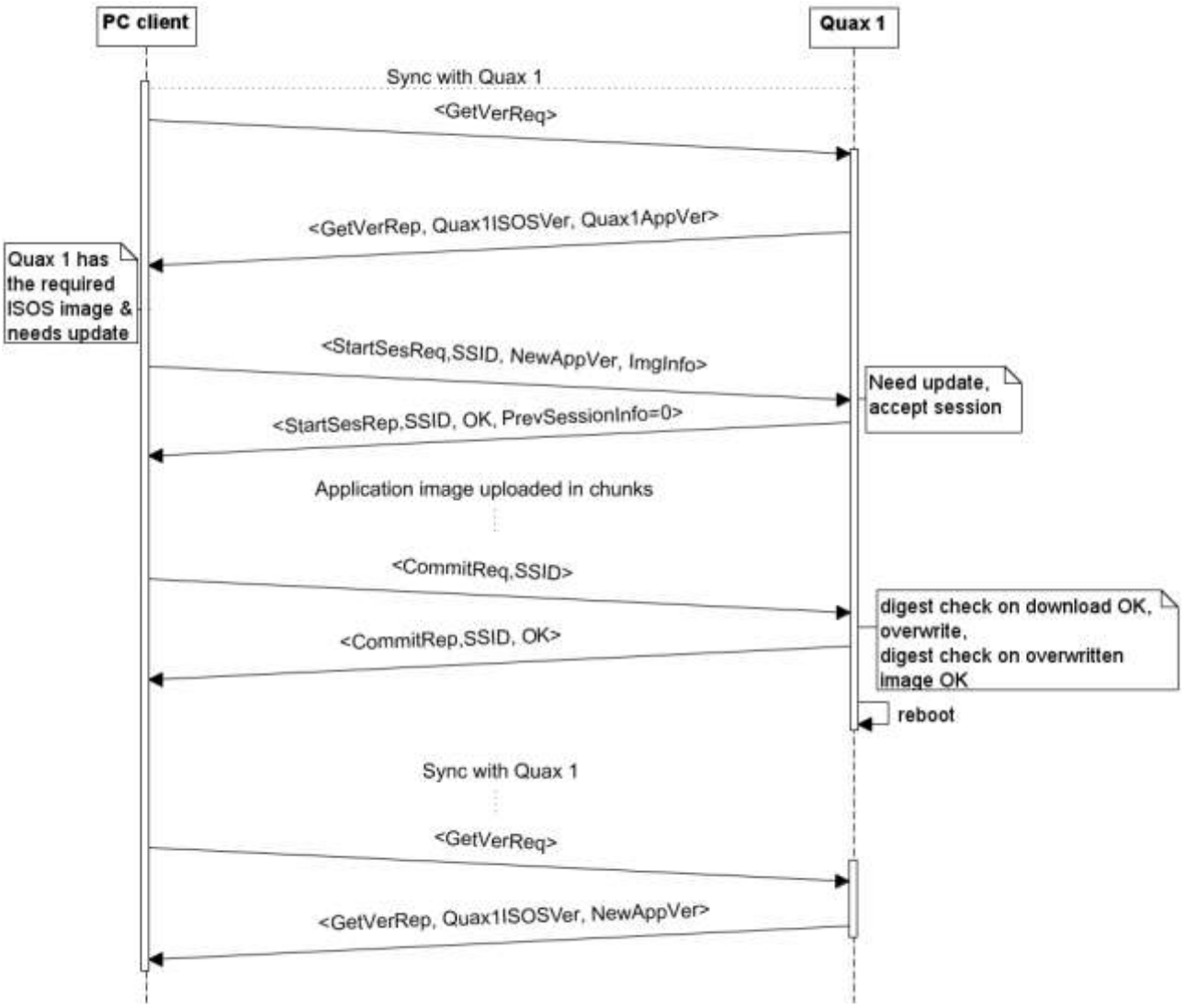


figure 4 version 2, regular flow of update procedure

Figure 4 shows the regular flow of the update procedure in version 2, where the application-specific image currently running on Quax 1 is of older version than the image update. The client first synchronizes the communication with Quax 1 (omitted) and retrieves the ISOS and application-specific versions of the Quax's currently running image. Then, since Quax 1has the required ISOS code and needs the update, the client initiates an update session ("Start Session" request-reply) and uploads the new image via subsequent chunk messages, each one acknowledged separately. After the image is uploaded, the client requests from Quax 1 to commit to it. Quax 1 confirms the integrity of the new image (digest checks pre and post overwrite), sends a positive Commit reply, and reboots to the new image. Finally, the client confirms the success of the reboot by retrieving the Quax's image versions anew.

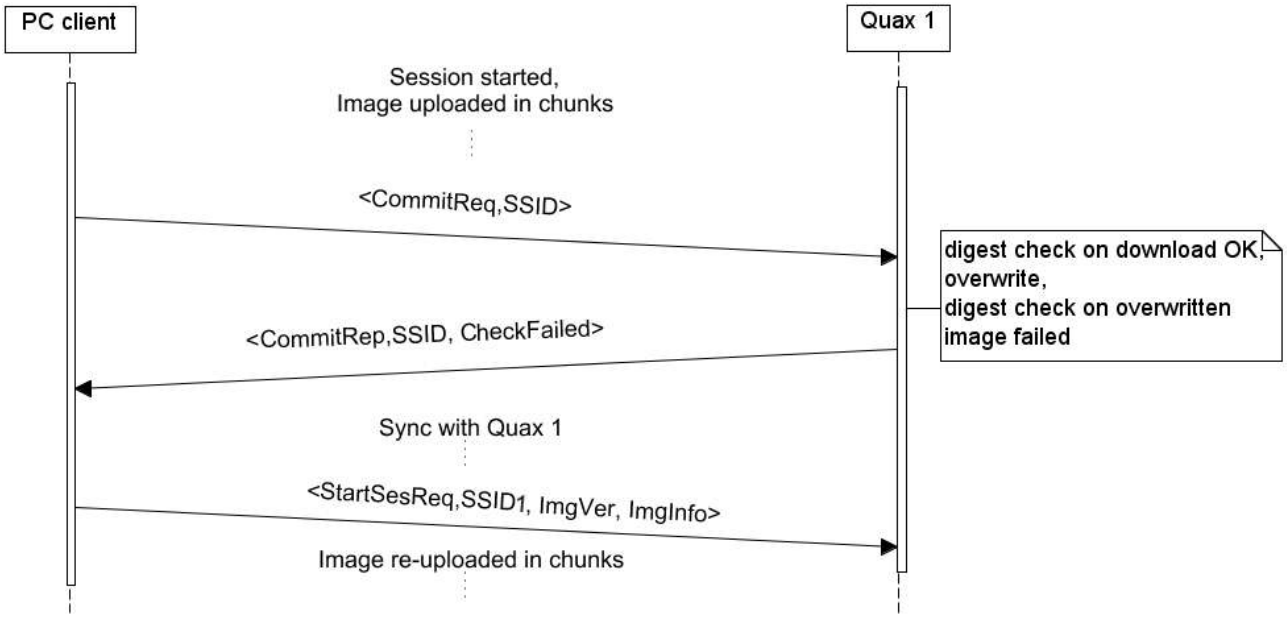


figure 5 version 2, digest check on overwritten image failure

In figure 5, the update procedure goes on normally (as in figure 2) up until the digest check on the overwritten application-specific image. The aforementioned check fails (e.g. due to an error during the image overwrite), Quax 1 informs the client about the failure via a proper Commit reply (“CheckFailed”) and forces the entire procedure to repeat. That is, the client synchronizes communication with Quax 1 anew, starts a new update session and uploads the entire new image to the Quax again.

6.3. Implementation

PC client code modifications

Using the information retrieved from the configuration files, the PC client computes the core and application-specific image versions (CRC) and loads only the application-specific part from the image file.

Quax code modifications

Updating only a part of the code running on a Quax node and expecting it work is no trivial task. Specifically, the following “tricks” were necessary:

- All components (source files) of the ISOS image are “dummy” referenced from inside the ISOS image, in order to guarantee that it remains the same on every build/compile and the compiler will not exclude any components, regardless of the application.
- The linker is configured to statically allocate all the functions and variables of the ISOS image at specific addresses in program memory and RAM, to avoid inconsistencies when referencing ISOS image functions from inside the application-specific image.
- A wrapper function that is statically allocated at the start of the application-specific area in the program memory is used to implicitly call application-specific functions from ISOS code image. Since these application-specific functions are free to modification for the purposes of each application, their actual allocation in the application-specific area of the program memory might changes from application to application. The wrapper function is also part of the application-specific image and compiled/linked to the rest of the application-specific functions on every build, while its own allocation is static.
- No constants (e.g. static arrays or stings) of the application-specific image can be used by ISOS code image. This must be done by copying the constant to a buffer variable and then handing it to ISOS code image.
- All global variables of the application-specific image must be initialized from within a function.

Nevertheless, another important improvement is that in this version, the overwrite operation is done in a stateful fashion and is able to continue even after an accidental reboot. For this reason, it saves and recovers its state to and from non-volatile memory. In addition, the information held in non-volatile memory (current image version and size, download status, overwrite procedure state) is checked for integrity via CRC, increasing the system’s reliability.

Regarding the computation of the core and application-specific version numbers, a custom tool was developed and added to the IDE toolchain. This way, the computation is done automatically on every build/compile, but the initialization of each Quax node with the two version numbers needs to be done by hand, along with the initial image burn.

Additionally, since all core functions and interrupt service routines are permanently present on the node (ISOS image), the interrupt vector table remains unchanged, regardless of the changes in the application-specific image. Thus, it is not uploaded and re-programmed, resulting in more stability for the system.

Finally, the online-offline strategy on each node is implemented as in version 1.

Pseudocode

A high-level pseudocode for each side is presented below (more detailed pseudocodes, also handling periods/timeouts and compatibility issues with version 1, in the Appendix on “Remote Reprogramming Protocol for Prisma WSN – v12”).

PC client pseudocode:

```
updateQuax(Quax) {
    synch communication;
    while(1) {
        switch( state ) {
            case GETVER:
                do {
                    send GETVER request to Quax;
                }while ( ! received GETVER reply );
                if ( Quax runs the latest image ) { return success; }
                state = NEWSES;
            case NEWSES: {
                do {
                    send NEWSES request to Quax;
                } while( ! received NEWSES reply from Quax );
                if ( answer == TOO_BIG ) { return failure; }
                if ( answer == COMMITED ) { return success; }
                if ( Quax has the entire image ) state = COMMIT;
                else state = NXTCHUNK;
            }
            case NXTCHUNK: {
                state = COMMIT;
                foreach chunk of the image {
                    do {
                        send chunk in a NXTCHUNK request to Quax;
                    } while( ! received ABORTED reply && ! received NXTCHUNK reply );
                    if ( Quax aborted ) {synch communication;state = NEW_SES;}
                }
            }
            case COMMIT: {
                do {
                    send COMMIT request to Quax;
                } while( ! received ABORTED reply && ! received COMMIT reply );
                if ( Quax aborted ) {synch communication;state = NEW_SES;}
                else if ( check failed on Quax) {synch communication;state = NEW_SES;}
                else state = GETVER;
            }
        }
    }
}
```

Quax pseudocode:

```
receive_handler( msg ) {
    switch( msg ) {
        case GETVER request: {
            send GETVER request;
        }
        case NEWSES request: {
            if( not enough space )
                answer = TOO_BIG;
            else if ( already running this image)
                answer = COMMITED;
            else {
                if ( this is a new update )
                {
                    initialize download data;
                }
                start new session;
                answer = <READY, last_chunk_recvd_id>;
            }
            send answer in a NEWSES reply;
        }
        case NXTCHUNK request: {
            if( request belongs to unknown session)
                send ABORTED reply;
            else if( chunk is in order){
                save chunk;
                update last_chunk_recvd_id;
            }
            send last_chunk_recvd_id in a NXTCHUNK reply;
        }
        case COMMIT request: {
            if( request belongs to unknown session)
                send ABORTED reply;
            else if ( download digest check fails) {
                erase download data;
                send CHECK_FAILED in a COMMIT reply;
            }
            else {
                overwrite old image with new;
                if(overwrite digest check fails){
                    send CHECK_FAILED in a COMMIT reply;
                    halt application;
                }
                else {
                    send OK in a COMMIT reply;
                    reboot;
                }
            }
        }
```

```
        case ABORT request: {
            if( request belongs to current session)
                send ABORTED reply;
        }
    }
}
```

7. Solution, version 3

7.1. Design

This version enables one-to-many code updates, to reprogram all the nodes at once, and reduce the overall latency when updating large networks. In short, the application-specific image is uploaded to all Quax nodes via efficient and controlled flooding, exploiting the native ZigBee broadcasts. The following paragraphs briefly describe the approaches considered and the obstacles met.

Initial approach

Obviously, since the propagation of the image chunks is the lengthiest part of the update procedure, its stop-and-wait scheme had to be replaced by a more scalable, non-ARQ one. As in Deluge [5] and other similar previous work, the image is split in fixed-size transfer units. Each of these transfer units consists of a fixed number of code chunks (chunk size is also fixed) and is propagated by the PC node via network-wide broadcasts, with the Quax nodes remaining quiet throughout this phase. Once the propagation of a transfer unit is over, the PC node initiates a repair phase, where each router Quax node repairs any missing chunks of the current transfer unit locally for its children. Specifically, the router iteratively requests reception reports from all its children and broadcasts locally any missing chunks, until its children report no missing chunks. Afterwards, a reversed-tree reporting scheme is used, in order for the PC node to determine when the entire network is ready for the propagation of the next transfer unit, or for the committing of the update. All other requests, regarding establishing/committing a code update session or retrieving the versions of the images running on the nodes, are broadcasted network-wide by the PC node and the respective replies are unicasted by each Quax node back to the PC. Since, the latter requests are sent ideally only once for each code update, they do not harm the overall latency of the procedure.

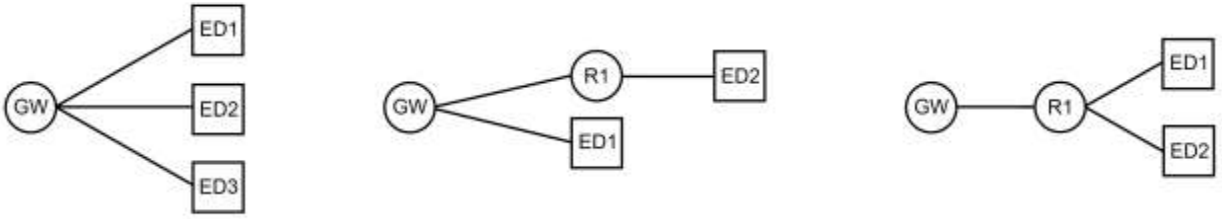
This initial approach was abandoned mainly due to the topology-awareness it requires from each Quax node (local repairs and reverse-tree reporting). Although an ND AT command can discover all end-device children of each router, it cannot discover the router children of a router, as routers are not aware of their parent (see subchapter 3.6). Consequently, a custom discovery phase is necessary before the actual update procedure starts, so that all routers become aware of their parent and their router children. During this phase, each router would scan its 1-hop neighborhood for router children. Thus, since actual one-hop broadcasts are not possible, a router could be assigned with a parent that is two hops away, resulting in an inefficient inferred topology. To make it worse, the chunks transmitted by neighboring router nodes during the repair phase would interfere with each other, causing further delay in proceeding to the next transfer unit's propagation.

Final approach

The final approach is a simplified version of the initial one, without any topology-awareness required and avoiding all the related problems. So, all messages are broadcasted by the PC node network-wide, while each Quax uses unicasts to reply. In addition, before starting a code update session, the PC node executes a trivial discovery phase by issuing an ND command to discover which Quax nodes are actually "alive".

The transfer unit size is set equal to the microcontrollers flash segment size (512 bytes), to ease the flash write operations for the buffering of the image in non-volatile memory. Moreover, several experiments on the broadcast[7] efficiency of the XBee modules took place to define the optimal value for the (fixed) size of each code chunk and the respective transmission schedule for the propagation of a transfer unit, i.e. wait time between the transmission of two chunks (inter-chunk wait), number of consecutive chunk transmissions with inter-chunk wait between them (group size), wait time before transmitting the next group of chunks (inter-group wait). In each test, the number of code chunks forming a transfer unit (512 bytes) was broadcasted with the specified wait times, and the reception rates on the nodes and the transmission latency of a transfer unit were measured. For each configuration of the chunk size and wait times, a series of 20 experiments was conducted on each of the 1-hop and 2-hop topologies that are feasible, given the 3 available Quax nodes and the PC / gateway node (figure 1, GW = PC/ gateway node, R = router node, ED = end-device node). The results are summarized by the following table (table 1), illustrating the mean measurements per configuration.

[7] The term "broadcast" is referring to network-wide broadcasts from this point on.



topology a *topology b* *topology c*

figure 1 topologies for broadcast efficiency experiments

Test configuration					Test results	
chunk size (bytes)	transfer unit chunks count[8]	inter-chunk wait (sec)	group size (# of chunks)	inter-group wait (sec)	mean reception rate (percentage)	mean transmission latency (sec)
64	8	1.4	4	6	99.06%	20.541
64	8	1.2	4	6	99.06%	19.313
64	8	1	4	6	91.88%	16.093
64	8	1.2	4	5	99%	17.324
52	10	1.2	5	5	98.83%	19.718
52	10	1.1	5	5	94.77%	18.894
52	10	1	5	5	85.67%	18.101
52	10	1.2	5	4	90.25%	17.713
44	12	1.2	6	5	69.79%	22.114

table 1 transfer unit experiment results

Although, the results indicate that the configuration <chunk size = 64 B, inter-chunk wait = 1.2 sec, group size = 4, inter-group wait = 5 sec> is the best, this may cause ISOS RAM buffer overflows due to the special byte value escaping (see subchapter 3.6 on XBee module interface with microcontroller). Thus, by selecting the combination <chunk size = 52 bytes, inter-chunk wait = 1.2 sec, group size = 5, inter-group wait = 5 sec>, the overflow risk is very unlikely, while the broadcast efficiency remains high.

Moreover, the synchronization of the PC-Quax node communication prior to starting an update session is omitted, as the parent-child packet caching of the XBee modules (see subchapter 3.6) combined with the online-offline strategy of the Quax node guarantee that the request will reach the node when its offline period ends.

Finally, version 3 also adopts the propagate-repair scheme in point-to-point updates as well, reducing their latency and simplifying the Quax side code implementation. The transmission schedule for each transfer unit in this case is much simpler, with the client waiting only a small amount of time between consecutive chunk unicasts (no groups).

The rest of the functionality is retained from version 2.

[8] The “transfer unit chunks count” configuration variable is dependent, as it is calculated as:

ceiling_function(transfer unit size ÷ chunk size) = ceiling_function(512 ÷ chunk size)

7.2. Update procedure

The update procedure is initiated by the PC client with the point-to-point or one-to-many option defined by the user, to reprogram all the Quax nodes at once or node-by-node respectively. The two sides exchange messages of the “Remote Reprogramming Protocol for Prisma WSN – v13” (thorough description in the respective section of the Appendix) and when the download completes, the currently running application-specific image is overwritten with the new one (figure 2).

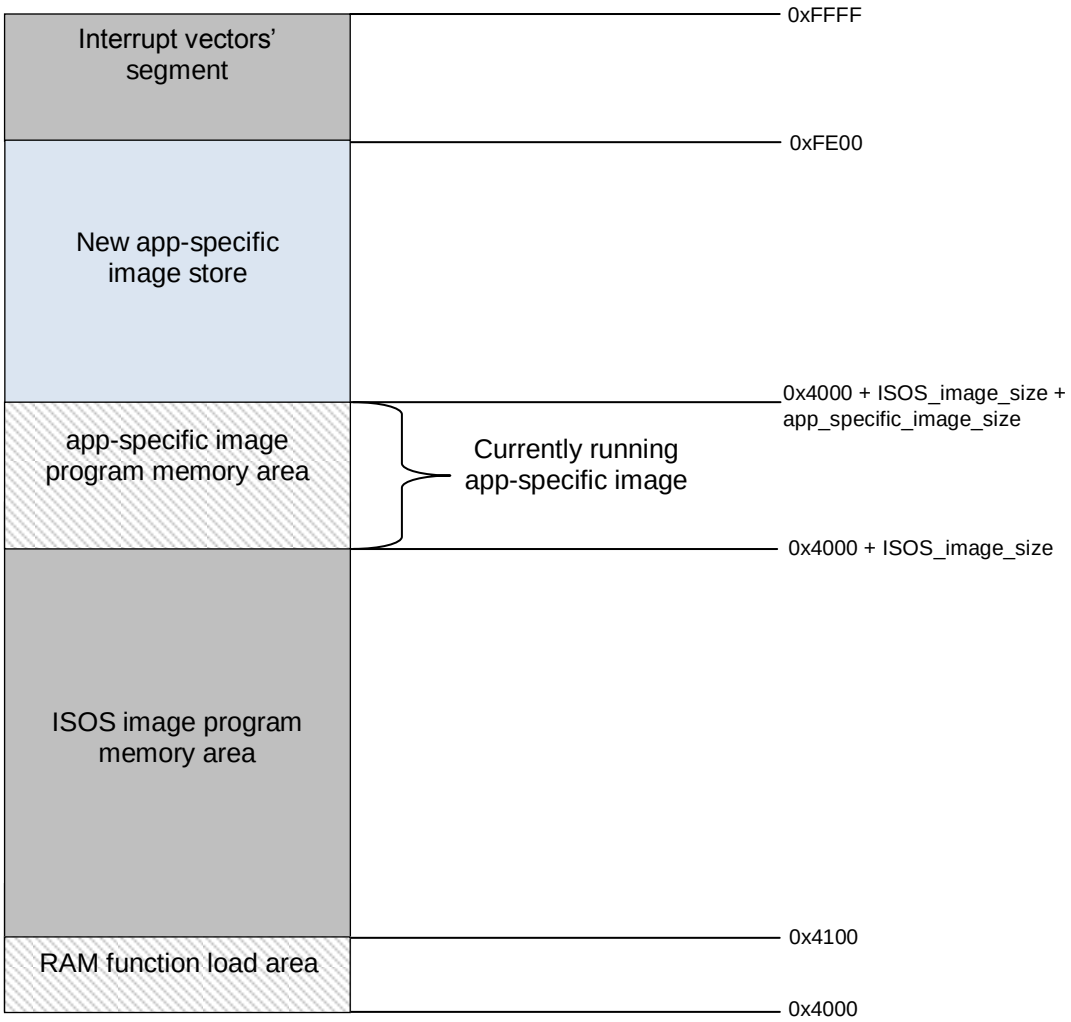


figure 2 version 3, program memory mapping

7.3. One-to-Many update procedure

Discovery phase:

First, the PC client executes a discovery phase, using a configuration file with a list of Quaxes, the ND AT command and user input (if some nodes listed in the configuration file were not discovered), to form the list of Quax nodes that will be updated. This phase is omitted both in the sequence diagrams and pseudocode. Then, the Init phase takes place.

Init phase:

The client broadcasts a “Start Session” request and waits for a certain timeout for unicast replies from all nodes. Each node replies, indicating whether it is ready to proceed, or the image is too big, or the node is already running the new image, or the ISOS code running on the node is not the same as the one required by the new application (not the ISOS version the application was compiled for). If the node has already received the specified image in part, it includes this information in the reply. The PC client excludes all non-ready Quaxes from its list. The procedure is repeated until all nodes in the client’s list reply.

Afterwards, the client determines if it should skip any transfer units and chunks (based on the reception information in all replies) and uploads the new image on the nodes by executing the Propagate and Repair phases for each transfer unit.

Propagate phase (for a transfer unit):

The client broadcasts all the chunks of the current transfer unit, scheduling the broadcasts as discussed (see subchapter 7.1). The Quax nodes do not reply to these broadcasts, but simply add the chunks to their list for the current transfer unit. After transmitting the last chunk of the current transfer unit the client proceeds to the Repair phase.

Repair phase (for a transfer unit):

The client broadcasts a NACK request once and waits a specified amount of time to collect replies. When a node receives a NACK request, if it is missing chunks, it replies to the client with a unicast NACK reply containing information about its missing chunks. If the node is not missing any chunks, it replies with an ACK reply to the first NACK request following any chunk message. If at least one node replied with a NACK reply, the client re-broadcasts all the reported missing chunks following the broadcast schedule, and starts the Repair phase again.

If the client receives no NACK replies within the timeout, but has not received ACK replies from all nodes, it broadcasts an ACK request and waits for replies repeatedly, until all nodes reply with an ACK reply, a NACK reply or a "session aborted" message. A node always replies to an ACK request, either with a NACK reply if it misses some chunks, an ACK reply is it has received all chunks, or a "session aborted" message if it has gone offline. If the client receives one or more NACK replies, it re-broadcasts all the reported missing chunks (scheduling the broadcasts as discussed) and starts the Repair phase again. If the client receives a "session aborted" message, it excludes the sender from its list. When the client receives ACK replies from all nodes in its list, it can start the Propagate phase for the next transfer unit, or execute the Commit phase, if there are no more transfer units to send.

Once a node has received all chunks of the current transfer unit, it saves these data in flash. Nevertheless, the node preserves the chunk reception related info, until receiving a chunk of the next transfer unit or a commit request.

Commit phase:

When the client finishes the Repair phase for the last transfer unit, it broadcasts a Commit request and waits for replies repeatedly, until all nodes reply with a Commit reply. Each node, on receiving a Commit request first checks if the ISOS image version in the request is different than its own. If not, it checks if it has received the entire application-specific image with version number identical to the one in the request. If the last check succeeds, it proceeds as in version 2, regarding the digest checks, the overwrite procedure. Finally, it sends a Commit reply indicating whether there is an ISOS code inconsistency, a part (or the entire) new image is missing, a digest check failed or it successfully committed.

When the client collects Commit replies from all nodes, it excludes any nodes that reported failure from its list. Then, it broadcasts a “Get Version” request and waits for replies repeatedly, until it collects a “Get Version” reply from each node. If the client confirms that all nodes rebooted normally to the new image, the update procedure ends. Otherwise, the entire update procedure is repeated for the nodes that did not

reboot successfully or failed to commit or are missing part of the image, starting from the Init phase.

Remarks:

In order to prevent nodes that have fallen behind in the update procedure from harming the overall latency, a node voluntarily aborts a session:

- silently, if it receives a chunk of a certain transfer unit while it is still missing chunks of a previous transfer unit, or if it receives a NACK request of a transfer unit for which it has not received any chunks.
- by sending a "session aborted" message, if it receives an ACK request of a transfer unit for which it has not received any chunks

Additionally, as in previous versions, a node silently aborts a session if it times out waiting for messages from the client.

The client removes these nodes from the list, along with those who have failed to reply to a “Start Session”, ACK, or Commit request within a specified amount of request efforts. All such nodes are discovered (if reachable) during the final “Get Version” message exchange and updated via a new procedure.

Diagrams

The one-to-many update procedure is illustrated in the following state-transition diagrams, one for each side (figure 3 for the PC client side, figure 4 for the Quax node side).

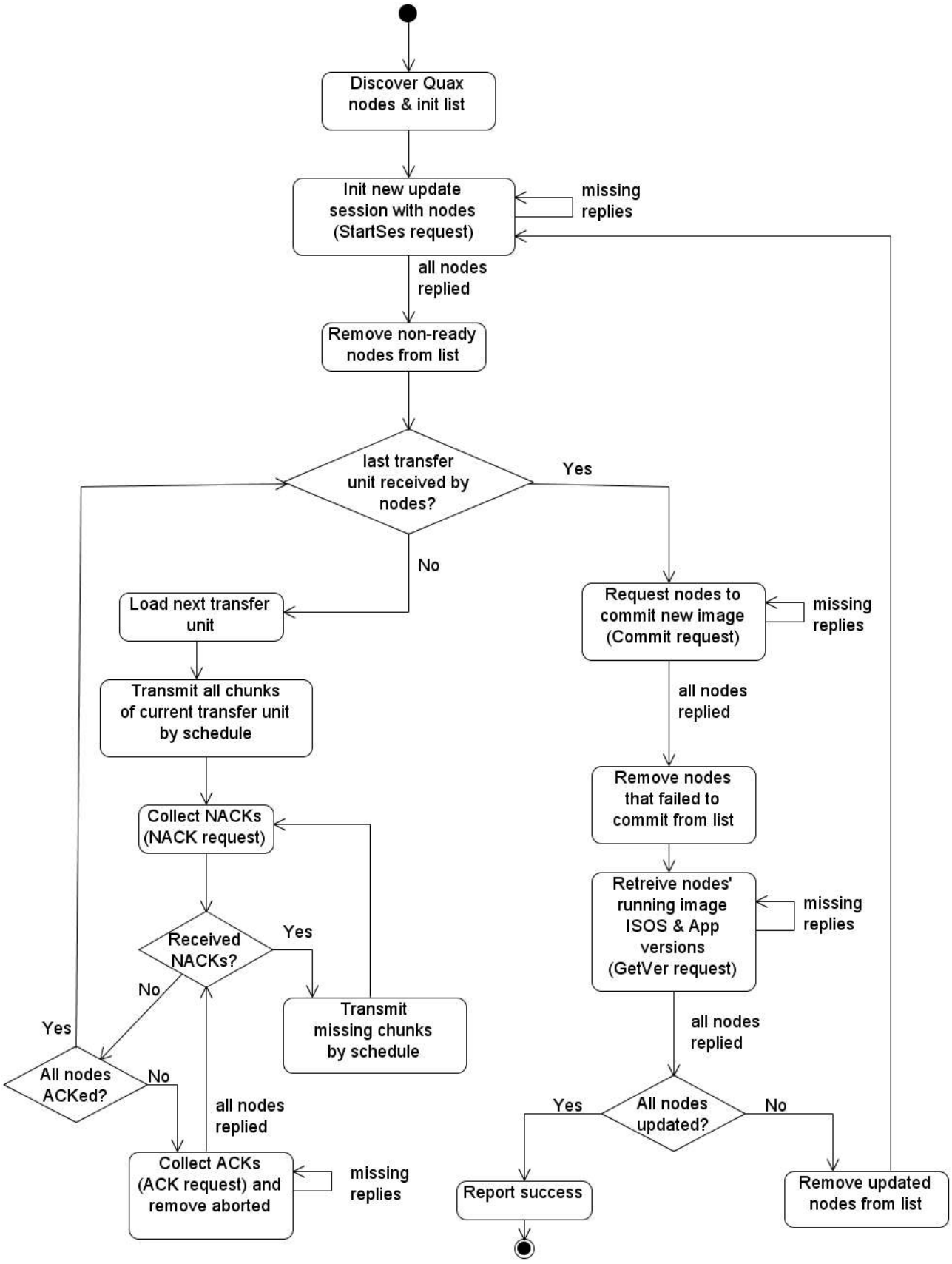


figure 3 version 3 one-to-many, PC client state-transition diagram

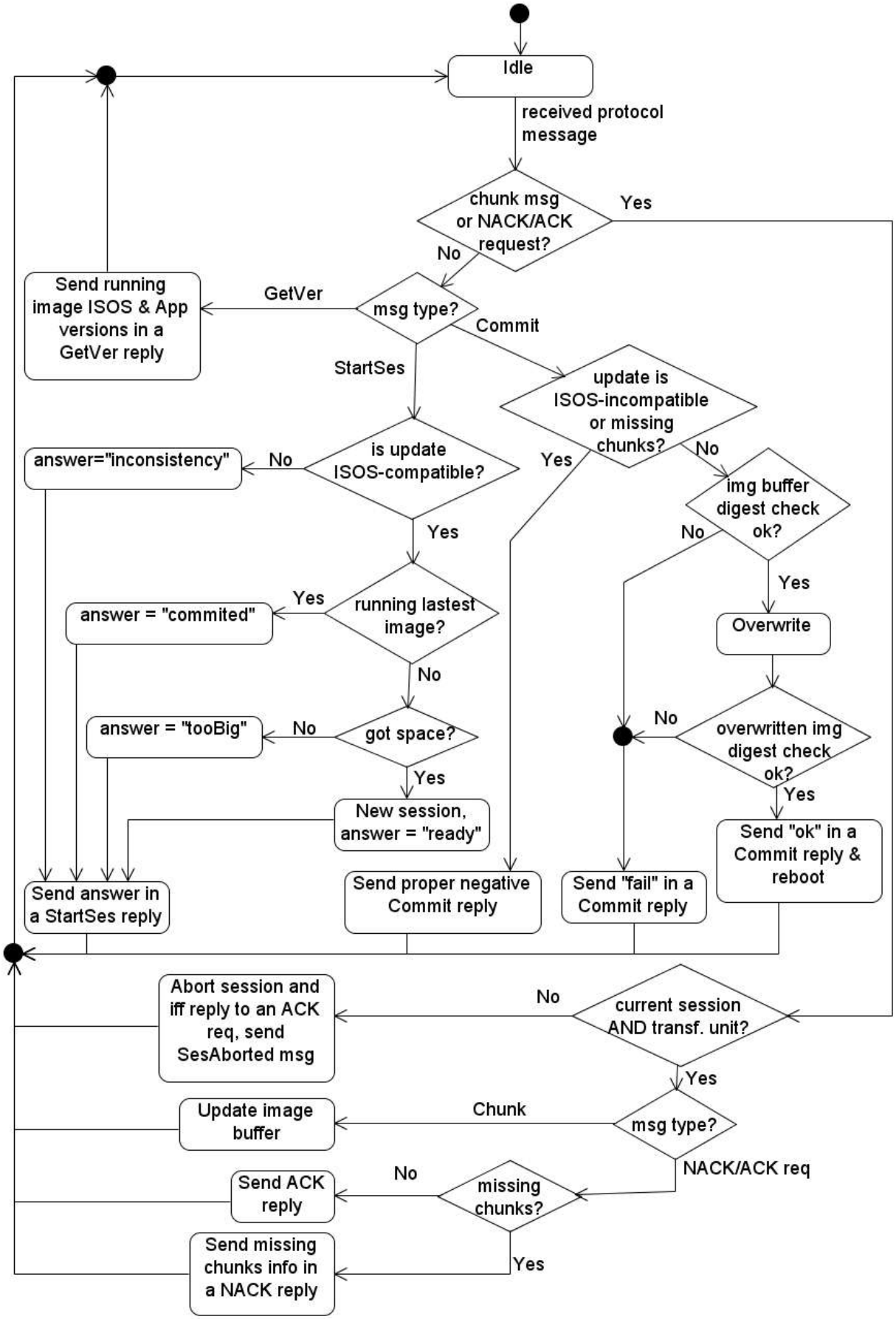


figure 4 version 3, Quax node state-transition diagram

Following, several indicative sequence diagrams, where all messages sent by the PC client are broadcast messages, and all messages sent by the Quax nodes are unicast messages. Additionally, each diagram is followed by a respective, brief description (figures 5-8).

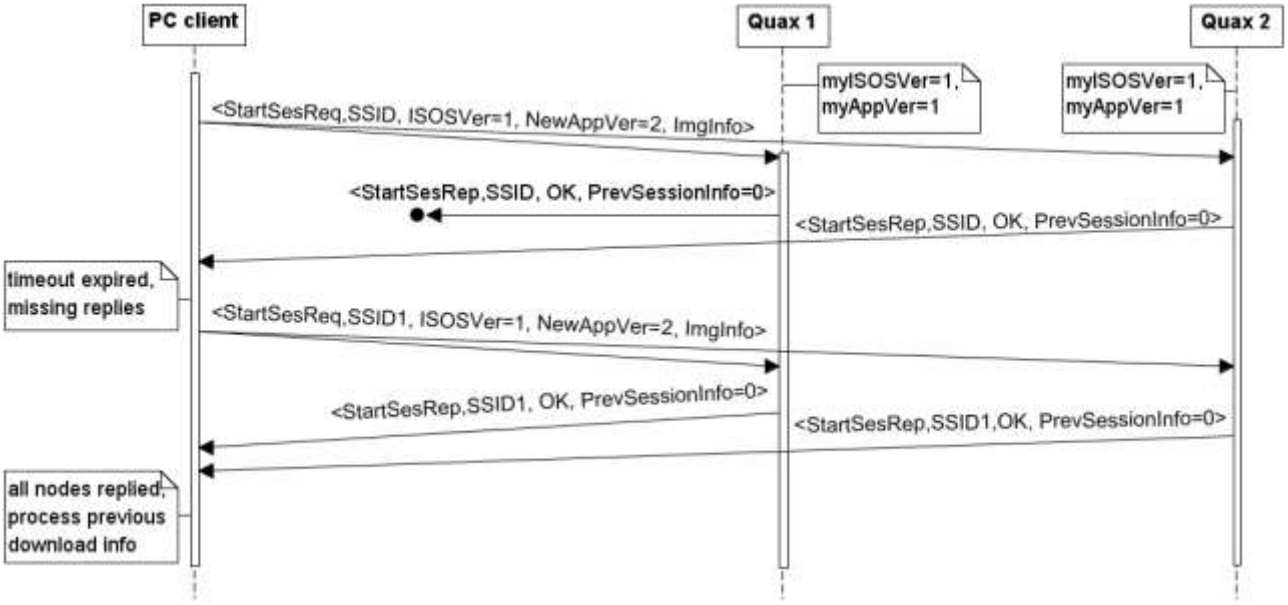


figure 5 version 3, init phase: 1 reply lost

Figure 5 shows the handling of a lost reply during the init phase of the update procedure. Specifically, the client initiates an update session, but fails to receive replies from all nodes within a specified timeout (Quax 1 reply lost). Then, the client initiates another update session, to which all nodes reply successfully as “ready to proceed”.

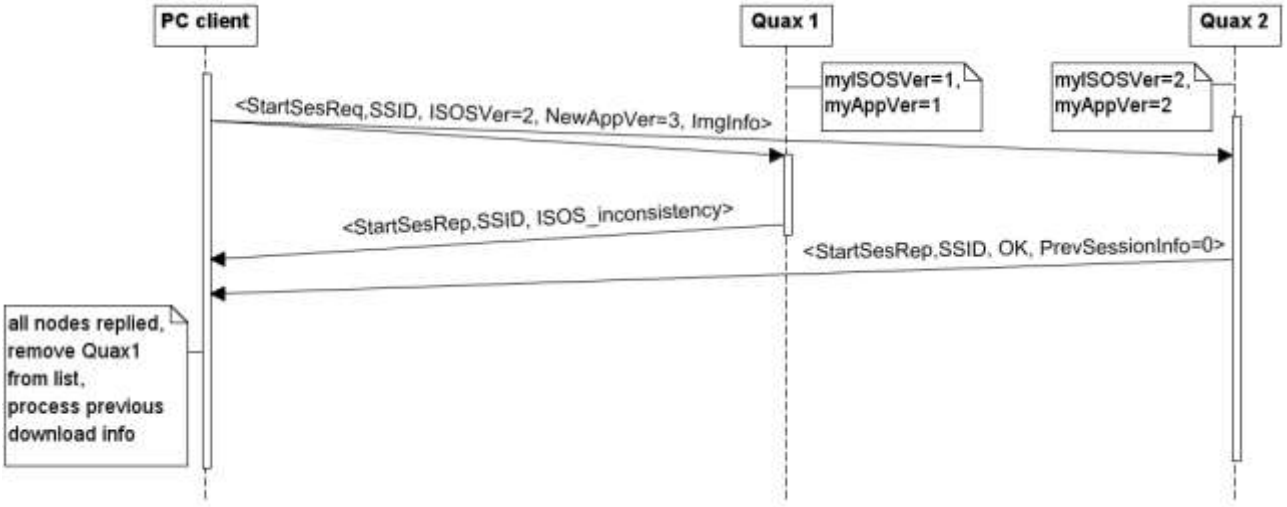


figure 6 version 3, init phase: Quax 1 does not have the required ISOS code

Figure 6 shows the handling of a negative reply during the init phase of the update procedure. Specifically, the client initiates an update session, but Quax 1 does not have the required ISOS code to host the new application-specific image. Thus, Quax 1 replies properly stating the inconsistency, and the client excludes Quax 1 node from the update procedure (Quax list).

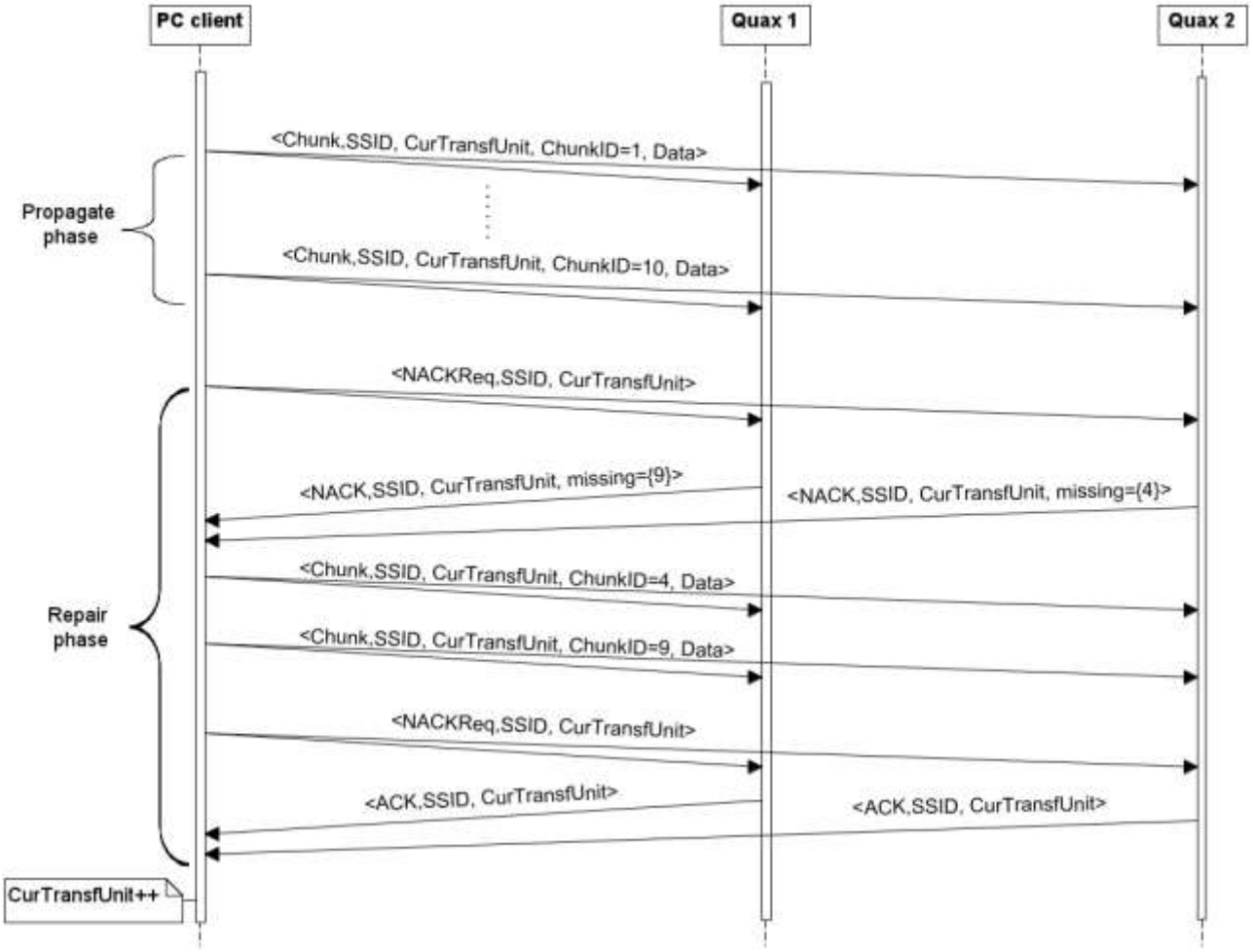


figure 7 version 3, propagate and repair phases (for a transfer unit)

Figure 7 depicts the propagate and repair phases of the update procedure. First, the client transmits all the chunks of the current transfer unit to the nodes (propagate phase). Following, the client requests from the nodes to report any chunks that they failed to receive, and each node replies with its missing chunks (NACK). The client re-transmits all the missing chunks and requests receipt status from the nodes anew. Both nodes reply with a positive acknowledgement (ACK), since they have now received all the chunks of the current transfer unit, and the client can proceed to the next transfer unit.

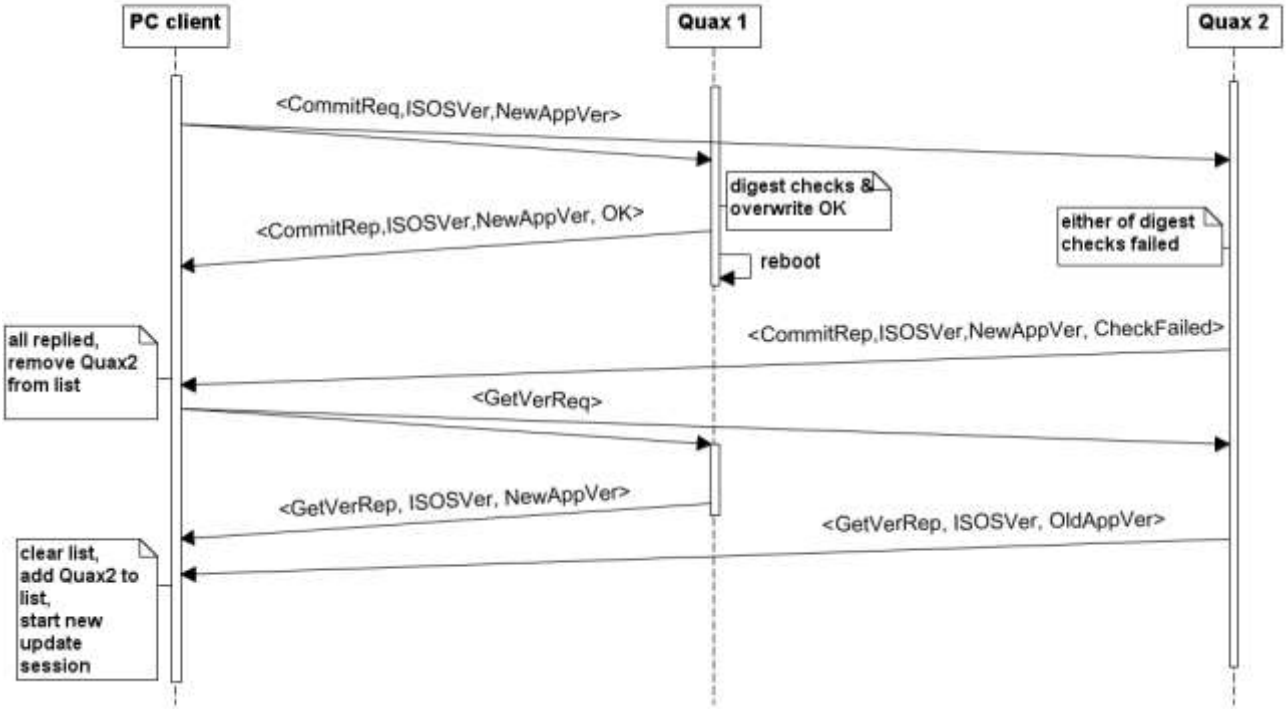


figure 8 version 3, commit phase: digest check failed on Quax 2

The commit phase of the update procedure is illustrated in figure 8. The client requests from the nodes to commit to the new application-specific image (already downloaded), and the nodes proceed as in version 2. Specifically, Quax 1 performs the overwrite and digest checks successfully, replies positively and reboots to the new image, while Quax 2 encounters a digest check failure and reports negatively to the client's request. Then, the client receives the commit replies, retrieves the image versions of the nodes and detects the failure on Quax 2, which will be fixed via a new update session.

7.4. Point-to-point update procedure

The point-to-point update procedure is quite similar to a trivial one-to-many update procedure with only one Quax in the client's list. As a consequence, the sequence diagrams for this update mode are ommitted. Also, the Discovery phase is spared and the two sides exchange unicasts messages with smaller timeouts on the client side. Additionally, the client skips the NACK request and always sends an ACK request in the Repair phase.

Init phase:

The client unicasts a "Start Session" request and waits for a certain timeout, repeatedly until the node replies. The node replies just as in the one-to-many update procedure and the client determines if the update is not applicable on the specific node or if it should skip any transfer units and chunks (based on the reception information in the reply).

Afterwards, the client uploads the new image on the node by executing the Propagate and Repair phases for each transfer unit.

Propagate phase (for a transfer unit):

The client broadcasts all the chunks of the current transfer unit, scheduling the transmissions as discussed for the point-to-point update procedure (see subchapter 7.1). The node behaves just as in the one-to-many update procedure. After transmitting the last chunk of the current transfer unit the client proceeds to the Repair phase.

Repair phase (for a transfer unit):

The client unicasts an ACK request and waits a specified amount of time, repeatedly until the node replies. The node replies either with a NACK reply if it misses some chunks, an ACK reply if it has received all chunks, or a "session aborted" message if it has gone offline. If the node is missing any chunks, the client re-transmits them. The Repair phase is repeated until the node has all the chunks of the current transfer unit.

Again, once the node has received all chunks for the current transfer unit, it saves these data in flash, but preserves the chunk reception related info, until receiving a chunk of the next transfer unit or a commit request.

Commit phase:

When the client finishes the Repair phase for the last transfer unit, it unicasts a Commit request and waits a specified amount of time, repeatedly until the node replies. The node, on receiving a Commit request behaves just as in a one-to-many update procedure. If the node fails to commit, the client repeats the update procedure from the Init phase. Else, the client initiates a "Get Version" message exchange to verify the success of the reboot.

Remarks:

As in previous versions, in case of communication problems or session abortion by the node, the client starts a new update session by repeating the procedure from the Init phase.

Diagrams

The point-to-point update procedure, regarding the client side, is illustrated in the following state-transition diagram (figure 9). Since the Quax side behaves just as in the one-to-many update procedure, the respective state-transition diagram is omitted.

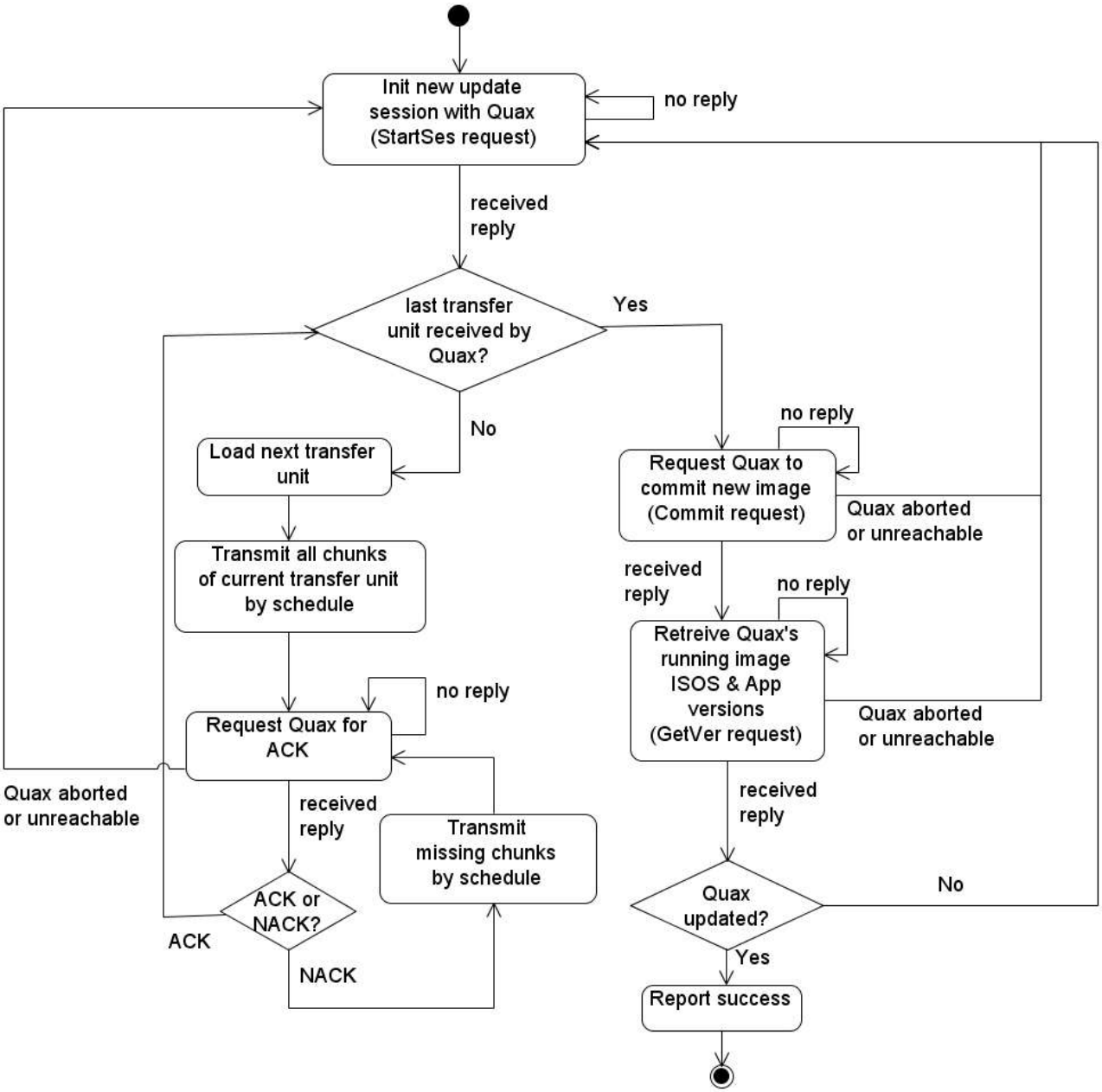


figure 9 version 3 point-to-point, PC client stat-transition diagram

7.5. Implementation

PC client code modifications

The PC client implements the procedure regarding one-to-many and point-to-point updates as described in the previous subchapters. Additionally, the user can choose between the one-to-many and the point-to-point modes, also selecting the Quax node to update in the latter case.

Quax code modifications

The remote reprogramming component of ISOS implements the procedure regarding the one-to-many and point-to-point updates as described in the previous subchapters. These modifications also simplify the buffering of the downloaded image, regarding both the software buffers in RAM and the new image buffer in non-volatile memory.

Several changes in ISOS code were made, in order for the ISOS image to be the same both for MS and DT Quax nodes. This way, any update's applicability is independent of the node type. The node simply selects its type (MS or DT) at runtime, based on a value stored in non-volatile memory. The initialization of each Quax node with this value (node type) needs to be done by hand, along with the current image related info initialization and the initial image burn.

Finally, since the synchronization of the PC-Quax node communication is no longer necessary, no synch messages are sent by the Quax nodes. Nevertheless, the rest of the online-offline strategy, regarding the online and offline periods, remains the same as in version 2.

Pseudocode

A high-level pseudocode for each side is presented below (more detailed pseudocodes, also handling periods/timeouts on the node side, as well as handling lists and endless waiting on the client side, in the Appendix on "Remote Reprogramming Protocol for Prisma WSN – v13").

PC client pseudocode (one-to-many update):

```
int uploadCode(QuaxList) {
   state = NEWSES;
   while(1) {
      switch( state ) {
          case NEWSES: {
              do {
                  broadcast NEWSES request;
              } while( ! received NEWSES reply from every Quax );
              updateList(QuaxList);
              if ( all nodes run the latest inage ) { return success; }
              else if ( all nodes have the entire image ) state = COMMIT;
              else {
                  form transmission list;
                  state = NXTCHUNK;
              }
          }
          case NXTCHUNK: {
              state = COMMIT;
              foreach chunk in the transmission list {
                  broadcast chunk in a NXTCHUNK request;
                  if( new group of chunks ) wait( INTERGROUP_TIMEOUT);
                  else wait( INTERCHUNK_TIMEOUT);
              }
              state = COLLECT_NACKs;
          }
          case COLLECT_NACKs: {
              broadcast NACK request;
              collect all replies within the timeout;
              updateList(QuaxList);
              state = REPAIR;
          }
          case COLLECT_ACKs: {
              do {
                  broadcast ACK request;
              } while( ! received proper reply from every Quax );
              updateList(QuaxList);
              state = REPAIR;
          }
          case REPAIR: {
              if (no missing chunks) {
                  if (! all nodes ACKed) state = COLLECT_ACKs;
                  else if (no more transfer units) state = COMMIT;
                  else {
                      form transmission list for next transfer unit;
                      state = NXTCHUNK;
                  }
              }
```

```
            else {
                form transmission list with missing chunks;
                state = NXTCHUNK;
            }
        }
        case COMMIT: {
            do {
                broadcast COMMIT request;
            } while( ! received COMMIT reply from every Quax );
            updateList(QuaxList);
            state = GETVER;
        }
        case GETVER:
            do {
                broadcast GETVER request;
            } while( ! received GETVER reply from every Quax );
            updateList(QuaxList);
            if ( all nodes run the latest image ) { return success; }
            state = NEWSES;
    }
  }
}
```

PC client pseudocode (point-to-point update)

```
int uploadCode(Quax) {
   state = NEWSES;
   while(1) {
      switch( state ) {
          case NEWSES: {
              do {
                  send NEWSES request to Quax;
              } while( ! received NEWSES reply from Quax );
              if ( answer == TOO_BIG ) { return failure; }
              if ( answer == COMMITED ) { return success; }
              if ( Quax has the entire image ) state = COMMIT;
              else {
                  form transmission list;
                  state = NXTCHUNK;
              }
          }
          case NXTCHUNK: {
              state = COMMIT;
              foreach chunk in the transmission list {
                  send chunk in a NXTCHUNK request to Quax;
                  wait( SMALL_INTERCHUNK_TIMEOUT);
              }
              state = COLLECT_ACK;
          }
          case COLLECT_ACK: {
              do {
                  send ACK request to Quax;
              } while( ! received proper reply from Quax );
              state = REPAIR;
          }
          case REPAIR: {
              if ( Quax aborted ) state = NEW_SES;
              else if ( Quax ACKed ){
                  if (no more transfer units) state = COMMIT;
                  else {
                      form transmission list for next transfer unit;
                      state = NXTCHUNK;
                  }
              }
              else
              {
                  form transmission list with missing chunks;
                  state = NXTCHUNK;
              }
          }
          case COMMIT: {
              do {
```

```
                broadcast COMMIT request to Quax;
            } while( ! received COMMIT reply from Quax );
            if ( Quax aborted ) state = NEW_SES;
            else if ( check failed on Quax) state = NEW_SES;
            else state = GETVER;
        }
        case GETVER:{
            do {
                send GETVER request to Quax;
            }while ( ! received GETVER reply );
            if ( Quax runs the latest image ) { return success; }
            state = NEWSES;
} } }   }
```

Quax pseudocode (both update modes):

```
receive_handler( msg ) {
   switch( msg ) {
       case GETVER request: {
           send GETVER request;
       }
       case NEWSES request: {
           if( not enough space )
               answer = TOO_BIG;
           else if ( already running this image)
               answer = COMMITED;
           else {
               if ( this is a new update )
               {
                   initialize download data;
               }
               start new session;
               answer = <READY, last_chunk_recvd_id>;
           }
           send answer in a NEWSES reply;
       }
       case NXTCHUNK request: {
           if( request belongs to unknown session
               || missing chunks from previous transfer unit)
               abort session;
           else{
               save chunk;
           }
       }
       case NACK request: {
           if( request belongs to unknown session
               || missing chunks from previous transfer unit)
               abort session;
           else if(missing chunks of current transfer unit)
               send reception info in a NACK reply;
           else
               send ACK reply;
       }
       case ACK request: {
           if( request belongs to unknown session
               || missing chunks from previous transfer unit){
               send ABORTED reply;
               abort session;
           }
           else if(missing chunks of current transfer unit)
               send reception info in a NACK reply;
           else
               send ACK reply;
```

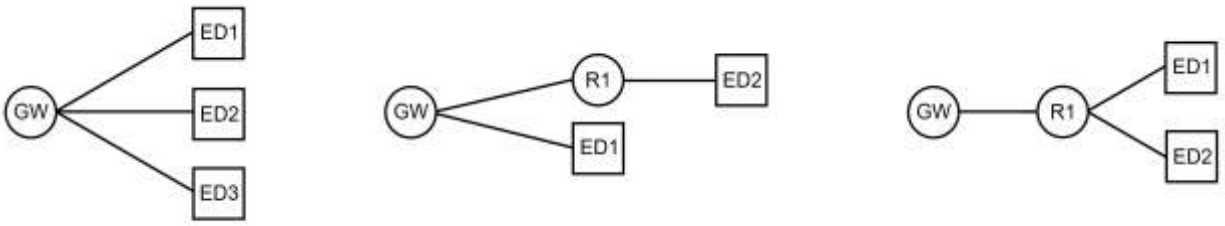
```
        }
        case COMMIT request: {
            if(ISOS image version in request != my ISOS image version)
                send ISOS_IMAGE_INCONSISTENCY in a COMMIT reply;
            else if(missing part of the image)
                send MISSING_IMAGE in a COMMIT reply;
            else if ( digest check fails) {
                erase download data;
                send CHECK_FAILED in a COMMIT reply;
            }
            else {
                overwrite old image with new;
                if(overwrite digest check fails){
                    send CHECK_FAILED in a COMMIT reply;
                    halt application;
                }
                else {
                    send OK in a COMMIT reply;
                    reboot;
                }
            }
        }
        case ABORT request: {
            if( request belongs to current session)
                send ABORTED reply;
} }   }
```

8. Evaluation

The functionality implemented in each version was validated by a series of test and scenarios. These tests were conducted using the Quax nodes available, i.e. 1 Quax Gateway node and 3 Quax nodes. More details on the validation tests and their results in the respective section of the Appendix, on Validation tests.

Furthermore, another series of tests took place to measure the latency of the update procedure for each implemented version. This allowed the comparison and evaluation of the gradual refinement and improvement achieved by each version, although, since versions 2 and 3 update only a part of the image, there is no point in comparing their latencies with that of version 1. However, the update procedure of version 1 takes about 4.5 to 5 minutes to update the entire image on a Quax node, depending on the new image size (20KB – 24KB).

Regarding the evaluation of the update procedure for version 2, version 3 in one-to-many mode, and version 3 in point-to-point mode, their image-download latency[9] and entire-procedure latency[10] were measured through a series of 30 tests on each of the topologies below (figure 1), where GW is the PC / gateway node (not reprogrammed), R is a ZigBee router node, and ED is a ZigBee end-device node. Version 2 and version 3 point-to-point update procedures reprogram the network node-by-node, whereas version 3 one-to-many update procedure updates all the nodes at once. The image update transferred over the air was of fixed size, equal to 1536 bytes. Moreover, the ISOS code of version 2 was adjusted to be identical to that of version 3, regarding the radio and serial interface drivers, in order for any comparisons with version 3 to be fair. This way, each code chunk consists of 52 bytes and the image update consists of 30 code chunks in both versions. Additionally, in version 3, this means 3 transfer units (of 10 chunks each) and 3 rounds of propagate-repair phases respectively.



topology a *topology b* *topology c*

figure 1 topologies for evaluation tests

The image-download latencies are summarized by the tables below (1, 2, 3), where the latencies are of the format {minutes:}seconds.milliseconds, the total latency is the time it took to reprogram the entire WSN and the per Quax latency is calculated by dividing the total latency by the number of Quax nodes, which is 3.

	(mean) total latency	(mean) per Quax latency
topology a	30.180	10.060
topology b	29.575	9.858
topology c	30.601	10.200
average (a,b,c)	30.119	10.040

table 1 version 2 (point-to-point) image-download latency

[9] Session-committed latency measured from session establishment (image transfer start) until the node(s) commit to the new image (commit reply/replies).

[10] Overall latency includes session-committed latency, as well as syncronization of communications and confirmation of reboot for version 2, init phase and confirmation of reboot for version 3 point-to-point, and discovery and init phases and confirmation of reboot for verion 3 one-to-many.

	(mean) total latency	(mean) per Quax latency
topology a	22.569	7.523
topology b	21.825	7.275
topology c	21.928	7.309
average (a,b,c)	22.107	7.369

table 2 version 3 (point-to-point) image-download latency

	(mean) total latency	min total latency	max total latency
topology a	1:09.497	1:06.121	1:34.403
topology b	1:10.944	1:06.382	1:33.650
topology c	1:11.147	1:06.201	1:38.444
average (a,b,c)	1:10.529	-	-

table 3 version 3 (one-to-many) image-download latency

min latency obtained (for one Quax)	ideal latency (lower bound)
6.640	≈ 6 sec[11]

table 4 version 3 (point-to-point) minimum and lower bound for image-download latency

min latency obtained (for one Quax)	ideal latency (lower bound)
1:06.121	≈ 1 min[12]

table 5 version 3 (one-to-many) minimum and lower bound for image-download latency

First of all, tables 4 and 5 show that the actual image-download latencies in version 3 are quite close to the ideal ones (calculated according to the transmission schedules[11,12]). The deviation between the actual and the ideal latencies are reasonable, since the delay of the repair and commit phases is variable. Even when no chunk packets were lost, the client must collect ACK(s) before moving on to the next transfer unit.

Regarding the point-to-point update procedure, the improvement obtained by replacing the stop-and-wait propagation scheme of version 2 with the NACK based of version 3 is obvious by comparing the mean total and per Quax latencies of tables 1 and 2 for each topology. In version 2, the client has to wait for explicit acknowledgements for each code chunk before sending the next one. So, each chunk costs an entire round-trip time. In contrast, in the update procedure of version 3 point-to-point, the client can propagate all the chunks of a transfer unit with a reasonable, but small inter-chunk wait time and ask for their acknowledgement in the repair phase. Thus, the propagation of each transfer unit (512 bytes) and the entire update is significantly accelerated.

As discussed in previous chapters, the one-to-many update procedure of version 3 follows a transmission schedule which obliges the client to wait for specified timeouts (of at least 1.2 seconds) between the broadcast transmissions of each message. In contrast, in the stop-and-wait, point-to-point update procedure of version 2, the client can send the next message as soon as it receives a reply, while the point-to-point update procedure of version 3 further expedites the propagation scheme, as explained in the previous paragraph. Thus, for a WSN of 3 nodes, both the point-to-point image downloads (version 2 and 3) are quite faster than the one-to-many image download of version 3 (total image-download latency columns of

[11] According to unicast schedule: (30 chunks)*(0,2 sec inter-chunk wait) + Repair_cost + 1 RRT (commit) = 6 sec + Repair_cost + 1 RRT (commit) ≈ 6 sec

[12] According to broadcast schedule: (3 transfer units)*(19,7 sec per t.u.) + Repair_cost + Commit_cost = 59,1 + Repair_cost + Commit_cost ≈ 1 min

the three tables).

However, the one-to-many update procedure of version 3 is designed to be rather independent of the network's size, so its total image-download latency should not be greatly harmed by the number of nodes in the WSN. Thus, by comparing the average per Quax image-download latencies of tables 1 and 2 with the average image-download latency of table 3 (last rows of the three tables), one can deduce the network sizes for which version 3 one-to-many update procedure outperforms the other two. Specifically, the one-to-many update procedure of version 3 is expected to be favored over the point-to-point one of version 2 for networks of eight or more nodes (1:10.529÷10.040 ≈ 7.025), and over the point-to-point one of version 3 for networks of ten or more nodes (1:38.128÷13.985 ≈ 9,571).

	(mean) total latency	(mean) per Quax latency
topology a	0:53.037	17.679
topology b	1:05.877	21.959
topology c	1:10.388	23.463
average (a,b,c)	1:03.101	21.034

table 6 version 2 (point-to-point) entire-procedure latency

	(mean) total latency	(mean) per Quax latency
topology a	39.201	13.067
topology b	43.036	14.345
topology c	43.628	14.543
average (a,b,c)	41.955	13.985

table 7 version 3 (point-to-point) entire-procedure latency

	(mean) total latency	min total latency	max total latency
topology a	1:37.787	1:19.165	2:09.056
topology b	1:38.121	1:22.447	2:13.105
topology c	1:38.476	1:22.017	2:10.175
average (a,b,c)	1:38.128	-	-

table 8 version 3 (one-to-many) entire-procedure latency

Additionally, a comparison of the one-to-many and point-to-point procedures based on the entire-procedure latency (tables 6-8) is also important, showing another interesting advantage of the one-to-many procedure. For the point-to-point procedures, the entire-procedure latency is the image-download latency plus the latencies of session establishment (including the communication synchronization and version retrieval for version 2) and reboot confirmation, which increase the total latency by a factor dependent on the number of WSN nodes. For the one-to-many procedure, the entire-procedure latency is the image-download latency plus the latencies of the Discovery and Init phases, as well as the reboot confirmation latency, which are generally independent of the number of WSN nodes present. Thus, based on the entire-procedure latencies, the one-to-many update procedure of version 3 is expected to be favored over the point-to-point one of version 2 for networks of five or more nodes (1:38.128÷21.034 ≈ 4.67), and over the point-to-point one of version 3 for networks of eight or more nodes (1:38.128÷13.985 ≈ 7.02).

Finally, the maximum image-download and entire-procedure latencies of the one-to-many update procedure (last columns of tables 3 and 8) show that, although the one-to-many update procedure may occasionally take quite more than the minimum time to complete, it is guaranteed that all reachable nodes will eventually be reprogrammed with a tolerable latency.

9. Conclusion and future work

This thesis described the challenging task of reprogramming wireless sensor networks and focused on the implementation of such a utility on a specific platform. The solution gradually refined and enhanced the functionality implemented in three separate versions, from node-by-node updating the entire code of all sensor nodes, to updating just the application-specific part of the code, and finally to further expediting the update procedure and enabling the reprogramming of all WSN nodes at once. Moreover, several series of tests were performed to validate each version and to define the network size for which the one-to-many and all-at-once update procedure should be preferred over the point-to-point and node-by-node one.

Several enhancements could greatly augment the current solution's functionality. To begin with, supporting the reprogramming of only a subset of all the nodes in the network, based on specific selection criteria, seems an interesting aspect to explore and is often useful in real-world deployments. The security of the reprogramming procedure is another important issue that should be addressed, as an intruder could reprogram the WSN for his own purposes. The procedure could also become more automated, requiring less user intervention, as well as incorporate a systematic way to adapt to network size changes (e.g. periodic advertising). Furthermore, the reliability of the current solution could be further enhanced by checking the updatable program memory part for integrity on each boot, and acting properly. Additionally, the existence of multiple application-specific images (on nodes that have external memory) and the ability to switch between them are quite useful in re-tasking and rolling back, in case of application failures.

Nonetheless, some other new features could be easily added to the PC client program without any radical changes in the entire-procedure reprogramming procedure. For example, it can be configured to automatically select between the one-to-many or point-to-point update modes, based on the number of nodes present. Moreover, the PC client program can be integrated or configured to forward the application-level messages received during the update procedure, thus exploiting the non-intrusive nature of the designed solution. Finally, modifying the client program so that it can be easily handled by users less familiar with the platform, and adding a graphic user interface would considerably increase its user-friendliness and acceptance.

Appendix

Remote Reprogramming Protocol for Prisma WSN v10 (implemented by solution v1)

Introduction

This is a description of the protocol (to be) used to remotely program the nodes of the WSN.

The assumption is that remote reprogramming is initiated using a program that resides on a PC to which the WSN gateway is connected. The WSN gateway is assumed to act as a dummy router (not to be re-programmed). Each node must be addressed explicitly, following the usual addressing and packet transmission/reception conventions.

The protocol allows the client to: (i) retrieve a node's current software version, (ii) start a code upload session, (iii) upload the code image via a stop-and-wait protocol; and (iv) commit the uploaded image and reboot the node.

The protocol is designed to support (but not to enforce) the continuation of a previously aborted upload, without requiring the client to re-send the entire code image from scratch.

The next sections describe the messages for the corresponding protocol exchanges, as well as the encapsulation format that application-level messages need to comply with.

Application-level messages

Client to node & node to client:

```
AppMsg = AppMsgTag
         Payload

AppMsgTag = x01
Payload = byte[]
```

Comments:

Application-level messages, exchanged between the client and the node, need to comply with this format in order to be properly handled/forwarded (on both sides) and avoid interference with the “Remote reprogramming protocol” (and perhaps other system-level protocol) messages.

The payload is application-specific and its size is bound by the maximum radio packet payload minus one (for the *AppMsgTag*).

Synchronize client – node communication

Client to node:

N/A

Node to client:

```
SynchMsg = SynchMsgTag
SynchMsgTag = x02
```

Comments:

The node keeps its radio turned off for most of the time, unless the application code running on it has something to send to the client. However, occasionally, the client may also wish to initiate communication with the node, but this may fail due to the node having its radio turned off at that time.

To address this problem, a “polling” timeout is set on the node, which counts down until the next time the node should poll the client for messages. The node’s radio remains in sleep enabled mode until the timeout expires. When this happens, the node turns its radio on, immediately sends to the client a synchronization message and keeps its radio on for some time in order to receive messages from the client (if any). The node extends the time during which the radio is on each time it receives a message from the client. This makes it possible to have longer sessions without the node entering a sleep mode in between. If the node does not receive a message within its online interval, it reverts back to its usual sleeping pattern by resetting the “polling” timeout (and aborts the current session, if this is still pending).

The same procedure is followed every time application code turns the node’s radio on and sends an application-level message to the client. In this case, no explicit synchronization message is required, since the application message can serve the same purpose (inform the client about the node’s radio activity).

The client must wait for a synchronization (or application) message before attempting to send the first message (of a possibly longer session) to the node. Else, messages sent to the node will most probably be lost. Once a session is completed (or aborted), the client must wait for a fresh synchronization message in order to initiate a new session.

Get version of currently running code

Client to node:

```
GetVersionReq = GetVersionReqTag

GetVersionReqTag = x03
```

Node to client:

```
GetVersionReply = GetVersionReplyTag
                  VersionNumber

GetVersionReplyTag = x04
VersionNumber = uint32_t
```

Comments:

Retrieve the version number of the code (image) running on a node. The node replies with the version number of the currently running image.

Start new code upload session

Client to node:

```
StartSessionReq = StartSessionReqTag
                  SessionNumber
                  ImageVersionNumber
                  ImageSize
                  DigestCode

StartSessionReqTag = x05
SessionNumber = uint32_t
ImageVersionNumber = uint32_t
ImageSize = uint32_t
DigestCode = uint32_t
```

Node to client:

```
StartSessionReply = StartSessionReplyTag
                    SessionNumber
                    (READY | TOO_BIG | COMMITTED)
                    [LastReceivedChunkNumber]

StartSessionReplyTag = x06
SessionNumber = uint32_t
READY = x01
TOO_BIG = x02
COMMITTED = x03
LastReceivedChunkNumber = uint16_t
```

Comments:

Start a new code upload session. The session number, assigned by the client, identifies the session. A new session always overrides an old one (which might still be in progress). The node replies, using the session number of the request, indicating whether it is ready to proceed, or the image is too big, or the image is already committed (the node is already running this code).

The digest of the code image is sent by the client at the start of the session. The client must store this value and compare it to the digest computed locally as the code image is received. In order for a session to be committed, the digest computed by the node must be identical to the digest sent by the client.

If the node is ready to commence the code upload, it sends the number of the last code chunk already received (this is to support a continuation of a previous upload). More specifically, if the image has not been uploaded, in part, the chunk number specified by the node will be 0, else it will be greater than 0. If the requested chunk number is equal to the number of the last chunk, the node has already received the entire code image (and the client may proceed to commit the session).

Upload next chunk of code

Client to node:

```
NextChunkReq = NextChunkReqTag
               SessionNumber
               ChunkNumber
               ChunkLength
               Data

NextChunkReqTag = x07
SessionNumber = uint32_t
ChunkNumber = uint16_t
ChunkLength = uint8_t
Data = byte[ChunkLength]
```

Node to client:

```
NextChunkReply = NextChunkReplyTag
                 SessionNumber
                 ChunkNumber

NextChunkReplyTag = x08
SessionNumber = uint32_t
ChunkNumber = uint16_t
```

Comments:

The code upload is performed using a stop-and-wait protocol. The client sends the "next" chunk (requested by the node when it confirms the start of a new session), and the node replies to confirm the reception (and storage) of the chunk. The reply serves as a prompt for the next chunk, i.e., if the chunk number of the reply is *k* then the client is requested to send the $k+1^{th}$ chunk.

The size of code chunks is equal to maximum chunk size (naturally, this will be equal to the maximum payload size minus the protocol header size), except the last chunk which could be smaller. The last chunk is **explicitly** identified via the most significant bit of the chunk number: the last chunk has this bit set to 1 and all other chunks have this bit set to 0.

The code image consists of two disjoint parts (main part and interrupt vector) which need to be mapped on different areas of the program memory. The 2^{nd} most significant bit of the chunk number is used to differentiate between these two parts. Specifically, this bit is set to 1 if the chunk contains part of the interrupt vector, 0 otherwise.

Identification of the interrupt vector and last chunk, via the 1^{st} and 2^{nd} most significant bits, applies only to the chunk numbers sent from the client to the node (the chunk numbers sent from the node to the client have these bits set to 0).

The node may save the code chunks it receives in non-volatile memory, so they can survive a reboot. Else, if the node reboots in the midst of a code upload session, the next session will start sending the image from scratch.

Note: The client may receive and must be able to handle a next chunk reply message with any chunk number (not necessarily the number of the most recently sent chunk). Also, the node may reply with a session aborted message (see respective section).

Commit code upload session

Client to node:

```
CommitReq = CommitReqTag
            SessionNumber

CommitReqTag = x09
SessionNumber = uint32_t
```

Node to client:

```
CommitReply = CommitReplyTag
              SessionNumber
              (OK | CHECK_FAILED)

CommitReplyTag = x0A
SessionNumber = uint32_t
OK = x01
CHECK_FAILED = x02
```

Comments:

Once the image is successfully uploaded on the node, the client can request the node to commit and start running it. The node verifies that the digest of the image is correct (the client sends the digest when the code upload session is started). If this is not the case, it sends a negative reply. Else, it sends a positive reply, remaps the image to the proper memory location (as required by the bootloader) and reboots to start execution using the new code.

Note: The node may also reply with a session aborted message (see respective section).

Abort code upload session

Client to node:

```
AbortSessionMsg = AbortSessionMsgTag
                  SessionNumber

AbortSessionMsgTag = x0B
SessionNumber = uint32_t
```

Node to client:

```
SessionAbortedMsg = SessionAbortedMsgTag
                    SessionNumber

SessionAbortedMsgTag = x0C
SessionNumber = uint32_t
```

Comments:

If the client decides to abort an upload session (this may happen for various reasons), it informs the node by sending an abort message. The node confirms this via a reply.

Also, the node may **unilaterally** decide to abort an ongoing session (e.g., using a timeout), to handle the case where the client does not “properly” complete, commit or abort a session (e.g., due to a failure, slow processing or networking problems). This decision is **silent**, i.e., the node does not attempt to notify the client about it. A unilateral abort may also occur due to a node reboot.

It is possible (e.g., due to using a small timeout) for the node to abort a session without the client actually having experienced a failure. Thus the node may receive a next chunk or commit request from the client for an aborted session. In this case, it replies with a session aborted message.

Note that when a session is aborted (explicitly by the client or silently by the node) the node may decide to discard the (partly) uploaded image in order to free memory. In this case, upload must start from scratch (using another session). Of course, the node is free to keep partly uploaded code images in memory so that upload can be resumed (via another session).

PC client pseudocode

```
int uploadCode(int nid) {
   setTimeout(A_VERY_GENEROUS_VALUE);
   while ( !recv(nid, <SYNCH>) && !recv(nid, <APPMSG, msg>) ) {}
   if ( timeout ) { return(-1); }
   if (tag == APPMSG){ deliver(msg); }
   do {
      send(nid, <GETVER_REQ>);
   }while ( !recv(nid, <GETVER_REP, ver2>) );
   if ( timeout ) { return(-1); }
   if ( ver2 == getVersion() ) { return(1); }
   state = NEWSES;
   while(1) {
      switch( state ) {
          case NEWSES: {
              sid = getUniqueSessionId();
              do {
                  send(nid, <NEWSES_REQ, sid, ver, size, digest> );
              } while( !recv(nid, <NEWSES_REP, sid, res, chunkno>) );
              if ( timeout ) { return(-1); }
              if ( res == TOO_BIG ) { return(-2); }
              if ( res == COMMITED ) { return(1); }
              if ( chunkno == getLastChunkNo() ) { state = COMMIT; break; }
              state = NXTCHUNK;
          }
          case NXTCHUNK: {
              chunkno++;
              d = getChunk(chunkno);
              chunkno2 = fixbits(chunkno);
              do {
                  send(nid, <NXTCHUNK_REQ, sid, chunkno2, sizeof(d), d> );
              } while( !recv(nid, <SES_ABORTED, sid>)
                  && !recv(nid, <NEXTCHUNK_REP, sid, chunkno>) );
              if ( timeout ) { return(-1); }
              if ( tag == SES_ABORTED ) { state = NEW_SES; break; }
              if ( chunkno < getLastChunkNo() ) { break; }
              state = COMMIT;
          }
          case COMMIT: {
              do {
                  send(nid, <COMMIT_REQ, sid> );
              } while( !recv(nid, <SES_ABORTED, sid>)
                                  && !recv(nid, <COMMIT_REP, sid, res>) )
              if ( timeout ) { return(-1); }
              if ( tag == SESABORTED_TAG ) { state = NEW_SES; break; }
              if ( res == CHECK_FAILED ) { state = NEW_SES; break; }
              return (1);
} } }   }
```

Quax pseudocode

```
int ver, digest, chunkno;  // version, digest and last chunk of last upload
int sid;                   // current session number
int cur_ver;              // version of the code that is running now
// RENEWED_ONLINE_INTERVAL fairly greater than INITIAL_ONLINE_INTERVAL
onReboot {
   sid = 0; cur_ver = getCurrentVersion();
   getUploadData(&ver, &digest, &chunkno);
   setTimeout(POLL_PERIOD, onPollPeriodTimeout);
}
onPollPeriodTimeout {
   if( sleep_enabled()) disable_sleep();
   send (0, <SYNCH>);
   if(isRouter()) setTimeout(POLL_PERIOD, onPollPeriodTimeout);
   else setTimeout(INITIAL_ONLINE_INTERVAL, onOnlineIntervalTimeout);
}
onOnlineIntervalTimeout {
   if( sid > 0) { sid = 0; abortSession(); }
   enable_sleep();
   setTimeout(POLL_PERIOD, onPollPeriodTimeout);
}
onRadioMsgSent(){
   if( sleep_enabled()){
      disable_sleep();
      setTimeout(INITIAL_ONLINE_INTERVAL, onOnlineIntervalTimeout);
   }
}
rx_handler( <int nid, msg> ) {
   switch( msg ) {
      case < APPMSG , app_msg> : {
          deliver(app_msg);
          return;
      }
      case <GETVER_REQ> : {
          send (nid, <GETVER_REP, cur_ver>);
          break;
      }
      case <NEWSES_REQ, sid2, ver2, size, digest2>: {
          if( !check_free_mem( size) )
              send(nid, <NEWSES_REP, TOO_BIG>);
          else if ( cur_ver == ver2)
              send(nid, <NEWSES_REP, COMMITED>);
          else {
              if ( (ver != ver2) || (digest != digest2) ) {
                  freeSavedChunks(); chunkno = 0;
                  ver = ver2; digest = digest2;
                  saveUploadData(ver, digest, chunkno);
```

```
            }
            sid = sid2;
            send(nid, <NEWSES_REP, sid, READY, chunkno>;
        }
        break;
    }
    case <NXTCHUNK_REQ, sid2, msg_chunkno, dlen, data>: {
        if( sid2 != sid)
            send(nid, <SESABORTED, sid2>);
        else {
            chunkno2 = unfixbits(msg_chunkno);
            if( chunkno2 != chunkno+1) // wrong chunk
                send(nid, <NXTCHUNK_REP, sid2, chunkno>);
            else {
                chunkno++;
                saveChunk(msg_chunkno, data, dlen);
                send(nid, <NXTCHUNK_REP, sid2, chunkno>);
            }
        }
        break;
    }
    case <COMMIT_REQ, sid2>: {
        if ( sid2 != sid ) {
            send(nid, <SESABORTED, sid2>);
        }
        else if ( computeDigest() != digest ) {
            freeSavedChunks(); chunkno = 0;
            saveUploadData(ver, digest, chunkno);
            send(nid, <COMMIT_REP, sid2, CHECK_FAILED>);
        }
        else {
            relocateSavedImage(); // handle the 2 parts appropriately
            send(nid, <COMMIT_REP, sid2, OK>);
            reboot();
        }
        break;
    }
    case <ABORT_REQ, sid2>: {
        if ( sid2 == sid ) {
            sid = 0; abortSession();
            send(nid, <SESABORTED, sid2>);
        }
        break;
    }
  }
  if(isRouter()) setTimeout(POLL_PERIOD, onPollPeriodTimeout);
  else setTimeout(RENEWED_ONLINE_INTERVAL, onOnlineIntervalTimeout);
}
```

Remote Reprogramming Protocol for Prisma WSN v11 (implemented by solution v1)

Introduction

This is a description of the protocol (to be) used to remotely program the nodes of the WSN.

The assumption is that remote reprogramming is initiated using a program that resides on a PC to which the WSN gateway is connected. The WSN gateway is assumed to act as a dummy router (not to be re-programmed). Each node must be addressed explicitly, following the usual addressing and packet transmission/reception conventions.

The protocol allows the client to: (i) retrieve a node's current software version, (ii) start a code upload session, (iii) upload the code image via a stop-and-wait protocol; and (iv) commit the uploaded image and reboot the node.

The protocol supports both full (entire image, i.e. ISOS and application code, is uploaded) and partial code update (ISOS code remains untouched and only application part is uploaded).[13]

The protocol is designed to support (but not to enforce) the continuation of a previously aborted upload, without requiring the client to re-send the entire code image from scratch.

The next sections describe the messages for the corresponding protocol exchanges, as well as the encapsulation format that application-level messages need to comply with.

[13] The code residing on the Quax is divided in two parts: core (i.e. hardware drivers, interrupt handler etc) and application. In order for the partial update to be feasible, these parts are allocated in two separate areas of the Quax's program memory (core part in the lowest area address-wise). Also, each of the two separate areas can be programmed without affecting the other (erase/write operations).

Application-level messages

Client to node & node to client:

```
AppMsg = AppMsgTag
         Payload

AppMsgTag = x01
Payload = byte[]
```

Comments:

Application-level messages, exchanged between the client and the node, need to comply with this format in order to be properly handled/forwarded (on both sides) and avoid interference with the “Remote reprogramming protocol” (and perhaps other system-level protocol) messages.

The payload is application-specific and its size is bound by the maximum radio packet payload minus one (for the *AppMsgTag*).

Synchronize client – node communication

Client to node:

N/A

Node to client:

```
SynchMsg = SynchMsgTag
SynchMsgTag = x02
```

Comments:

The node keeps its radio turned off for most of the time, unless the application code running on it has something to send to the client. However, occasionally, the client may also wish to initiate communication with the node, but this may fail due to the node having its radio turned off at that time.

To address this problem, a “polling” timeout is set on the node, which counts down until the next time the node should poll the client for messages. The node’s radio remains in sleep enabled mode until the timeout expires. When this happens, the node turns its radio on, immediately sends to the client a synchronization message and keeps its radio on for some time in order to receive messages from the client (if any). The node extends the time during which the radio is on each time it receives a message from the client. This makes it possible to have longer sessions without the node entering a sleep mode in between. If the node does not receive a message within its online interval, it reverts back to its usual sleeping pattern by resetting the “polling” timeout (and aborts the current session, if this is still pending).

The same procedure is followed every time application code turns the node’s radio on and sends an application-level message to the client. In this case, no explicit synchronization message is required, since the application message can serve the same purpose (inform the client about the node’s radio activity).

The client must wait for a synchronization (or application) message before attempting to send the first message (of a possibly longer session) to the node. Else, messages sent to the node will most probably be lost. Once a session is completed (or aborted), the client must wait for a fresh synchronization message in order to initiate a new session.

Get version of currently running code

Client to node:

```
GetVersionReq = GetVersionReqTag

GetVersionReqTag = x03
```

Node to client:

```
GetVersionReply = GetVersionReplyTag
                  ISOSCode_VersionNumber
                  AppCode_VersionNumber

GetVersionReplyTag = x04
ISOSCode_VersionNumber = uint32_t
AppCode_VersionNumber = unit32_t
```

Comments:

Retrieve the version number of the code (image) running on a node. The node replies with the version numbers of the currently running ISOS and application code. If the node supports only full updates, both version numbers are equal. If the node supports only partial updates, or both full and partial updates, the version numbers are different.

If the ISOS code running on the node is not the same as the one required by the application (not the ISOS version the application was compiled for), the client does not proceed to a code update session with the node.

Start new code upload session

Client to node:

```
StartSessionReq = StartSessionReqTag
                  SessionNumber
                  RemoteRerogrammingMode
                  VersionNumber
                  ImageSize
                  [ApplicationPartOffset]
                  DigestCode

StartSessionReqTag = x05
SessionNumber = uint32_t
RemoteRerogrammingMode = FULL | PARTIAL
FULL = x01
PARTIAL = x02
VersionNumber = uint32_t
ImageSize = uint16_t
ApplicationPartOffset = uint16_t
DigestCode = uint32_t
```

Node to client:

```
StartSessionReply = StartSessionReplyTag
                    SessionNumber
                    ReplyAnswer
                    [LastReceivedChunkNumber]

StartSessionReplyTag = x06
SessionNumber = uint32_t
ReplyAnswer = READY | TOO_BIG | COMMITTED | MODE_NOTSUPPORTED
READY = x01
TOO_BIG = x02
COMMITTED = x03
MODE_NOTSUPPORTED = 0x04
LastReceivedChunkNumber = uint16_t
```

Comments:

Start a new code upload session. The session number, assigned by the client, identifies the session. The remote reprogramming mode is identified by the respective field and can be either full (entire image, ISOS and application code, is uploaded) or partial (ISOS code remains untouched and only application part is uploaded). In the case of a partial update the version number identifies the application code version, while in the case of a full update, both the ISOS and application code versions. The application part offset indicates the offset of the application code start from the program memory start (field present only if the start session message concerns a full update and the new image supports **both** update modes).

A new session always overrides an old one (which might still be in progress). The node replies, using the session number of the request, indicating whether it is ready to proceed, or the image is too big, or the image is already committed (the node is already running this code), or the requested update mode is not supported.

The digest of the code image is sent by the client at the start of the session. The client must store this value and compare it to the digest computed locally as the code image is received. In order for a session to be committed, the digest computed by the node must be identical to the digest sent by the client.

If the node is ready to commence the code upload, it sends the number of the last code chunk already received (this is to support a continuation of a previous upload). More specifically, if the image has not been uploaded, in part, the chunk number specified by the node will be 0, else it will be greater than 0. If the requested chunk number is equal to the number of the last chunk, the node has already received the entire code image (and the client may proceed to commit the session).

Upload next chunk of code

Client to node:

```
NextChunkReq = NextChunkReqTag
               SessionNumber
               ChunkNumber
               ChunkLength
               Data

NextChunkReqTag = x07
SessionNumber = uint32_t
ChunkNumber = uint16_t
ChunkLength = uint8_t
Data = byte[ChunkLength]
```

Node to client:

```
NextChunkReply = NextChunkReplyTag
                 SessionNumber
                 ChunkNumber

NextChunkReplyTag = x08
SessionNumber = uint32_t
ChunkNumber = uint16_t
```

Comments:

The code upload is performed using a stop-and-wait protocol. The client sends the "next" chunk (requested by the node when it confirms the start of a new session), and the node replies to confirm the reception (and storage) of the chunk. The reply serves as a prompt for the next chunk, i.e., if the chunk number of the reply is *k* then the client is requested to send the $k+1^{th}$ chunk.

The size of code chunks is equal to maximum chunk size (naturally, this will be equal to the maximum payload size minus the protocol header size), except the last chunk which could be smaller. The last chunk is **explicitly** identified via the most significant bit of the chunk number: the last chunk has this bit set to 1 and all other chunks have this bit set to 0.

In the case of a full update, the code image consists of two disjoint parts (main part and interrupt vector table) which need to be mapped on different areas of the program memory. The 2^{nd} most significant bit of the chunk number is used to differentiate between these two parts. Specifically, this bit is set to 1 if the chunk contains part of the interrupt vector table, 0 otherwise. In the case of a partial update, there is no interrupt vector table part of the code image, so no chunk has its 2^{nd} most significant bit set to 1.

Identification of the interrupt vector and last chunk, via the 1^{st} and 2^{nd} most significant bits, applies only to the chunk numbers sent from the client to the node (the chunk numbers sent from the node to the client have these bits set to 0).

The node may save the code chunks it receives in non-volatile memory, so they can survive a reboot. Else, if the node reboots in the midst of a code upload session, the next session will start sending the image from scratch.

Note: The client may receive and must be able to handle a next chunk reply message with any chunk number (not necessarily the number of the most recently sent chunk). Also, the node may reply with a session aborted message (see respective section).

Commit uploaded code

Client to node:

```
CommitReq = CommitReqTag
            SessionNumber

CommitReqTag = x09
SessionNumber = uint32_t
```

Node to client:

```
CommitReply = CommitReplyTag
              SessionNumber
              (OK | CHECK_FAILED)

CommitReplyTag = x0A
SessionNumber = uint32_t
OK = x01
CHECK_FAILED = x02
```

Comments:

Once the image is successfully uploaded on the node, the client can request the node to commit and start running it. The node verifies that the digest of the image is correct (the client sends the digest when the code upload session is started). If this is not the case, it sends a negative reply. Else, it sends a positive reply, remaps the image to the proper memory location (as required by the bootloader) and reboots to start execution using the new code.

Note1: The node may also reply with a session aborted message (see respective section).

Note2: A positive reply does not mean that the code has been successfully committed. It merely acknowledges the fact that the digest is correct and that the node **will** start the commit process (copy the code to the proper memory location and reboot). The client can verify that the commit actually succeeded by retrieving the current version number(s) via a subsequent *GetVersion* message exchange; obviously, the node will be able to resume communication after it reboots, so the client should re-synch with the node (wait for a *Synch* message) before initiating the *GetVersion* exchange.

Abort code upload session

Client to node:

```
AbortSessionMsg = AbortSessionMsgTag
                  SessionNumber

AbortSessionMsgTag = x0B
SessionNumber = uint32_t
```

Node to client:

```
SessionAbortedMsg = SessionAbortedMsgTag
                    SessionNumber

SessionAbortedMsgTag = x0C
SessionNumber = uint32_t
```

Comments:

If the client decides to abort an upload session (this may happen for various reasons), it informs the node by sending an abort message. The node confirms this via a reply.

Also, the node may **unilaterally** decide to abort an ongoing session (e.g., using a timeout), to handle the case where the client does not “properly” complete, commit or abort a session (e.g., due to a failure, slow processing or networking problems). This decision is **silent**, i.e., the node does not attempt to notify the client about it. A unilateral abort may also occur due to a node reboot.

It is possible (e.g., due to using a small timeout) for the node to abort a session without the client actually having experienced a failure. Thus the node may receive a next chunk or commit request from the client for an aborted session. In this case, it replies with a session aborted message.

Note that when a session is aborted (explicitly by the client or silently by the node) the node may decide to discard the (partly) uploaded image in order to free memory. In this case, upload must start from scratch (using another session). Of course, the node is free to keep partly uploaded code images in memory so that upload can be resumed (via another session).

PC client pseudocode

```
int uploadCode(int nid) {
   setTimeout(A_VERY_GENEROUS_VALUE);
   while ( !recv(nid, <SYNCH>) &&
            !recv(nid, <APPMSG, msg>) ) {}
   if ( timeout ) { return(-1); }
   if (tag == APPMSG){ deliver(msg); }
   do {
      send(nid, <GETVER_REQ>);
   }while ( !recv(nid, <GETVER_REP, isos_ver, app_ver>) );
   if ( timeout ) { return(-1); }
   if ( upload_mode == PARTIAL && isos_ver != required_isos_ver )
   { return(-4);}
   if ( ver2 == getVersion() ) { return(1); }
   state = NEWSES;
   while(1) {
      switch( state ) {
         case NEWSES: {
            sid = getUniqueSessionId();
            do {
               if( upload_mode == PARTIAL )
                  send(nid, <NEWSES_REQ, upload_mode, sid, ver, size,    digest>);
               else
                  send(nid, <NEWSES_REQ, upload_mode, sid, ver,size,addr_offset, digest>);
            } while( !recv(nid, <NEWSES_REP, sid, res, chunkno>) );

            if ( timeout ) { return(-1); }
            if ( res == MODE_NOTSUPPORTED) { return(-3); }
            if ( res == TOO_BIG ) { return(-2); }
            if ( res == COMMITED ) { return(1); }
            if ( chunkno == getLastChunkNo() ) { state = COMMIT; break; }
            state = NXTCHUNK;
         }
         case NXTCHUNK: {
            chunkno++;
            d = getChunk(chunkno);
            chunkno2 = fixbits(chunkno);
            do {
               send(nid, <NXTCHUNK_REQ, sid, chunkno2, sizeof(d), d> );
            } while( !recv(nid, <SES_ABORTED, sid>)
                     && !recv(nid, <NEXTCHUNK_REP, sid, chunkno>) )
            if ( timeout ) { return(-1); }
            if ( tag == SES_ABORTED ) { state = NEW_SES; break; }
            if ( chunkno < getLastChunkNo() ) { break; }
            state = COMMIT;
         }
         case COMMIT: {
            do {
```

```
            send(nid, <COMMIT_REQ, sid> );
          } while( !recv(nid, <SES_ABORTED, sid>)
                    && !recv(nid, <COMMIT_REP, sid, res>) )
          if ( timeout ) { return(-1); }
          if ( tag == SESABORTED_TAG ) { state = NEW_SES; break; }
          if ( res == CHECK_FAILED ) { state = NEW_SES; break; }
          return (1);
        }
      }
    }
}
```

Quax pseudocode

```
int ver, digest, chunkno;   // version, digest and last chunk of last upload
int sid;                     // current session number
int cur_isos_ver;          // version of the ISOS code that is running now
int cur_app_ver;           // version of the application code that is running now
// RENEWED_ONLINE_INTERVAL fairly greater than INITIAL_ONLINE_INTERVAL
onReboot {
   sid = 0; cur_ver = getCurrentVersion();
   getUploadData(&ver, &digest, &chunkno);
   setTimeout(POLL_PERIOD, onPollPeriodTimeout);
}
onPollPeriodTimeout {
   if( sleep_enabled()) disable_sleep();
   send (0, <SYNCH>);
   if(isRouter()) setTimeout(POLL_PERIOD, onPollPeriodTimeout);
   else setTimeout(INITIAL_ONLINE_INTERVAL, onOnlineIntervalTimeout);
}
onOnlineIntervalTimeout {
   if( sid > 0) { sid = 0; abortSession(); }
   enable_sleep();
   setTimeout(POLL_PERIOD, onPollPeriodTimeout);
}
onRadioMsgSent(){
   if( sleep_enabled()){
      disable_sleep();
      setTimeout(INITIAL_ONLINE_INTERVAL, onOnlineIntervalTimeout);
   }
}
rx_handler( <int nid, msg> ) {
   switch( msg ) {
      case < APPMSG , app_msg> : {
         deliver(app_msg);
         break;
      }
      case <GETVER_REQ> : {
         send (nid, <GETVER_REP, cur_isos_ver, cur_app_ver>);
         break;
      }
      case <NEWSES_REQ, sid2, upload_mode,ver2, size, [addr_offset,] digest2>: {
         if( !check_free_mem( size) )
            send(nid, <NEWSES_REP, TOO_BIG>);
         else if ( !modeSupported(upload_mode))
            send(nid, <NEWSES_REP, MODE_NOTSUPPORTED >);
         else if ((upload_mode == FULL && cur_isos_ver == ver2)
                  || (upload_mode == PARTIAL && cur_app_ver == ver2))
            send(nid, <NEWSES_REP, COMMITED>);
         else {
            if ( (ver != ver2) || (digest != digest2) ) {
```

```
            freeSavedChunks(); chunkno = 0;
            ver = ver2; digest = digest2;
            saveUploadData(ver, digest, chunkno, addr_offset);
         }
         sid = sid2;
         send(nid, <NEWSES_REP, sid, READY, chunkno>;
      }
      break;
   }
   case <NXTCHUNK_REQ, sid2, msg_chunkno, dlen, data>: {
      if( sid2 != sid)
         send(nid, <SESABORTED, sid2>);
      else {
         chunkno2 = unfixbits(msg_chunkno);
         if( chunkno2 != chunkno+1) // wrong chunk
            send(nid, <NXTCHUNK_REP, sid2, chunkno>);
         else {
            chunkno++;
            saveChunk(msg_chunkno, data, dlen);
            send(nid, <NXTCHUNK_REP, sid2, chunkno>);
         }
      }
      break;
   }
   case <COMMIT_REQ, sid2>: {
      if ( sid2 != sid )       {
         send(nid, <SESABORTED, sid2>);
      }
      else if ( computeDigest() != digest ) {
         freeSavedChunks(); chunkno = 0;
         saveUploadData(ver, digest, chunkno);
         send(nid, <COMMIT_REP, sid2, CHECK_FAILED>);
      }
      else {
         send(nid, <COMMIT_REP, sid2, OK>);
         relocateSavedImage();
         reboot();
      }
      break;
   }
   case <ABORT_REQ, sid2>: {
      if ( sid2 == sid ) {
         sid = 0; abortSession();
         send(nid, <SESABORTED, sid2>);
      }
      break;
   }
}
```

Remote Reprogramming Protocol for Prisma WSN v13 (implemented by solution v3)

Introduction

This is a description of the protocol (to be) used to remotely program the nodes of the WSN.

The assumption is that remote reprogramming is initiated using a program that resides on a PC to which the WSN gateway is connected. The WSN gateway is assumed to act as a dummy router (not to be re-programmed). When sending unicast packets each node must be addressed explicitly, following the usual addressing and packet transmission/reception conventions. On the contrary, when sending native broadcast packets, a special address is used.

The protocol allows the client to: (i) start a code upload session, (ii) upload the code image via a non-ARQ, NACK based, propagate-repair scheme, (iii) commit the uploaded image and reboot the node, (iv) retrieve a node's current software version to confirm the successful reboot.

The protocol supports out-of-order reception of code chunk messages belonging to the same transfer unit (code unit/segment of 512 bytes, from this point on referred to as segment) and allows scalable 1-to-many code updates, by postponing the acknowledgement of the chunk messages until after the transmission of the last chunk of the transfer unit.

The protocol supports both full (entire image, i.e. ISOS and application code, is uploaded) and partial code update (ISOS code remains untouched and only application part is uploaded).[14]

The protocol is designed to support (but not to enforce) the continuation of a previously aborted upload, without requiring the client to re-send the entire code image from scratch.

The next sections describe the messages for the corresponding protocol exchanges, as well as the encapsulation format that application-level messages need to comply with.

[14] The code residing on the Quax is divided in two parts: core (i.e. hardware drivers, interrupt handler etc) and application. In order for the partial update to be feasible, these parts are allocated in two separate areas of the Quax's program memory (core part in the lowest area address-wise). Also, each of the two separate areas can be programmed without affecting the other (erase/write operations).

Application-level messages

Client to node & node to client:

```
AppMsg = AppMsgTag
         Payload

AppMsgTag = x01
Payload = byte[]
```

Comments:

Application-level messages, exchanged between the client and the node, need to comply with this format in order to be properly handled/forwarded (on both sides) and avoid interference with the “Remote reprogramming protocol” (and perhaps other system-level protocol) messages.

The payload is application-specific and its size is bound by the maximum radio packet payload minus one (for the *AppMsgTag*).

Synchronize client – node communication (deprecated)

Client to node:

N/A

Node to client (no longer used):

```
SynchMsg = SynchMsgTag
SynchMsgTag = x02
```

Comments:

The radio polling strategy on the node is preserved, but the SYNCH messages are no longer necessary. Specifically, since each ZigBee parent caches messages on behalf of its children, it is guaranteed that any message sent by the client will eventually reach the destination node (when the latter turns its radio on). Moreover, the radio polling strategy guaranties that the node will turn its radio on periodically to check for messages pending for reception. The client just has to be patient enough when the node is expected to have its radio off (e.g. when starting a code upload session).

Get version of currently running code

Client to node:

```
GetVersionReq = GetVersionReqTag

GetVersionReqTag = x03
```

Node to client:

```
GetVersionReply = GetVersionReplyTag
                  ISOSCode_VersionNumber
                  AppCode_VersionNumber

GetVersionReplyTag = x04
ISOSCode_VersionNumber = uint32_t
AppCode_VersionNumber = unit32_t
```

Comments:

Retrieve the version number of the code (image) running on a node. The node replies with the version numbers of the currently running ISOS and application code. If the node supports only full updates, both version numbers are equal. If the node supports only partial updates, or both full and partial updates, the version numbers are different.

Start new code upload session

Client to node:

```
StartSessionReq = StartSessionReqTag
                  SessionNumber
                  RemoteRerogrammingMode
                  ISOSCode_VersionNumber
                  AppCode_VersionNumber
                  ImageSize
                  [ApplicationPartOffset]
                  DigestCode

StartSessionReqTag = x05
SessionNumber = uint32_t
RemoteRerogrammingMode = FULL | PARTIAL
FULL = x01
PARTIAL = x02
ISOSCode_VersionNumber = uint32_t
AppCode_VersionNumber  = uint32_t
ImageSize = uint16_t
ApplicationPartOffset = uint16_t
DigestCode = uint32_t
```

Node to client:

```
StartSessionReply = StartSessionReplyTag
                    SessionNumber
                    ReplyAnswer
                    [LastReceivedSegment]
                    [ReceivedChunksBitmap]

StartSessionReplyTag = x06
SessionNumber = uint32_t
ReplyAnswer = READY | TOO_BIG | COMMITTED | MODE_NOTSUPPORTED |
ISOS_CODE_INCONSISTENCY
READY = x01
TOO_BIG = x02
COMMITTED = x03
MODE_NOTSUPPORTED = x04
ISOS_CODE_INCONSISTENCY = x05
LastReceivedSegment = uint8_t
ReceivedChunksBitmap = uint16_t
```

Comments:

Start a new code upload session. The session number, assigned by the client, identifies the session. The remote reprogramming mode is identified by the respective field and can be either full (entire image, ISOS and application code, is uploaded) or partial (ISOS code remains untouched and only application part is uploaded). In the case of a partial update the ISOS code and Application code version numbers are different (identifying the ISOS and application code versions respectively), while in the case of a full update, both version numbers are equal (identifying both the ISOS and application code versions). The application part offset indicates the offset of the application code start from the program memory start (field present only if the start session message concerns a full update and the new image supports **both** update modes).

A new session always overrides an old one (which might still be in progress). The node replies, using the session number of the request, indicating whether it is ready to proceed, or the image is too big, or the image is already committed (the node is already running this code), or the requested update mode is not supported, or the ISOS code running on the node is not the same as the one required by the new application in a partial update (not the ISOS version the application was compiled for).

The digest of the code image is sent by the client at the start of the session. The client must store this value and compare it to the digest computed locally as the code image is received. In order for a session to

be committed, the digest computed by the node must be identical to the digest sent by the client.

If the node is ready to commence the code upload, it sends the number of the code segment to which the last code chunk received belongs. Additionally, the node sends a bitmap indicating all the chunks of this segment already received ('1' indicates reception of the respective chunk, e.g. if the 1st chunk of the segment has been received, the least significant bit of the bitmap is set to '1' and so on). Consequently, continuation of a previous upload is possible. More specifically, if the image has not been uploaded, in part, the segment number specified by the node will be 0, else it will be greater than 0. If the requested segment number is equal to the number of the last segment and the bitmap field indicates reception of all chunks of the last segment, the node has already received the entire code image (and the client may proceed to commit the session).

In case of a 1-to-many code update, the client collects all start session replies reporting READY and computes the minimum segment reported. If this is greater than 0 (every node has downloaded at least a part of the image), the client uses the bitmaps of the replies bearing the specified segment number, to infer the missing chunks of the segment (union of the missing chunks of each node reporting this segment).

Upload next chunk of code

Client to node:

```
NextChunkReq = NextChunkReqTag
               SessionNumber
               SegmentNumber
               ChunkNumber
               ChunkLength
               Data

NextChunkReqTag = x07
SessionNumber = uint32_t
SegmentNumber= uint8_t
ChunkNumber = uint8_t
ChunkLength = uint8_t
Data = byte[ChunkLength]
```

Node to client:

N/A

Comments:

The code upload is performed using a non-ARQ, NACK based, propagate-repair protocol. The client sends all the chunks of the currently propagated segment in successive NextChunkReq messages (no reply by the node).

The size of code segments is equal to 512 bytes, except for the last segment which might be smaller. The last segment is **explicitly** identified via the most significant bit of the segment number: the last chunk has this bit set to 1 and all other chunks have this bit set to 0.

The size of code chunks is equal to maximum chunk size (naturally, this will be equal to the maximum payload size minus the protocol header size), except the last chunk which could be smaller. The last chunk of each segment is **explicitly** identified via the most significant bit of the chunk number: the last chunk has this bit set to 1 and all other chunks have this bit set to 0.

In the case of a full update, the code image consists of two disjoint parts (main part and interrupt vector table) which need to be mapped on different areas of the program memory. The 2nd most significant bit of the segment number is used to differentiate between these two parts. Specifically, this bit is set to 1 if the segment contains part of the interrupt vector table, 0 otherwise. In the case of a partial update, there is no interrupt vector table part of the code image, so no segment has its 2nd most significant bit set to 1.

Identification of the interrupt vector segment, last segment and last chunk of a segment, via the 1st and 2nd most significant bits, applies only to the segment and chunk numbers sent from the client to the node (the segment and chunk numbers sent from the node to the client have these bits set to 0).

The node may save the code segments (and chunks) it receives in non-volatile memory, so they can survive a reboot. Else, if the node reboots in the midst of a code upload session, the next session will start sending the image from scratch.

Report missing chunks of a segment

Client to node:

```
NACKReq =   NACKReqTag
            SessionNumber
            SegmentNumber

NACKReqTag = x09
SessionNumber = uint32_t
SegmentNumber = unint8_t
```

Node to client:

```
NACKRep =     NACKRepTag
              SessionNumber
              SegmentNumber
              MissingChunksBitmap

NACKRepTag = x0A
SessionNumber = uint32_t
SegmentNumber = unint8_t
ReceivedChunksBitmap = uint16_t
```

Comments:

In a 1-to-many code update, after propagating all the chunks of the current segment, the client sends a negative acknowledgement request and waits a specified amount of time to get informed about any chunks that the nodes failed to receive during the propagation. The segment number field indicates the current segment, for which the client requests acknowledgement.

On reception of a NACKReq, the node replies with a NACKRep only if it has not received one or more chunks of the current segment (segment number in request). The chunks bitmap field of the reply indicates the already received chunks.

Following, the client repairs all missing chunks reported by the nodes, via successive NextChunkReq messages, as described in the respective section. Then, the client sends a new NACKReq and the above procedure is repeated, if necessary.

The node may also reply to a NACKReq with an ACKRep (see next section).

If the node receives a NACKReq with a session number different than its local session number or segment number greater its local segment number, the node silently aborts the session. This way, a node that has fallen behind in will not delay the code update procedure for the rest of the nodes. Any such nodes are taken care of by the client after the current upload session is complete.

Note: This phase is skipped in a point-to-point update (see next section).

Report reception of all chunks of a segment

Client to node:

```
ACKReq =    ACKReqTag
            SessionNumber
            SegmentNumber

ACKReqTag = x0B
SessionNumber = uint32_t
SegmentNumber = unint8_t
```

Node to client:

```
ACKRep =      ACKRep Tag
              SessionNumber
              SegmentNumber

ACKRepTag = x0C
SessionNumber = uint32_t
SegmentNumber = unint8_t
```

Comments:

In a 1-to-many code update, if the client does not receive any NACKRep messages during the specified amount of time and after sending a NACKReq, it sends a positive acknowledgement request (ACKReq) to confirm that all nodes are ready for the next segment's propagation. Then, the client waits for replies and if it does not receive an appropriate reply from each node, it sends the request again.

A node always replies to an ACKReq. If it is missing any chunks of the current segment it replies with a NACKRep. If it has already received all chunks of the current segment it replies with an ACKRep. If the node receives an ACKReq with a session number different than its local session number or segment number greater its local segment number, the node replies with a SessionAbortedMsg and aborts the session.

Once all nodes reply with an appropriate message, or a limited amount of efforts is reached the client can proceed. Any nodes that replied with a SessionAbortedMsg or failed to reply at all are excluded from the current upload session. Then, if the client received at least one NACKRep, it repairs the missing chunks reported as discussed in the previous section. Otherwise, the client proceeds to the propagation of the next segment, or the commit phase if there are no more segments to propagate (next section).

In a point-to-point code update, after propagating the current segment, the client sends an ACKReq and not a NACKReq. Otherwise, the repair phase is similar to the 1-to-many case, with small differences in the amount of efforts and the timeouts.

Note: In a 1-to-many code update, a node replies with an ACKRep to the first NACKReq it receives following one or more chunk messages of the current transfer unit. This way, if the client receives an ACKRep from every node as a reply to a NACKReq, it can skip the ACKRep phase and proceed.

Commit uploaded code

Client to node:

```
CommitReq = CommitReqTag
            ISOSCode_VersionNumber
            AppCode_VersionNumber_

CommitReqTag = x0D
ISOSCode_VersionNumber = uint32_t
AppCode_VersionNumber = unit32_t
```

Node to client:

```
CommitReply = CommitReplyTag
              ISOSCode_VersionNumber
              AppCode_VersionNumber
              (OK | CHECK_FAILED | NO_IMAGE|
               ISOS_CODE_INCONSISTENCY)

CommitReplyTag = x0E
ISOSCode_VersionNumber = uint32_t
AppCode_VersionNumber = unit32_t
OK = x01
CHECK_FAILED = x02
NO_IMAGE = x03
ISOS_CODE_INCONSISTENCY = x05
```

Comments:

Once the image is successfully uploaded on the node(s), the client can request the node(s) to commit and start running it.

On receiving a commit request, a node first checks if the ISOS code version number of the request is identical to the one it is currently running and if not, it sends a reply stating the inconsistency (ISOS_CODE_INCONSISTENCY). Otherwise, if the node has the entire image, it verifies that the digest of the image is correct (the client sends the digest when the code upload session is started). If digest check fails, it sends an appropriate negative reply (CHECK_FAILED). If the digest check succeeds, it sends a positive reply (OK), remaps the image to the proper memory location (as required by the bootloader) and reboots to start execution using the new code. If the node is missing a part or even the entire new image, it replies with a commit reply indicating the absence of (part of) the image. Finally, the node might be already running the image that the client requests to commit, in which case the node replies again with a positive reply (OK).

Once all nodes have replied (in the 1-to-many case), or the single target node replies (in the point-to-point case), the node proceeds to a *GetVersion* message exchange to confirm that the node(s) rebooted successfully to the new image.

Note1: Since the commit request (in the 1-to-many case) may reach only a subset of the nodes, it can lead the network to an unpredicted state. Thus, nodes that failed to receive the request may be temporarily unreachable and go out of session, before the commit request reaches them. For this reason, the commit request was modified to be session-less, bearing the ISOS and Application code versions instead of the session number, in order to refer to a specific image download.

Abort code upload session

Client to node:

```
AbortSessionMsg = AbortSessionMsgTag
                  SessionNumber

AbortSessionMsgTag = x0F
SessionNumber = uint32_t
```

Node to client:

```
SessionAbortedMsg = SessionAbortedMsgTag
                    SessionNumber

SessionAbortedMsgTag = x10
SessionNumber = uint32_t
```

Comments:

If the client decides to abort an upload session (this may happen for various reasons), it informs the node by sending an abort message. The node confirms this via a reply.

Also, the node may **unilaterally** decide to abort an ongoing session (e.g., using a timeout), to handle the case where the client does not “properly” complete, commit or abort a session (e.g., due to a failure, slow processing or networking problems). This decision is **silent**, i.e., the node does not attempt to notify the client about it. A unilateral abort may also occur due to a node reboot.

It is possible (e.g., due to using a small timeout) for the node to abort a session without the client actually having experienced a failure. Thus the node may receive an ack request or commit request from the client for an aborted session. In this case, it replies with a session aborted message.

Note that when a session is aborted (explicitly by the client or silently by the node) the node may decide to discard the (partly) uploaded image in order to free memory. In this case, upload must start from scratch (using another session). Of course, the node is free to keep partly uploaded code images in memory so that upload can be resumed (via another session).

PC client pseudocode (1-to-many)

```
// quaxes_to_update list is produced by a pre-init discovery phase
// INTERCHUNK_TIMEOUT,INTERGROUP_TIMEOUT and CHUNK_TRANSMISSION_GROUP_SIZE defined via
// experiments on the native broadcast efficiency.
int uploadCode(Quaxes_list quaxes_to_update) {
   Quaxes_list quaxes_replied, quaxes_ready;
   int cur_segment_no, cur_chunk_no, efforts = 0;
   List<int> chunks_list;
   state = NEWSES;
   upload_mode = PARTIAL;
   while(1) {
    switch( state ) {
     case NEWSES: {
        do {
           efforts++;
           sid = getUniqueSessionId();
           quaxes_replied = {};quaxes_ready = {};
           cur_segment_no = MAX_SEG_NO;
           send(bcast_addr, <NEWSES_REQ,upload_mode,sid,isos_ver,app_ver,size, digest>);
           setTimeout(A_LITTLE_BIGGER_THAN_POLL_PERIOD);
           while( recv(nid, <NEWSES_REP, sid, res, segment_no, chunks_bitmap>, timeout)
                     && (nid !E quaxes_replied))
           {
              if( nid !E quaxes_replied) quaxes_replied += nid;
              if(res = READY) {
                 if( nid !E quaxes_ready) quaxes_ready += nid;
                 if(segment_no < cur_segment_no) {
                    cur_segment_no = segment_no;
                    chunks_list = {};
                    if( MissingChunks(chunks_bitmap))
                       chunks_list += get_missing_chunks(chunks_bitmap);
                    else
                    {
                       cur_segment_no++;
                       chunks_list = get_all_seg_chunks(cur_segment_no);
                    }
                 }
                 else if( segment_no == cur_segment_no && MissingChunks(chunks_bitmap)) {
                    chunks_list += get_missing_chunks(chunks_bitmap);
                 }
              }
              else {
                 if( nid !E quaxes_replied) quaxes_replied += nid;
              }
           }
        } while( !subsetOf(quaxes_to_update, quaxes_replied) || efforts == MAX_EFFORTS
                  || user_interrupted_waiting());
        if( quaxes_to_update !=  quaxes_ready) quaxes_to_update = quaxes_ready;
```

```
   if( quaxes_to_update = {}) return 2;
   if ( cur_segment_no == getLastSegNo() && chunks_list == {} ) {
      state = COMMIT;
      break;
   }
   else if (chunkno == MAX_CHUNK_NO)  {
      cur_segment_no++;
      chunks_list = get_all_seg_chunks(cur_segment_no);
   }
   state = NXTCHUNK;
}
case NXTCHUNK: {
   cur_segment_no2 = fixbits(cur_segment_no);
   int bcasts_in_a_row = 0;
   foreach(chunkno in chunks_list) {
      bcasts_in_a_row++;
      d = getChunk(cur_segment_no, chunkno);
      chunkno2 = fixbits(chunkno);
      send(bcast_addr, <NXTCHUNK_REQ, sid, cur_segment_no2,chunkno2,sizeof(d),d>);
      if( bcasts_in_a_row % CHUNK_TRANSMISSION_GROUP_SIZE == 0 )
         wait( INTERGROUP_TIMEOUT);
      else wait( INTERCHUNK_TIMEOUT);
   }
   state = COLLECT_NACKs;
}
case COLLECT_NACKs: {
   quaxes_ready = {};
   send(bcast_addr, <NACK_REQ, sid, cur_segment_no> );
   setTimeout(MODERATE_VALUE);
   chunks_list = {};
   while( recv(nid, <tag, sid, cur_segment_no[, chunks_bitmap]>, timeout) ){
      if( tag == ACK_REP && && nid !E quaxes_ready) quaxes_ready += nid;
      else {
         chunks_list += get_missing_chunks(chunks_bitmap);
      }
   }
   if( quaxes_to_update == quaxes_ready ) {
      if(cur_segment_no < getLastSegNo()) {state = COMMIT;break;}
      cur_segment_no++;chunks_list = get_all_seg_chunks(cur_segment_no);
   }
   else if( chunks_list == {} ) {state = COLLECT_ACKs;efforts = 0;break;}
   state = NXTCHUNK;break;
}
case COLLECT_ACKs: {
   quaxes_ready = {};
   quaxes_replied = {};
   while( !subsetOf(quaxes_to_update, quaxes_replied) && efforts < MAX_EFFORTS) {
      efforts++;quaxes_replied = {};quaxes_ready = {};
```

```
      send(bcast_addr, <ACK_REQ, sid, cur_segment_no> );
      missing_chunks = {};
      setTimeout(MODERATE_VALUE);
      while(recv(nid,<tag,sid[,cur_segment_no2,missing_chunks_bitmap]>,timeout))
      {
         if( tag != SESSION_ABORTED && nid !E quaxes_replied)
            quaxes_replied += nid;
         if(tag == SESSION_ABORTED && nid E quaxes_to_update) {
            quaxes_to_update -= nid;
         }
         else if( tag == ACK_REP && cur_segment_no2 == cur_segment_no
               && nid !E quaxes_ready){
            quaxes_ready += nid;
         }
         else if(tag == NACK_REP && cur_segment_no2 == cur_segment_no){
            chunks_list += get_missing_chunks(chunks_bitmap);
         }
      }
   }
   if(!subsetOf(quaxes_to_update, quaxes_replied) )
       quaxes_to_update = quaxes_replied;
   if(quaxes_to_update == {}) return 2;
   if( quaxes_to_update != quaxes_ready ) {state = NXTCHUNK;break;}
   if( cur_segment_no < getLastSegNo()) {
      cur_segment_no++;chunks_list = get_all_seg_chunks(cur_segment_no);
      state=NXTCHUNK;break;
   }
   state = COMMIT;efforts = 0;
}
case COMMIT: {
   quaxes_replied = {};quaxes_ready = {};
   while( !subsetOf(quaxes_to_update, quaxes_replied) && efforts < MAX_EFFORTS) {
      send(bcast_addr, <COMMIT_REQ, isos_ver, app_ver> );
      efforts++;
      setTimeout(MODERATE_VALUE);
      while( recv(nid, <COMMIT_REP, isos_ver, app_ver, res>, timeout) ) {
         if( nid !E quaxes_replied) quaxes_replied += nid;
         if( res != OK  && nid !E quaxes_ready) quaxes_ready += nid;
      }
   }
   if(!subsetOf(quaxes_to_update, quaxes_replied) )
       quaxes_to_update = quaxes_replied;
   if(quaxes_to_update == {}) return 2;
   state = GET_VER;efforts = 0;//to confim successful reboot to new image
}
case GET_VER: {
   quaxes_replied = {}; quaxes_ready = {};
   while( !subsetOf(quaxes_to_update, quaxes_replied) && efforts < MAX_EFFORTS) {
      efforts++;
```

```
        send(bcast_addr, <GETVER_REQ>);
        setTimeout(A_VERY_GENEROUS_VALUE);
        while ( recv(nid, <GETVER_REP, isos_ver1, app_ver2>, timeout) ) {
          if( isos_ver1 == isos_ver && nid !E quaxes_replied) quaxes_replied += nid;
          if( isos_ver1 == isos_ver && nid !E quaxes_to_update )
             quaxes_to_update += nid;
          if( isos_ver1 == isos_ver && app_ver2 == app_ver
            && nid !E quaxes_ready) quaxes_ready += nid;
        }
      }
      if( quaxes_ready != quaxes_to_update) {
        quaxes_to_update -= quaxes_ready;
        state = NEWSES;efforts = 0;break;
      }
      return 1;
    }
   }
  }
```

PC client pseudocode (point-to-point)

```
// quaxes_to_update list is produced by a pre-init discovery phase
// PNT2PNT_INTERCHUNK_TIMEOUT defined via experiments on the native unicast efficiency.
int uploadCode(int nid) {
   int cur_segment_no, cur_chunk_no, efforts = 0;
   List<int> chunks_list;
   state = NEWSES;
   upload_mode = PARTIAL;
   while(1) {
    switch( state ) {
     case NEWSES: {
        do {
           efforts++;
           sid = getUniqueSessionId();
           send(nid, <NEWSES_REQ, upload_mode, sid, isos_ver, app_ver, size, digest>);
           setTimeout(A_LITTLE_BIGGER_THAN_POLL_PERIOD);
           if( recv(nid, <NEWSES_REP, sid, res, segment_no, chunks_bitmap>, timeout))
           {
              if(res = READY) {
                 chunks_list = get_missing_chunks( chunks_bitmap);
                 if ( segment_no == getLastSegNo() && chunks_list == {} ) {
                    state = COMMIT;
                    break;
                 }
                 else if (chunkno == MAX_CHUNK_NO) {
                    cur_segment_no++;
                    chunks_list = get_all_seg_chunks(cur_segment_no);
                 }
                 state = NXTCHUNK;
              }
              else {
                 return -1
              }
           }
        } while(efforts == MAX_PNT2PNT_EFFORTS || user_interrupted_waiting());
        if(user_interrupted_waiting())  return 2;
        if(state == NXTCHUNK) break;
        return -1;
     }
     case NXTCHUNK: {
        cur_segment_no2 = fixbits(cur_segment_no);
        foreach(chunkno in chunks_list) {
           d = getChunk(cur_segment_no, chunkno);
           chunkno2 = fixbits(chunkno);
           send(nid, <NXTCHUNK_REQ, sid, cur_segment_no2, chunkno2, sizeof(d), d> );
           wait( PNT2PNT_INTERCHUNK_TIMEOUT);
        }
        state = COLLECT_ACKs;
```

```
   efforts = 0;
}
case COLLECT_ACKs: {
   do {
      efforts++;
      send(nid, <ACK_REQ, sid, cur_segment_no> );
      setTimeout(MODERATE_VALUE);
      if( recv(nid, <tag,sid[,cur_segment_no2,missing_chunks_bitmap]>, timeout))
      {
         if(tag == SESSION_ABORTED) {
            state = NEWSES;
            efforts = 0;
            break;
         }
         else if( tag == ACK_REP && cur_segment_no2 == cur_segment_no){
            if( cur_segment_no < getLastSegNo()) {
               cur_segment_no++;
               chunks_list = get_all_seg_chunks(cur_segment_no);
               state=NXTCHUNK;break;
            }
            else
            {
               state = COMMIT;efforts = 0;
            }
            break;
         }
         else if(tag == NACK_REP && cur_segment_no2 == cur_segment_no){
            chunks_list = get_missing_chunks( missing_chunks_bitmap);
            state = NXTCHUNK;break;
         }
      }

   } while(efforts == MAX_PNT2PNT_EFFORTS || user_interrupted_waiting());
   if(user_interrupted_waiting())  return 2;
   if(state == COLLECT_ACKs) return -1;
   break;
}
case COMMIT: {
   do {
      send(nid, <COMMIT_REQ, isos_ver, app_ver> );
      efforts++;
      setTimeout(MODERATE_VALUE);
      if( recv(nid, <COMMIT_REP, isos_ver, app_ver, res>, timeout) ) {
         if( res == OK ){
            state = GET_VER;efforts = 0;         //confirm successful reboot
            break;
         }
         else if(res == CHECK_FAILED) {
```

```
                state = NEWSES;efforts = 0;          //confirm successful reboot
              }
              else return -2
            }
          }while(efforts == MAX_PNT2PNT_EFFORTS || user_interrupted_waiting());
          if(user_interrupted_waiting())  return 2;
          if(state == COMMIT) return -1;
          break;
        }
        case GET_VER: {
          do {
            efforts++;
            send(nid, <GETVER_REQ>);
            setTimeout(A_VERY_GENEROUS_VALUE);
            if( recv(nid, <GETVER_REP, isos_ver1, app_ver2>, timeout) ) {
              if( isos_ver1 == isos_ver && app_ver2 == app_ver) return 1;
              else {
                state = NEWSES;efforts = 0;          //confirm successful reboot
              }
            }
          }while(efforts == MAX_PNT2PNT_EFFORTS || user_interrupted_waiting());
          if(user_interrupted_waiting())  return 2;
          if(state == GET_VER) return -1;
          break;
        }
      }
    }
}
```

Quax pseudocode

```
int upt_isos_ver, upt_app_ver, digest, segment_no, chunkno;
int chunks_bitmap;   // bitmap indicating chunks received
int segment_acked;
int sid;             // current session number
int cur_isos_ver;    // version of the ISOS code that is running now
int cur_app_ver;     // version of the application code that is running now
// RENEWED_ONLINE_INTERVAL fairly greater than INITIAL_ONLINE_INTERVAL
onReboot {
   sid = 0; cur_ver = getCurrentVersion();
   getUploadData(&upt_app_ver, &digest, &segment_no, &chunkno, &chunks_bitmap);
   setTimeout(POLL_PERIOD, onPollPeriodTimeout);
}
onPollPeriodTimeout {
   if( sleep_enabled()) disable_sleep();
   send (0, <SYNCH>);
   if(isRouter()) setTimeout(POLL_PERIOD, onPollPeriodTimeout);
   else setTimeout(INITIAL_ONLINE_INTERVAL, onOnlineIntervalTimeout);
}
onOnlineIntervalTimeout {
   if( sid > 0) { sid = 0; abortSession(); }
   enable_sleep();
   setTimeout(POLL_PERIOD, onPollPeriodTimeout);
}
onRadioMsgSent()
{  if( sleep_enabled())
   {
    disable_sleep();
    setTimeout(INITIAL_ONLINE_INTERVAL, onOnlineIntervalTimeout);
   }
}
rx_handler( <int nid, msg> ) {
   switch( msg ) {
    case < APPMSG , app_msg> : {
     deliver(app_msg);
     break;
    }
    case <GETVER_REQ> : {
     send (nid, <GETVER_REP, cur_isos_ver, cur_app_ver>);
     break;
    }
    case <NEWSES_REQ, upload_mode, sid, isos_ver, app_ver, size, digest2>:{
     if( !check_free_mem( size ) )
        send(nid, <NEWSES_REP, TOO_BIG>);
     else if(cur_isos_ver != isos_ver)
        send(nid, <NEWSES_REP, ISOS_CODE_INCONSISTENCY>);
     else if(app_ver == cur_app_ver)
        send(nid, <NEWSES_REP, COMMITED>);
```

```
 else {
    if (upt_app_ver != app_ver) {
       freeSavedChunks(); segment_no =0; chunkno=0;
       upt_app_ver = app_ver; digest = digest2;
       chunks_bitmap = 0;
       saveUploadData(&upt_app_ver, &digest, &segment_no, &chunkno, &chunks_bitmap);
    }
    sid = sid2;
    segment_acked = 0;
    send(nid, <NEWSES_REP, sid, READY, segment_no, chunks_bitmap>;
 }
 break;
}
case <NXTCHUNK_REQ, sid2, msg_segment_no, msg_chunkno, dlen, data>: {
 segment_no2 = unfixbits(msg_segment_no);
 if(sid2 != sid || (segment_no2 > segment_no && segment_no2 != segment_no + 1)
 {sid=0;abortSession();break;}
 if(segment_acked) segment_acked = 0;
 if(segment_no2 < segment_no){break;}
 if(segment_no2 == segment_no + 1) {
    if(missing_chunks(chunks_bitmap, msg_segment_no)){
       sid=0;abortSession();break;
    }
    segment_no++; chunks_bitmap = 0;
 }
 chunkno2 = unfixbits(msg_chunkno);
 saveChunk(msg_chunkno, data, dlen);
 updateBitmap(&chunks_bitmap, chunkno2);
}
case <NACK_REQ, sid2, msg_segment_no>: {
 segment_no2 = unfixbits(msg_segment_no);
 if( sid!= sid2 || segment_no < segment_no2)
 {sid=0;abortSession();}
 if( segment_no == segment_no2 && missing_chunks(chunks_bitmap, msg_segment_no)){
    send(nid, <NACK_REP, sid, segment_no, chunks_bitmap>);
 }
 else if(!segment_acked){
    segment_acked = 1;
    send(nid, <ACK_REP, sid, segment_no2>);
 }
}
case <ACK_REQ, sid2, msg_segment_no>: {
 segment_no2 = unfixbits(msg_segment_no);
 if( sid!= sid2 || segment_no < segment_no2) {
    sid = 0; abortSession();
    send(nid, <SESABORTED, sid2>);
    break;
 }
```

```
      if( segment_no == segment_no2 && missing_chunks(chunks_bitmap, msg_segment_no)){
         send(nid, <NACK_REP, sid, cur_segment_no, chunks_bitmap>);
      }
      else{
         send(nid, <ACK_REP, sid, cur_segment_no>);
      }
     }
     case <COMMIT_REQ, isos_ver2, app_ver2>: {
      if ( isos_ver2 != cur_isos_ver ){
         send(nid, <COMMIT_REP, isos_ver2, app_ver2, ISOS_CODE_INCONSISTENCY>);
      }
      else if (cur_app_ver == app_ver2){
         send(nid, <COMMIT_REP, isos_ver2, app_ver2, OK>);
      }
      else if ( app_ver2 != upt_app_ver){
         send(nid, <COMMIT_REP, isos_ver2, app_ver2, NO_IMAGE>);
      }
      else if ( computeDigest() != digest ) {
         freeSavedChunks(); segment_no =0; chunkno=0; chunks_bitmap=0;
         saveUploadData(&upt_app_ver, &digest, &segment_no, &chunkno, &chunks_bitmap);
         send(nid, <COMMIT_REP, isos_ver2, app_ver2, CHECK_FAILED>);
      }
      else {
         send(nid, <COMMIT_REP, isos_ver2, app_ver2, OK>);
         relocateSavedImage();
         reboot();
      }
      break;
     }
     case <ABORT_REQ, sid2>: {
      if ( sid2 == sid ) {
         sid = 0; abortSession();
         send(nid, <SESABORTED, sid2>);
      }
      break;
     }
    }
    if(isRouter()) setTimeout(POLL_PERIOD, onPollPeriodTimeout);
    else setTimeout(RENEWED_ONLINE_INTERVAL, onOnlineIntervalTimeout);
}
```

Validation tests

The following tables summarize the tests performed, regarding the validation of the implemented software. Table 1 num summarizes the test conditions, while table 2 num summarizes the functionality tested and the result of each test. Several scenarios were not tested for version 1, since the target functionality was not implemented yet. Additionally, v3.a stands for version 3 one-to-many update mode, while v3.b stands for version 3 point-to-point update mode.

Test	Versions tested				# of Quaxes	Scenario, key conditions
	v1	v2	v3.a	v3.b		
1	X	X	X	X	1	Sensor sampling period < Poll period
2	X	X	X	X	1	Sensor sampling period > Poll period
3	X	X	X	X	3	Various topologies
4	X	X	X	X	1	Quax reboots while in update session.
5	X	X	X	X	1	Simulation of accidental corruption of buffered image data before committing.
6		X	X	X	1	Quax reboot while overwriting the old image.
7		X	X	X	1	Simulation of accidental corruption of overwritten image data after overwriting and before rebooting to the new image.
8		X	X	X	1	Simulation of accidental corruption of info memory data.

table 1 validation tests: test conditions

Test	Functionality tested	Result
1	Initialization and completion of a code update procedure.	PC client initialized a code update session with the Quax, code download completed, and the Quax committed and rebooted to new image successfully.
2	<<	Same as test 1.
3	<<	Same as test 1. All Quaxes programmed successfully.
4	Continuation of aborted update session.	Same as test 1. After Quax failure simulation (reboot in midst of download session), the code download resumed from the point last saved in non-volatile memory.
5	Downloaded image integrity check.	The corruption was detected by the digest check at the end of the download, and the commit response sent to the client indicated the failure. Then, a new code upload procedure was re-initiated by PC client.
6	Overwrite procedure reboot resistance.	After the accidental reboot simulation, the overwrite procedure resumed from the point last saved in INFO A memory.
7	Integrity check on the overwritten image.	The corruption was detected by the by digest check by the overwrite procedure, before rebooting to the new image. Then, PC client detected the overwrite fail by means of the GET_VERSION message exchange and requested commit again.
8	Integrity check on the image, download and overwrite state stored in info memory.	Corruption detected by digest check and data discarded.

table 2 validation tests: functionality tested and test results

References

[1] http://www.prismaelectronics.eu/site/index.php?option=com_content&view=article&id=226&Itemid=652

[2] Thanos Stathopoulos, John Heidemann, and Deborah Estrin. “A Remote Code Update Mechanism for Wireless Sensor Networks”. Technical report CENS-TR-30, University of California, Los Angeles, Center for Embedded Networked Computing, November, 2003, CA, USA, 2003.

[3] P. Levis, N. Patel, D. Culler, and S. Shenker. “Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks”. In Proceedings of NSDI’04, March 2004.

[4] P. Levis, D. Gay, and D. Culler, “Active sensor networks,” in NSDI, Mar. 2005.

[5] Jonathan W. Hui and David Culler. “The dynamic behavior of a data dissemination protocol for network programming at scale”. In Proceedings of SenSys’04, Baltimore, Maryland, USA, November 2004.

[6] Sandeep S. Kulkarni and Mahesh Arumugam. “Infuse: A TDMA Based Data Dissemination Protocol for Sensor Networks”. Technical Report MSU-CSE-04-46. Dept. of Computer Science and Engineering, Michigan State University, MI (2004).

[7] Rajesh Krishna Panta, Saurabh Bagchi, and Issa M. Khalil. “Efficient wireless reprogramming through reduced bandwidth usage and opportunistic sleeping”. Ad Hoc Networks 7 (2009) 42–62. doi:10.1016/j.adhoc.2007.11.015.

[8] Rajesh Krishna Panta, Issa Khalil, and Saurabh Bagchi. “Stream: Low Overhead Wireless Reprogramming for Sensor Networks”. IEEE INFOCOM 2007 proceedings, pp. 928-936.

[9] Rajesh Panta, Saurabh Bagchi, Issa Khalil, and Luis Montestruque. “Single versus Multi-hop Wireless Reprogramming in Sensor Networks”. Tridentcom 2008 March 18–20, 2008, Innsbruck, Austria.

[10] Sandeep S. Kulkarni and Limin Wang. “MNP: Multihop Network Programming for Sensor Networks”. Proc. 25th IEEE International Conference on Distributed Computing Systems (ICDCS 2005), Columbus OH, June 2005, pp. 7-16.

[11] Sandeep S. Kulkarni and Limin Wang. “Energy-Efficient Multi-hop Reprogramming For Sensor Networks”. ACM Transactions on Sensor Networks (TOSN), Volume 5 , Issue 2, March 2009.

[12] L. A. Phillips, “Aqueduct: Robust and Efficient Code Propagation in Hetero-geneous Wireless Sensor Networks,” Master’s thesis, Univ. CO, 2005.

[13] Michele Rossi, Giovanni Zanca, Luca Stabellini, Riccardo Crepaldi, Albert F. Harris III and Michele Zorzi, “SYNAPSE: A Network Reprogramming Protocol for Wireless Sensor Networks using Fountain Codes”. Proc. IEEE SECON 2008, pp. 188-196.

[14] Krasniewski, M. D., Panta, R. K., Bagchi, S., Yang, C-L., and Chappell, W. J., “Energy-Efficient On-Demand Reprogramming of Large-Scale Sensor Networks”, ACM Transactions on Sensor Networks, Vol. 4, No. 1, Article 2, January 2008. DOI: 10.1145/1325651.1325653.

[15] Yang Yu, Loren J. Rittle, Vartika Bhandari, Jason B. LeBrun “Supporting Concurrent Applications in Wireless Sensor Networks”. SenSys '06, Proceedings of the 4th international conference on Embedded networked sensor systems.

[16] W. Li, Y. Zhang, and B. Childers, “MCP: an Energy-Efficient Code Distribution Protocol for Multi-Application WSNs”, 5th IEEE International Conference on Distributed Computing in Sensor Systems, LNCS 5516, Springer-Verlag, pages 259-272, Marina Del Rey, California, June 2009.

[17] K. Maier, A. Hessler, O. Ugus, J. Keller, D. Westhoff, “Multi-Hop Over-The-Air Reprogramming of Wireless Sensor Networks using Fuzzy Control and Fountain Codes” SOMSED'09, Self-Organising Wireless Sensor and Communication Networks Hamburg, Germany 8 - 9. October 2009.

[18] W. Dong, C. Chen, X. Liu, J. Bu, Y. Gao, “A Light-Weight and Density-Aware Reprogramming Protocol for Wireless Sensor Networks”, IEEE Transactions on Mobile Computing, Dec. 2010.

[19] Määttä, L., Suhonen, J., Laukkarinen, T., Hämäläinen, T.D. & Hännikäinen, M. 2010. “Program image dissemination protocol for low-energy multihop wireless sensor networks”. In: Nurmi, J. et al. (eds.). 2010 International Symposium on System-on-Chip Proceedings, 29-30 September 2010, Tampere, Finland Tampere. pp. 133-138.