UPPSALA
UNIVERSITET

IT 19 072
Examensarbete 15 hp
November 2019

Implementation of an Age of Information-aware Scheduler for Resource Constrained Devices

Lukas Wirne

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Implementation of an Age of Information-aware Scheduler for Resource Constrained Devices

Lukas Wirne

We examine a system where remote applications frequently request fresh information from sensor nodes through an edge and cloud based infrastructure. We study a scheduling policy that ensures fresh information while reducing energy consumption. In this work we build a testbed for evaluating the performance of the policy. We present a solution to synchronizing devices over a wireless network. We discuss the challenges of using resource constrained devices in an environment where power is limited. We run experiments to evaluate the timing of the testbed and to see how the scheduling policy performs compared to the typical periodic schedule. Our results when testing the schedule are inline with the findings in the theoretical literature. We find that good timing of the testbed requires extensive communication for adjusting the timers.

Handledare: Lorenzo Corneo
Ämnesgranskare: Per Gunningberg
Examinator: Johannes Borgström
IT 19 072
Tryckt av: Reprocentralen ITC

Acknowledgements

I would like to express my deepest gratitude to the following: PhD student Lorenzo Corneo at Uppsala University for introducing me to this project and guiding me throughout it. I could not ask for a better supervisor; Professor Per Gunningberg at Uppsala University for taking the time and effort to criticize this report into legibility.

Contents

1 Introduction

The number of devices that are connected to the internet increases rapidly, which is largely due to the Internet of Things (IoT). IoT is a collective term for when things in our everyday life are provided with sensor nodes, or computers which are then connected to the internet. This includes household appliances, machines, vehicles and much more. Some of the IoT-enabled applications rely on fresh information, such as environmental monitoring in smart cars. The freshness of information is measured with a metric called Age of Information (AoI) [1, 2]. AoI describes the freshness of information about a remote system and can be calculated with the following formula. t describes the current time and $p(t)$ is the time the data was produced.

$$AoI = t - p(t)$$

The concept of Age of Information was introduced in 2011 in [2].

IoT sensor nodes are often constrained in terms of memory, computing capacity and energy resources, because of this cloud computing [3] complements IoT well. Cloud computing can be accessed everywhere and delivers everything that IoT lacks in terms of computational power and memory. IoT applications that requires off-load storage and/or computation, is therefore often using cloud computing as well. One consequence of this is that the computations are not done in proximity to where the data is produced and the communication latency between the sensor nodes and the cloud is high.

Concepts called fog and edge computing [4] was proposed for IoT applications where low latency is required when performing safety- and time-critical operations. Edge computing is exactly what it sounds like, running calculations and processing data in the "edge of the network" [5] instead of processing the data on cloud servers.

Some sensor node, cloud and edge based infrastructures involves remote applications that are interested in fresh data produced by the sensor nodes. If each application had direct access to the sensor nodes they could request fresh information at any time. Because of the sensor nodes' limited energy resources they can only handle a limited request frequency without running out of power.

A more reliable way of providing the applications with fresh information is to let the sensor nodes use a periodic scheduling policy. Using this policy makes the sensor nodes schedule periodic updates using duty-cycling. Duty-cycling is when a system is switching between active and passive state in a cyclic manner to preserve energy. In this context, the sensor nodes will generate and send new data updates during the active state. The data will then be cached in the cloud where the applications can access the information. This approach is widely used in this type of system. The frequency of the scheduled updates dictates the energy consumption and the freshness of the data. Higher frequency will result in higher energy consumption and better AoI.

An AoI-aware scheduling policy has been proposed by Corneo, Rohner, and Gunningberg [6] as an alternative to the periodic scheduling policy. This scheduling policy is called *grouping window*. The goal for the grouping window policy is to provide the applications with fresher information while sending the same amount of updates, compared to the periodic scheduling policy. This is done by delivering updates at specific times, using the knowledge of when applications will fetch data.

A grouping window schedule holds information about when a sensor node should feed the edge device new information. To produce a grouping window schedule, each application has to provide requirements regarding the AoI. They also have to provide when they will fetch new information, in terms of a period. This information is used by the edge device to produce a grouping window schedule for a sensor node.

Good performance of the grouping window policy in regards to AoI requires that the updates are sent with precision. This can be done if the edge and sensor nodes are synchronized. The sensor nodes are not connected to the Internet and has therefore no reliable way to receive information about global time. Instead they have to rely on their Internal clocks for synchronization. These internal clocks will drift and deviate more and more from the internet global time due to temperature changes.

In this paper we present an implementation of a testbed for the grouping window scheduling and periodic scheduling. The goal for this testbed is to mimic a real example of a system where a sensor node is feeding information to an edge device according to a given schedule. We present solutions to the challenges of synchronizing the devices over a wireless network and storing data on a device with limited memory. We will evaluate how the grouping window scheduling policy compares to the periodic scheduling policy in terms of AoI. We then evaluate how well the testbed performed in terms of timing, average AoI, packet loss and error rate.

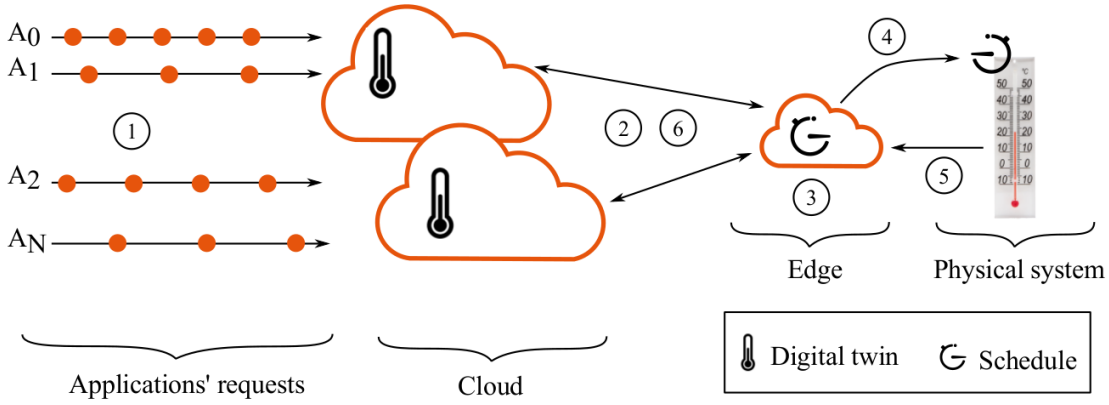

Figure 1: An overview of the system [6, p. 2477]

2 Background

2.1 System

The system in 1 of which the testbed is based on is composed of a physical system (sensor nodes), an edge device, a cloud and a number of remote applications. Applications (1) in Figure 1 has their own specific interest in the information collected by the physical system. Applications are linked with the closest cloud server and their interest are sent as a request to the cloud. Each application will fetch data periodically from the cloud. The application request specifies the period and two other values. The first is τ and describes the highest acceptable AoI. The second is ϵ and describes the delay budget, which is the maximum duration an application is willing to wait for fresh information.

Figure 2 describes how a sensor node sends updates to the cloud and how an application fetches the data from the cloud. The application uses τ=200ms, ϵ=50ms and a period of 250ms. The example assumes no communication delay between the entities so that the AoI isn't affected by the communication delay. For first fetch at 250ms the application receives data that was produced at 100ms. 250ms-100ms = 150ms. The AoI of the data of the first fetch is therefore 150ms. The second fetch at 500ms receives data produced at 400ms, which results in data with

AoI of 100ms. The response to the third fetch at 750ms is delayed by the cloud because of the applications 40ms delay budget. The cloud knows that it will receive an update in less than 50ms and can therefore provide the application with completely fresh data, that is, AoI = 0ms.

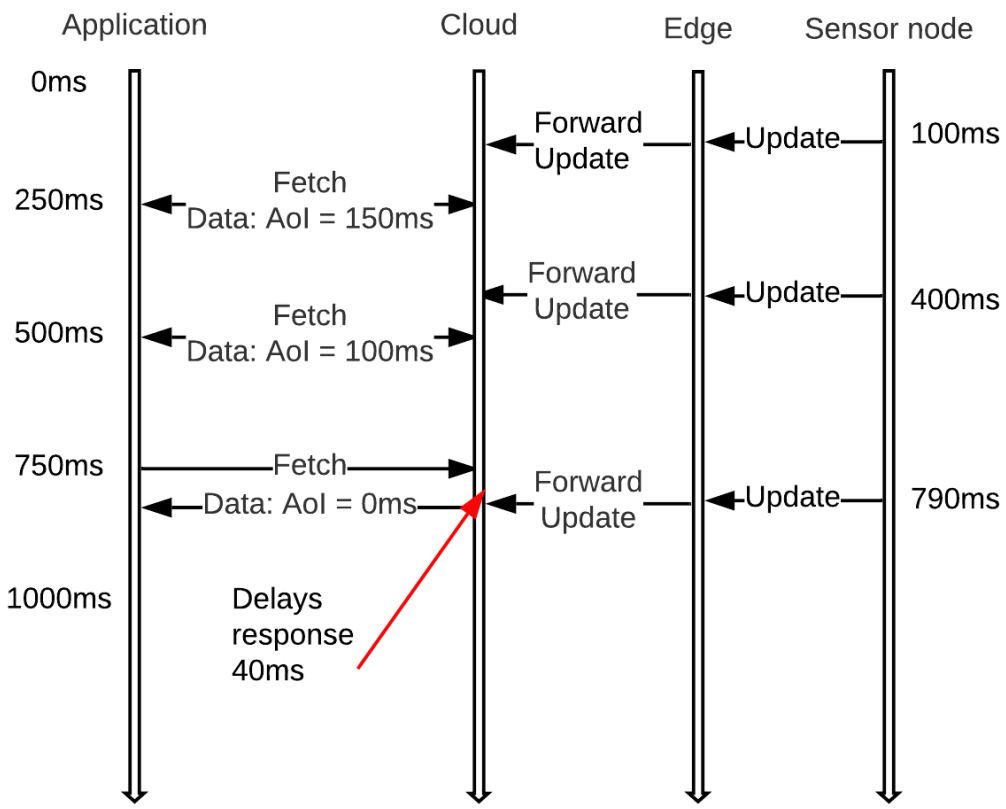

Figure 2: A sensor node sends updates to the cloud and an application fetches the data from the cloud. The. The application uses τ=200ms, ϵ=50ms and a period of 250ms. No communication delay.

The cloud will collect the requests from the applications. (2) The cloud will then forward the requests to the edge device. The cloud collects all the data that is pushed from the edge device. The data received originates from a specific sensor node, and is accessible by the applications through a digital twin [7] node. A digital twin is a digital replica of an entity. In this case there is a digital twin of each specific sensor node, represented at each cloud server. This will let the application access the data as if they had direct access to all sensor nodes, independent on which cloud server they are linked with. Each time the cloud receives data from a sensor node, all the corresponding digital twins for that sensor node are updated with the new data.

The edge is a server/gateway that connects to the sensor nodes in the physical system and to the cloud. It is placed close to physical system to reduce latency. (3) The edge will compute a schedule from the requirements for all requests aimed towards a certain sensor node, and then

send it to the sensor node (4). The information provided by the schedule lets the sensor node know when it should push fresh information to the edge.

(5) The physical system is a set of sensor nodes, each connected to the edge. The sensor nodes will feed the edge with fresh data at specific times. The specific times are given by the schedule which is compiled and sent by the edge. The data produced by the sensor node is distributed from the sensor node to the edge device, and then to all cloud servers that requested the data (6).

2.2 Scheduling policies

In the type of system which is described in this section a periodic scheduling policy is widely used. A periodic schedule specifies a period and the number of updates that should be sent. A sensor node using a periodic schedule will send new information after each time period. The other scheduling policy is called grouping window. The grouping window schedule specifies timestamps when a sensor should send new information. A grouping window schedule is calculated using the application requests described in section 2.1. The algorithm used to calculate these timestamps is described in appendix A. This is done by providing cached data to the cloud whenever the AoI requirements can be met. In other words, there is no need to read valid data from the sensor node when a cached value can be used.

2.3 Testbed

The testbed is based on the system described in section 2.1, but is limited to the edge device and one sensor node. The goal for the testbed is to mimic a real example of a system where a sensor node is feeding information to an edge device according to a given schedule. A schedule is given to the edge as a file, (1) in figure 3. The file specifies either a grouping window schedule or a periodic schedule. The edge device sends the schedule to the sensor node (2). After the sensor node receives the schedule it starts to send data to the edge according to the schedule (3). The edge device will measure the arrival time, error rate and packet loss for each update and log the measurements (4).

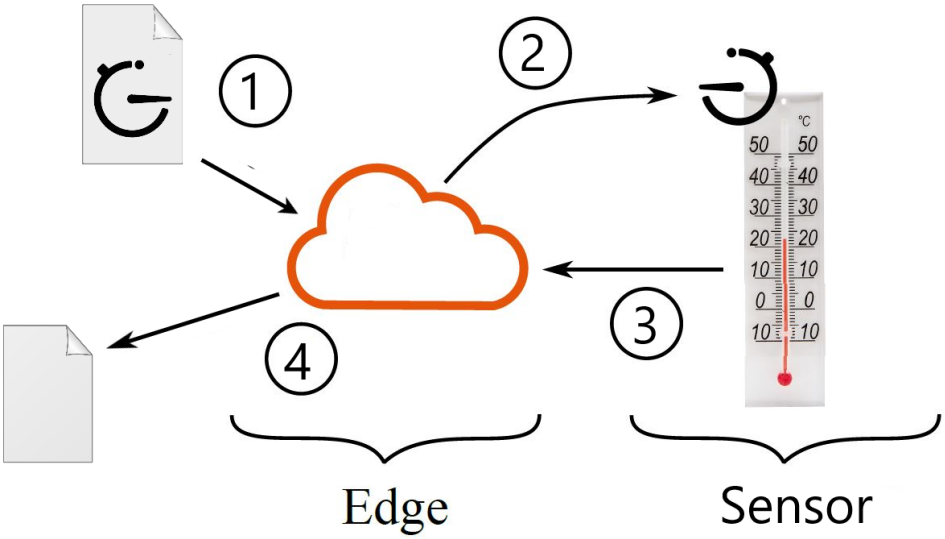

Figure 3: Testbed [6, p. 2477]

2.4 Hardware

The sensor node is implemented using a Texas Instrument MSP430fr5969 Launchpad [8] with a Texas Instrument CC2500 2.4GHz radio interface [9]. The edge consists of a BeagleBone Black [10] mounting the same radio interface as the sensor. The CC2500 is connected using the USART interface of each device with jumper cables.

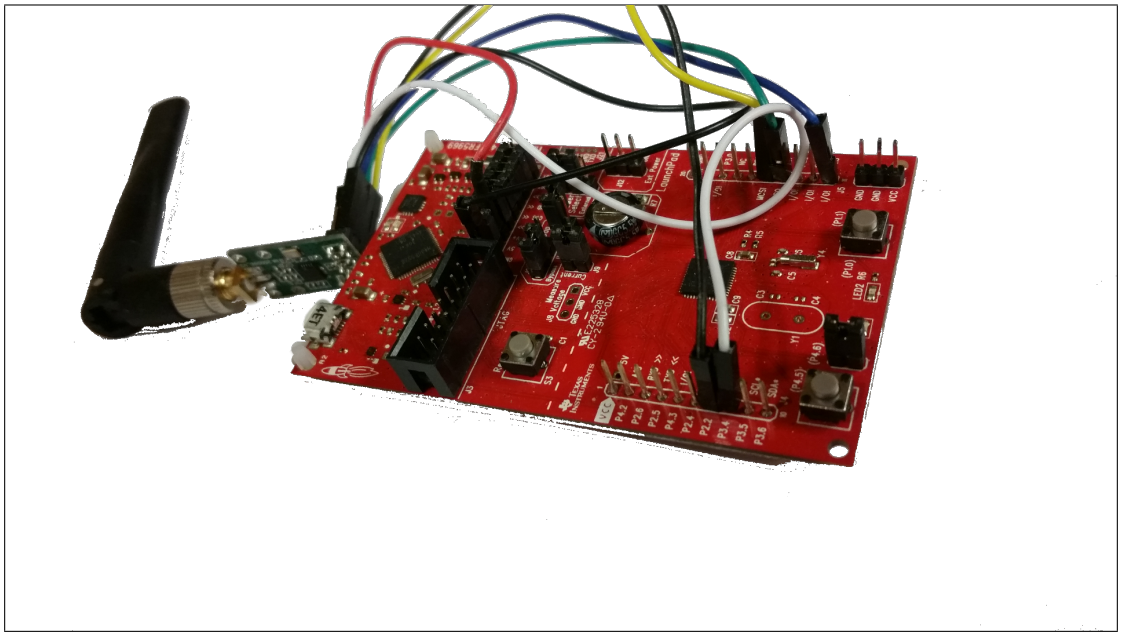

Figure 4: A photo of the MSP430fr5969 Launchpad

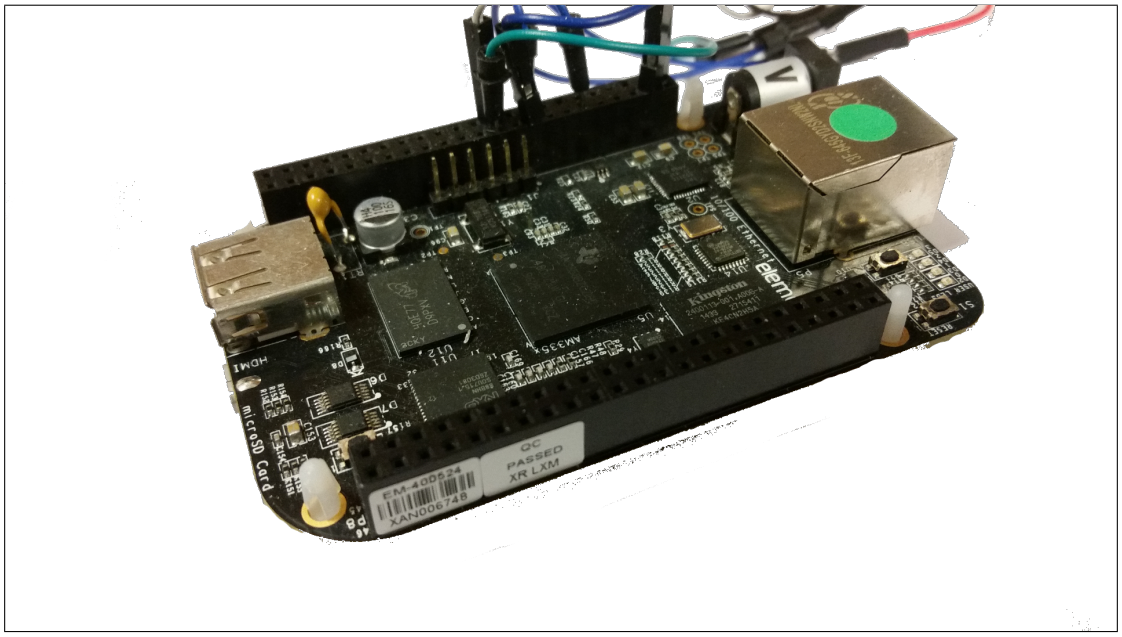

Figure 5: A photo of the BeagleBone Black

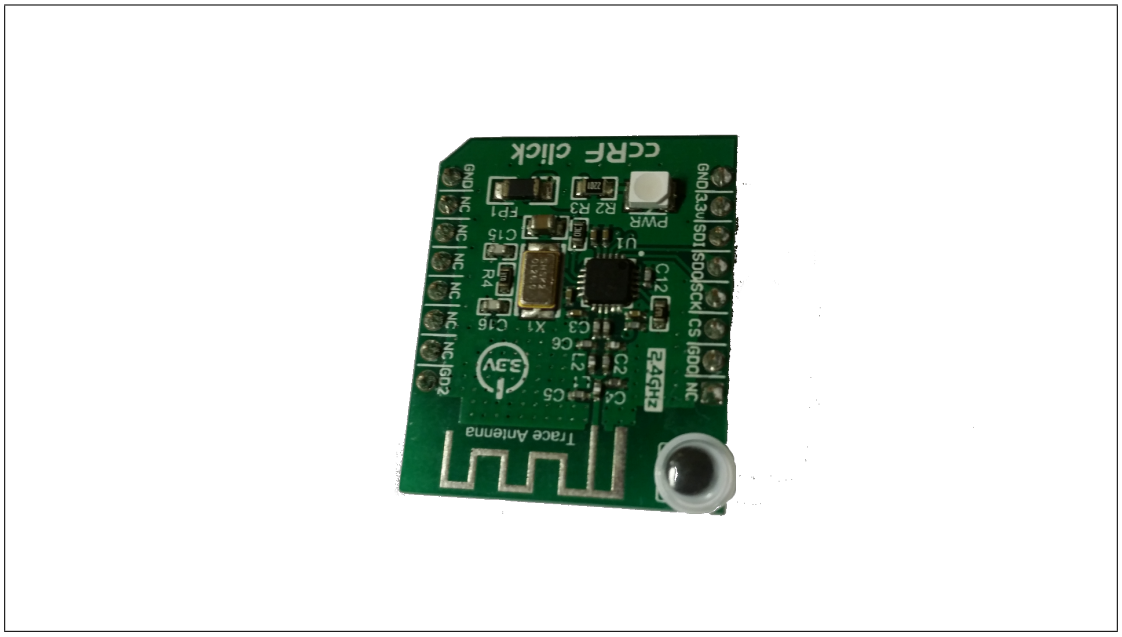

Figure 6: A photo of CC2500 RF transceiver

3 Challenges

3.1 Clock synchronization

The sensor node should send updates to the edge device according to the schedule generated by the edge device. Since the sensor node is not connected to the internet, it has no reliable way to receive information about global time (provided by the NTP protocol). Instead it has to rely on internal clocks which will drift and deviate more and more from the internet global time. The edge device has 1GHz crystal, the sensor node has a 16Mhz crystal. For the edge device to measure the arrival time of each update, and for the sensor node to know when to send the update, the edge and the sensor node timers has to be synchronized. The frequency of the internal crystals of the devices are prone to change with temperature. The sensor node internal crystal will change $0.01\%/°C$ [8, p. 29]. This translates to 300ms difference over 60s between running the sensor node at $25°C$ compared to $75°C$. This becomes a problem when using the grouping window policy. The time difference has to be compensated for.

3.2 Memory usage

A periodic schedule doesn't require much memory space, since the next time to send an update can be calculated locally using the period given by the edge device. The grouping window schedule, otoh, is a set of timestamps which all have to be stored in the sensor. This can be done in intervals or all at once. The sensor node has two different memory types, SRAM and FRAM [11]. SRAM and FRAM have different characteristics and limitations. Where we store the schedule may therefore limit our design choices. SRAM can be accessed faster than FRAM, but has a memory limit of 2kB. The access time to FRAM is longer (126ns per WORD(8 bit))[11, p. 3], but it stores up to 64KB of data. The data representation of each timestamp will decide

how large schedules that may fit into either memory of the sensor node. The schedule could be split up into smaller chunks, and the edge could push each chunk one by one, and have the sensor node send updates in between. This is only an option if the schedule can be split into small enough chunks such that the sending of each chunk will be finished before the next update is sent.

3.3 Energy consumption

Energy consumption should be minimized at the sensor node-side. There are several factors that has an effect on the energy consumption of the sensor node. The CC2500 which the devices are using to transmit and receive data will draw more current while in active state [8, p. 7–8]. Keeping it in an idle state will reduce the power consumption, but will prevent the sensor node to send or receive data. Duty-cycling between idle and active state will minimize energy consumption. The sensor node has five different low power mode settings. Writing software that utilizes these low power modes by duty-cycling will decrease the energy consumption greatly.

4 Design

4.1 Data representation

The data representation of a schedule will have an impact on the schedule size, which is the number of bits that is required to represent it. Reducing the size will reduce the energy consumption of transferring the schedule from the edge to the sensor node, and will increase how many timestamps the sensor node can hold at a time. Time information like timestamps and periods are measured in milliseconds. A grouping window schedule will be represented as a list of timestamps. Instead of letting each timestamp be relative to time zero, we let each timestamp be relative to the previous one, except for the first timestamp that is relative to zero. Let S_i be the ith timestamp of schedule S of size n. We can calculate the time t_i, which when the ith update should be sent by applying this formula.

$$t_i = \sum_{k=0}^{i} S_k$$

Representing the timestamps in this manner will lower the number of bits that represent each timestamp. In this system all periods and timestamps are represented as 16-bit unsigned integers. Unsigned since timestamps and periods are all non-negative, and a 16-bit representation can hold values between 0 and 65,536, which should be more than enough for this application. A grouping window schedule with 2,000 timestamps of 16-bit each results in a size of $2000 * 16/8 = 4KB$. The schedule will not fit into the 2kB of SRAM available in the sensor node, but will most probably fit into the 64kB of FRAM. The timestamps could possibly be represented with fewer bits, but not to the extent that it would fit into the SRAM.

4.2 Synchronize clocks

If the edge and sensor node start their timers at the same moment, the transmission time of the update will cause all updates to be delayed. Figure 7 shows how this can be adjusted right from the start. We let the sensor node start its timer t2 when all schedule information from the edge has been successfully received. After this the sensor node sends an acknowledgement message to the edge device to give a notification that the communication was successful and that its time to

start timer t1. If the communication went alright and if the end-to-end communication latency X for an update is the same as for an acknowledgement Y, the timers should be synchronized. This is all the synchronization that is required for running the testbed with a periodic schedule.

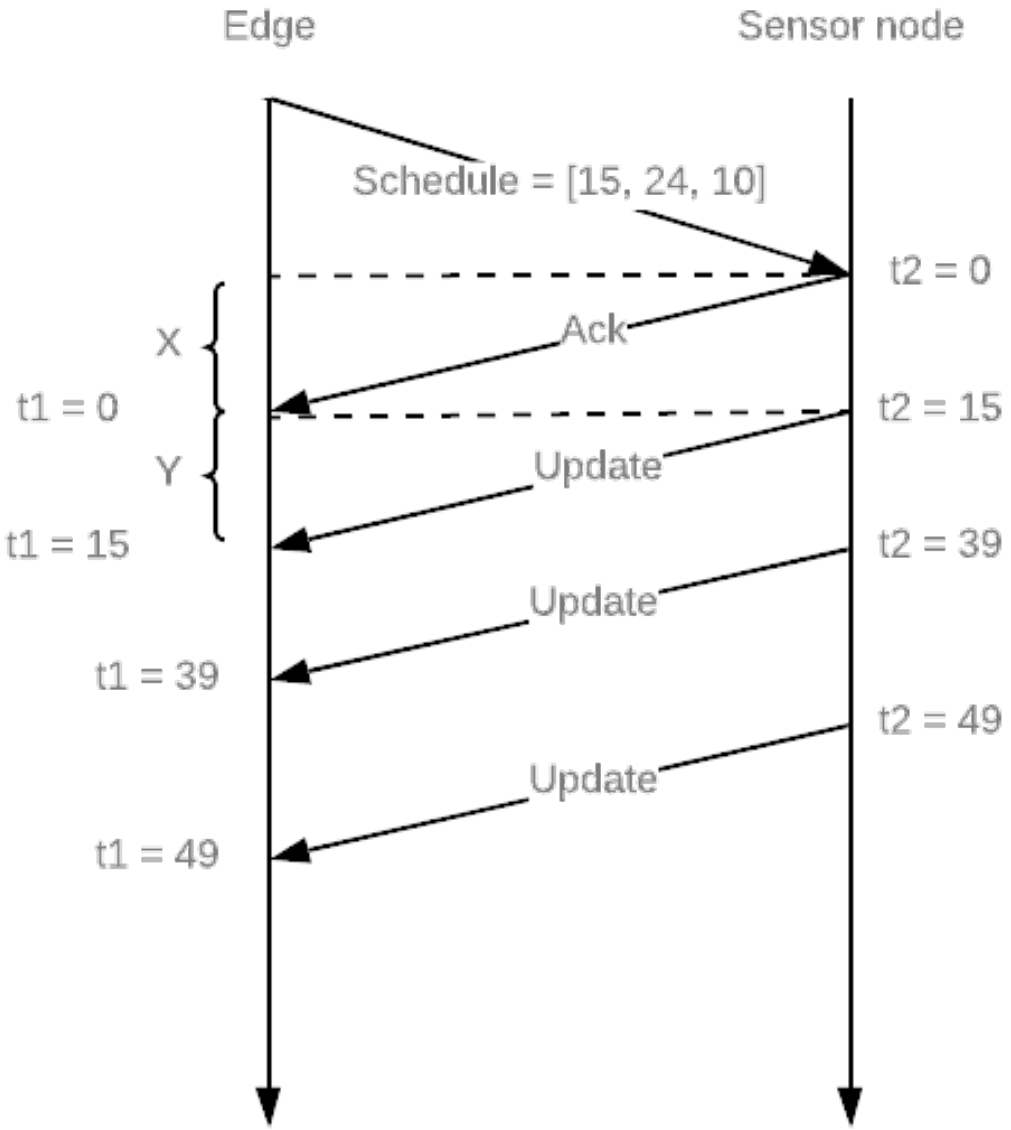

Figure 7: Edge device timer t1 is synchronized with the sensor node timer t2

But for grouping window schedules we still have to deal with drift caused by temperature. The solution to this is a correction packet, how it's used as is shown in Figure 8. This packet contains the value δ, which is the difference between expected arrival time and actual arrival

time of an update received by the edge. In 8 we see that the sensor node timer t2 is drifting. The edge expects update packets at 15, 39 and 69. The first update is 1ms late and the second is 3ms late. A correction packet is sent with $\delta = 3$ms after the second update is received. The sensor node corrects timer t2 according to the value of δ. The third update packet arrives on time. This raises three other questions: When should this packet be sent, how often, and how do the edge and sensor node agree on when to send and receive?

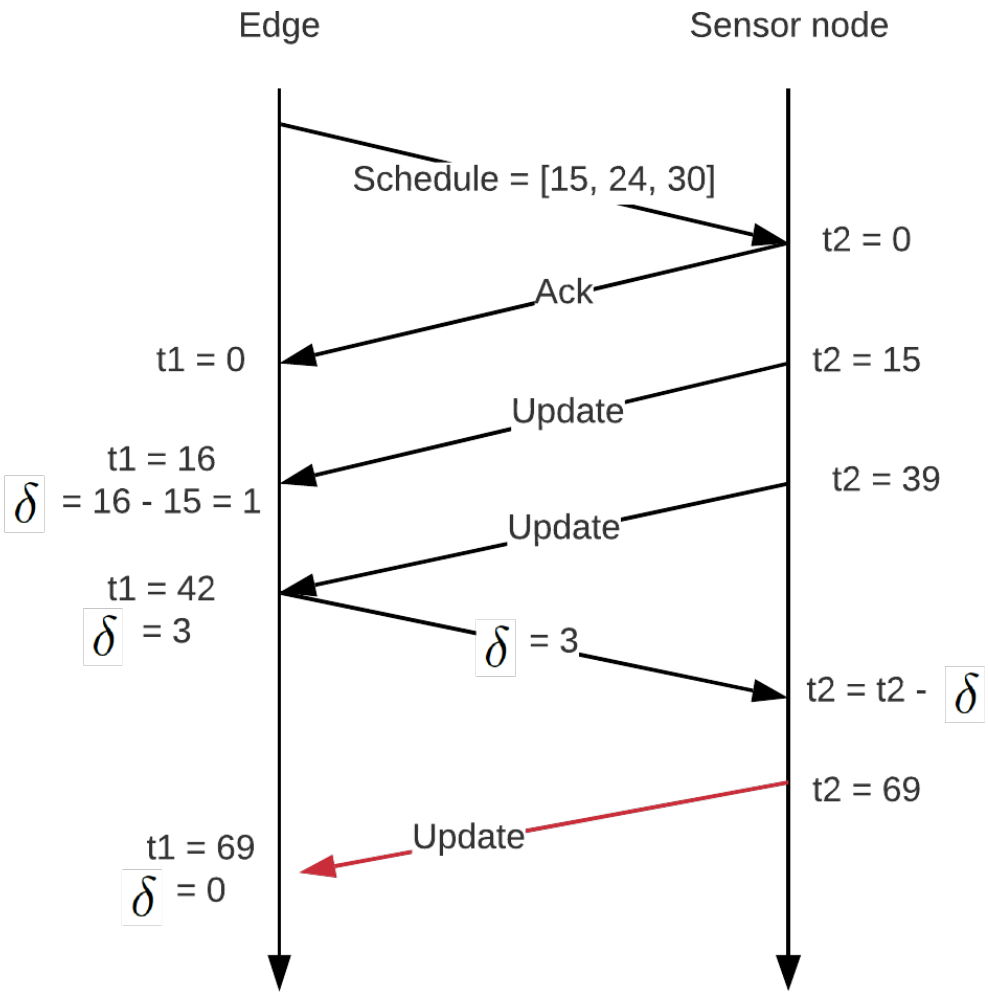

Figure 8: The drift between edge device timer t1 and sensor node timer t2 is adjusted for using a correction packet

The δ caused by drift gets worse as time passes. If a correction packet is sent between every

data message from the sensor node, the timer synchronization will be optimized. This is a bad idea, since the communication overhead will make the energy consumption rise tremendously, and sending packets that frequent has a higher risk of causing delays to the update packets. What if we send correction packets periodically? This would be better for the energy consumption of the sensor node, but the problem with causing delays still persists. Figure 9 shows how the sensor node is scheduled to send an update at 69ms, but is waiting for the arrival of a correction packet. Because of the late arrival of the correction packet, the third update packet is never sent by the sensor node, and the edge device will see that as a packet loss. Let S_i and S_{i+1} be two adjacent timestamps in schedule S. Let t_U be the transmission time for an update and t_C be the transmission time for a correction. Then sending a correction packet between S_{i+1} and S_{i+1} will cause a delay if

$$S_{i+1} < t_U + t_C$$

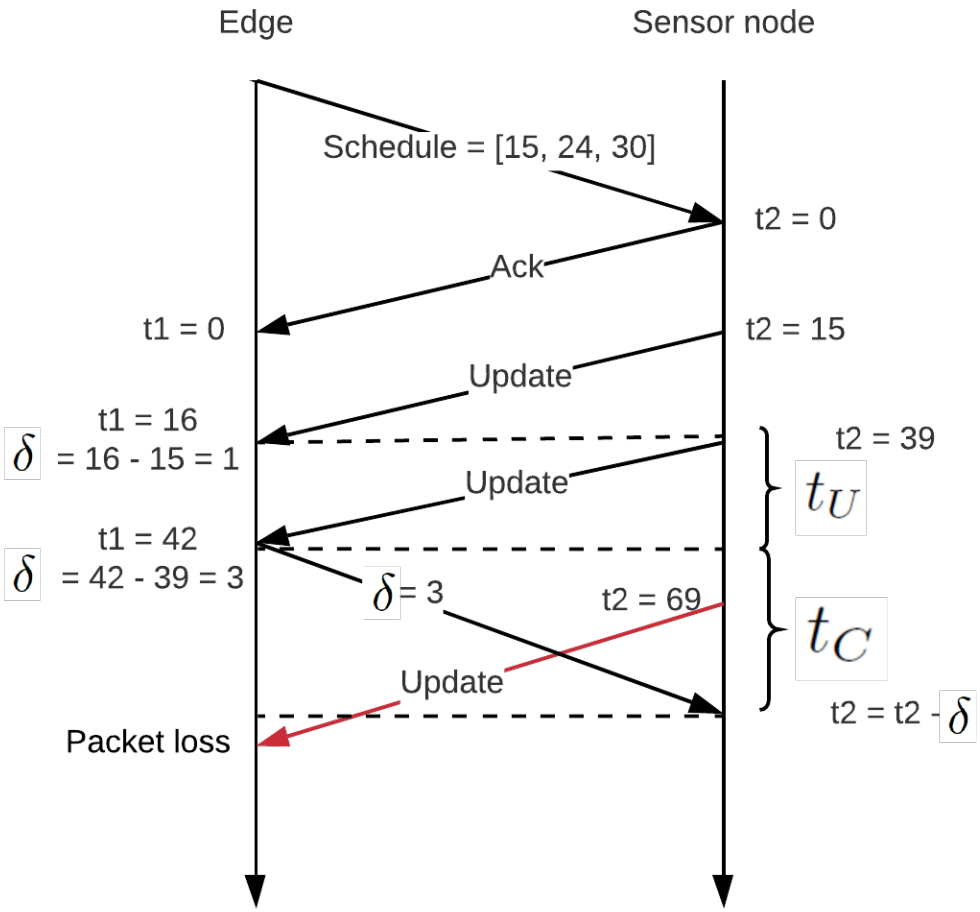

Figure 9: Sensor node waiting for a correction packet such that the third update packet is not sent

We will from now on let $\alpha = t_U + t_C$. Knowing α lets us search the timestamps in a schedule to find the ones where a correction packet could be sent and received, and not cause the following updates to be delayed. Algorithm 1 will find the most suitable timestamps of a schedule and return their indices. A correction packet is sent by the edge when the index of the next expected update packet is one of these indices. This list of indices is generated by the edge and sent to the sensor node, this way both devices know when they should communicate. Algorithm 1 finds indices such that a correction packet should be sent at least once every β packet, but with a maximum of twice every β packet. If none of the following β timestamps are suitable during the

run of the algorithm, it chooses the last index of the next β timestamps. This will cause the sensor node to miss at least the next scheduled update packet. Instead of sending the update(s) late, the sensor node drops these packets. This will be seen as packet loss at the edge device when collecting data. Generating indices from a large schedule with a low β will generate many indices. The number of indices are limited by the SRAM of the sensor node since they have to be stored somewhere. They cannot be stored in FRAM because that's where the schedule is stored.

Algorithm 1 Finds suitable indices for correction packets with help of β and α, for a given schedule S.

```
S, list of timestamps
i ← 0
R ← ∅
c ← 0
best ← None


while i < |S| do
    c ← c + 1
    if S_i >= α then
        best ← i
    if c == β then
        if best == None then
            best ← i
        R ← R⋃{best}
        best ← None
        c ← 0
    i ← i + 1
return R
```

4.3 Tracking time

We use duty-cycling to save power between each scheduled update transmission. An interrupt from the timer would wake up the sensor node when it's time to send the next update packet. Resetting the timer to zero after each interrupt and adjusting the interrupt threshold would wake up the sensor node correct for the first interrupt, but would cause drift after some iterations. Resetting the timer and adjusting it according to the next scheduled update requires some CPU clock ticks. If each adjustment would require the same clock ticks, this could be compensated for, but that isn't the case since the the timestamps in a schedule could be of different values.

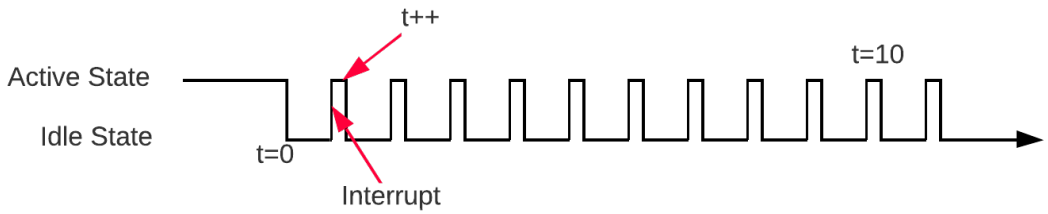

Figure 10: Sensor node switching between idle and active state using hardware interrupts each millisecond. When an interrupt occurs, the node wakes up and increments counter t by one.

We avoid adjusting the timer in the sensor node by letting a timer count to 16000, which is equivalent to 1ms when using the 16Mhz crystal. Each time the timer is triggered, a millisecond counter is incremented by one. This millisecond counter will be used to track the current time of the system. When a correction packet is received, this counter is adjusted accordingly to the information received. Figure 10 shows how a hardware interrupt each millisecond causing the sensor node to go wake up and increment the software counter.

4.4 Packets

Here follows information about the packets sent between the edge and the sensor node. The CC2500 can be configured with three different packet lengths settings: Fixed length, variable and infinite length setting [9, p. 30–31]. Fixed and variable length supports packets up to 255 bytes, where the infinite packet setting supports packets of unlimited length. This system uses the variable and the infinite packet length settings. This is because the information that has to be shared between the devices is not uniform in size. Fixed length could still be used, but would require short packets to have large overhead and/or large packets to be split into smaller chunks. Both cases will result in more data transmission, which is not desirable for this application.

4.4.1 Edge to Sensor node

There are five different packet types that are sent from the edge to the sensor node: *period*, *intermediate*, *final*, *correction IDs* (also referred to as *IDs*) and *correction* packet. The infinite packet length setting was chosen for all packet types sent from the edge to the sensor node, the motivation behind this is that the size of a schedule is not limited to 255 bytes, and the possibility to send a larger sized schedule in one packet may be desirable. The infinite packet length setting also has it limits when run in an environment with limited memory. The sensor node has 2kB of SRAM and all bytes are not available to use as a receiver buffer, since the 2kB is also used as stack for example. The size of the receiver buffer of the sensor node is adjustable. It's also more complex to implement support for infinite packets compared to the other settings. All packets sent from the edge device have three standard information fields. Two bytes describes the length of the packet, one byte describes the destination address and one type byte describes how the meta data field and the payload should be interpreted. The infinite packet field representation is shown in Table 1. How the dynamic fields are represented for the different packet types is shown in table 2. The Period packet type is used to let the sensor node know that it should send update packets periodically, with values such as the period and the number of updates the edge expects to receive. Intermediate and final packets are used to send grouping window schedule information. If a schedule is larger than the receiver buffer, the schedule is split into chunks of

the maximum supported size and sent as intermediate packets. The final packet is sent after the intermediate packets holding what's left of the schedule. When the sensor node receives a final packet, it is notified that this is the last packet holding schedule information. The IDs packet holds all the indices that is generated by the algorithm in Figure 1. The correction packet will be sent with the δ of the last received packet, and the sensor node will correct its' timer accordingly to the diff.

Information	Length	Destination	Meta data	Type	Payload
Bytes	2	1	2	1	2 * x

Table 1: Field representation for an infinite packet. x depends on the length of the payload.

Type	Meta data		Payload	
	Information	Value	Information	Value
Period	Number of periods	0 - 65535	Period	(0 - 65535) * 1
Intermediate	Sequence number	0 - 65535	Timestamps	(0 - 65535) * x
Final	-	-	Timestamps	(0 - 65535) * x
IDs	-	-	Indices	(0 - 65535) * x
DC	-	-	δ	(32768 to 32767) * 1

Table 2: Field representation for the different types of packets. x is the number of timestamps.

4.4.2 Sensor node to Edge

There are two different packets that are sent from the sensor node to the edge device, an *acknowledgement*(sometimes referred to as *ack*) packet and an *update* packet. These two packets are sent with the variable packet length setting in the CC2500 because both packets are under 255 bytes long.

All packets sent from the sensor node have three standard information fields. One byte describes the length of the packet, one byte describes the destination address of the packet, one type byte describes how the the payload should be interpreted. The Ack packet is used to let the edge know when information was successfully received. The field representation can be found in Table 3. The update packet is sent according to the schedule, the *id* field holds the id of the update packet. The first update packet sent always have id zero, and the index is incremented by one for every update packet scheduled to be sent. This helps the edge device to track lost packets and find the expected arrival time for each packet. The Field representation for an update packed is shown in Table 4.

Information	Length	Destination	Type	Source
Bytes	1	1	1	1

Table 3: Field representation for an acknowledgement packet.

Information	Length	Destination	Type	Source	ID
Bytes	1	1	1	1	2

Table 4: Field representation for an update packet.

5 Evaluation

The evaluation is based on two experiments. The first experiment will give us data that can be compared to results presented in [6]. It will also give us data about packet loss and error rate which will be analyzed and compared to the expected values of the CC2500. The second experiment will run the testbed using a grouping window schedule with correction packets generated with different configurations, so we can evaluate how the different configurations affects the timing of the update packets.

The CC2500 of both devices are configured for both experiments to transmit with a symbol rate of 2.9kBaud. This used with 2-FSK signal modulation results in a byte rate of 5.9kb/s. Before running the experiments we needed a suitable value for α in algorithm 1. To find this value, we tracked the time it took to for the sensor node to send an update packet and for the edge to respond with a correction packet. We did this 1000 times and the results each time was 79ms which is now used as α.

5.1 Experiment 1

We evaluate the performance of 25 applications with periodicity randomly selected in the interval [200, 400] ms. The experiment consists of pushing the schedule from the edge device to the sensor node, and then let the sensor node deliver updates back to the edge device according to the schedule. After this we extract the AoI and the Δ AoI from the arrival time measurements the edge collected during the run. We also measured the packet loss percentage and error rate. An error occurs when a corrupt packet is received. The experiment was done for both periodic schedules and grouping window schedules generated with different values of τ and ϵ. For the distribution of the correction packets we used $\beta = 300$ for all grouping window schedules.

A grouping window schedule, π_D, is from now on described as a combination of τ and ϵ, written as $\pi_D(\tau, \epsilon)$. A periodic schedule will be described as $\pi_P(\rho)$, where ρ is the period of the schedule. Six different π_D schedules was generated from the application requests, $\pi_D(60, 0)$, $\pi_D(60, 10)$, $\pi_D(60, 20)$, $\pi_D(100, 0)$, $\pi_D(100, 10)$ and $\pi_D(100, 20)$. π_D schedules where $\tau = 60$ was generated using 4450 data points from the application, where π_D schedules with $\tau = 100$ was generated from 3000. To make the π_D schedules comparable to π_P schedules, we will decide the value of ρ for the π_P schedule depending on the average of all timestamps in each π_D schedule. The average timestamp value in the $\pi_D(60, \epsilon)$ schedules equals 67ms, and $\pi_D(100, \epsilon)$ we got 101ms. Therefore only two periodic schedules had to be run for this experiment, $\pi_P(67)$ and $\pi_P(101)$. The $\pi_D(60, \epsilon)$ and the $\pi_P(67)$ schedules will be called test group 1 and $\pi_D(100, \epsilon)$ and $\pi_P(101)$ schedules are in test group 2. We only run the experiment once for each of the schedules.

5.1.1 Age of information

In Figure 11 and Figure 12 we see the evolution of the average age of information, Δ_{AoI}, for different values of ϵ. Δ_{AoI} is lower for the π_D schedules in both groups, independent of the values of ϵ. The difference in Δ_{AoI} grows bigger as ϵ goes up. The biggest difference in Δ_{AoI} is for

$\pi_D(60, 20)$ where we see a 60% decrease. When $\epsilon = 0$ the π_D schedules still gives better Δ_{AoI} than the periodic schedules, but only slightly. This result is in line with the conclusions in [6].

Figure 13, 14, 15, 16, 17 and 18 describes the distribution of AoI for the different schedules. The distribution is close to uniform for the periodic schedules, ranging between 0 and ρ. With the π_D schedules we can see that the likelihood of the AoI being zero increases as the ϵ goes up. This result is also similar to the results in the theoretic literature. Another thing to notice is that each of the π_D schedules except for $\pi_D(100, 10)$ have a small portion of AoI with a much larger value. The reason behind this could be either because of packet corruption or dropped update packet(s) due to the sensor node waiting for a correction packet. In this case the AoI for each application request will map to the next update packet that is successfully received.

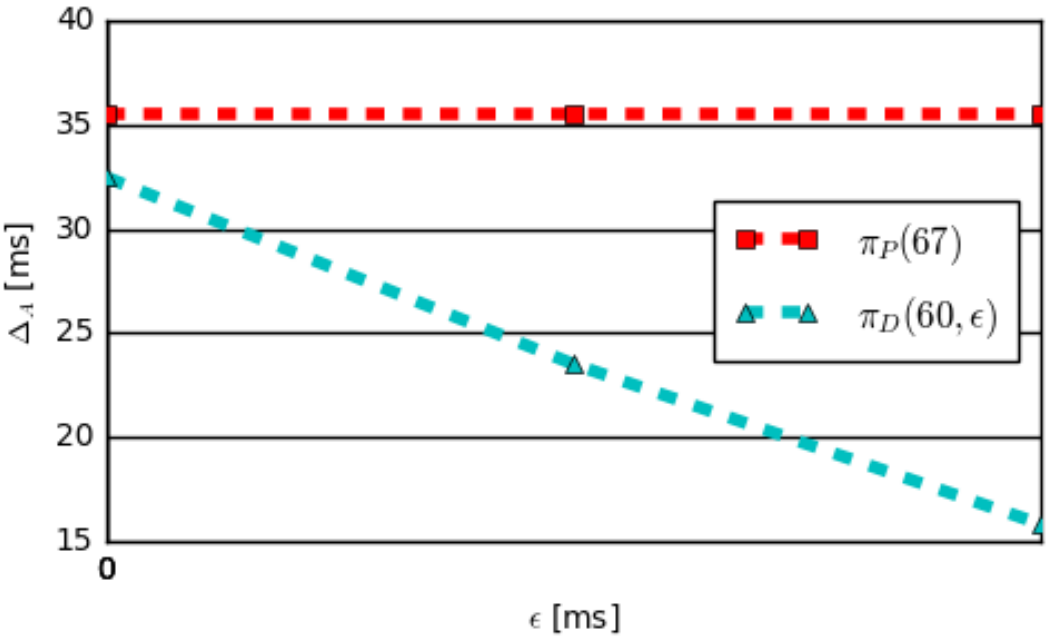

Figure 11: A chart describing the relationship between Δ_{AoI} and ϵ for test group 1.

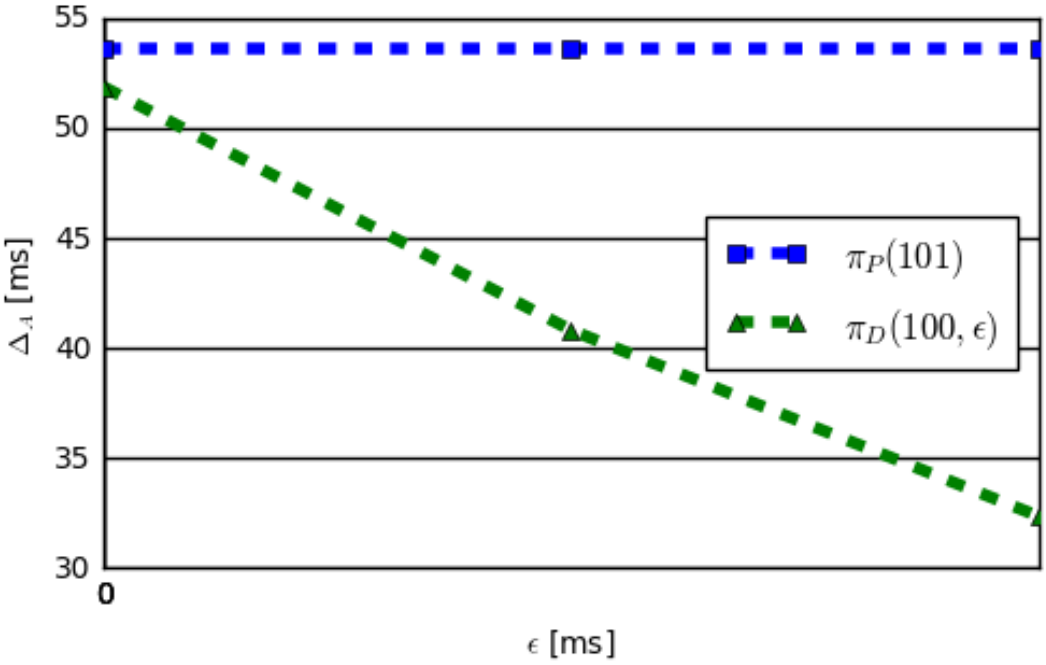

Figure 12: A chart describing the relationship between Δ_{AoI} and ϵ for test group 2.

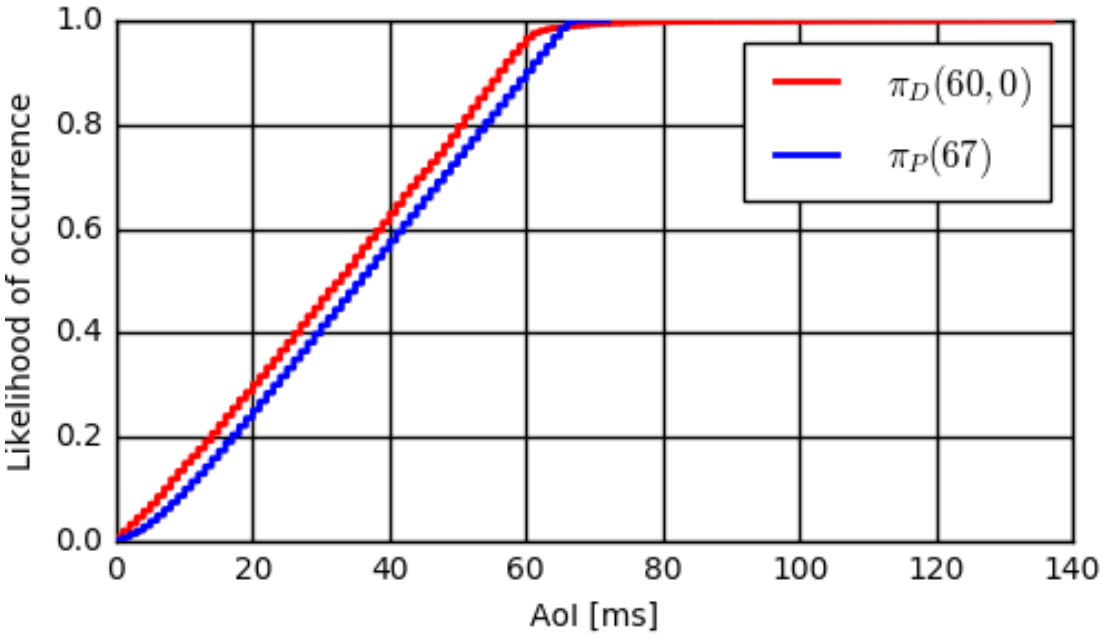

Figure 13: CDF comparison between a periodic schedule π_P and a grouping window schedule π_D on AoI.

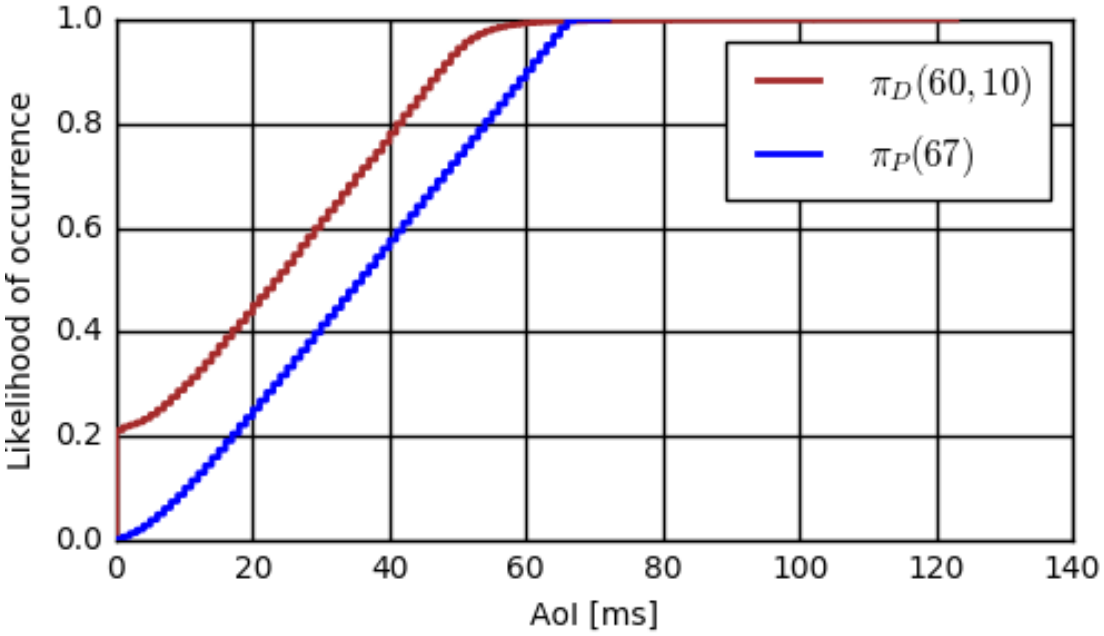

Figure 14: CDF comparison between a periodic schedule π_P and a grouping window schedule π_D on AoI.

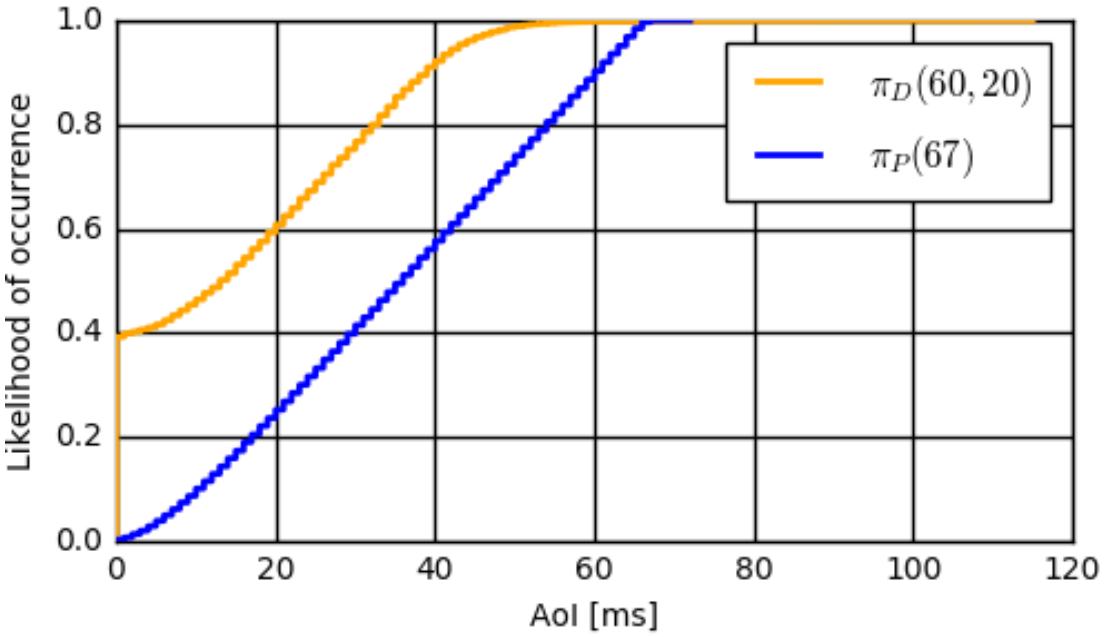

Figure 15: CDF comparison between a periodic schedule π_P and a grouping window schedule π_D on AoI.

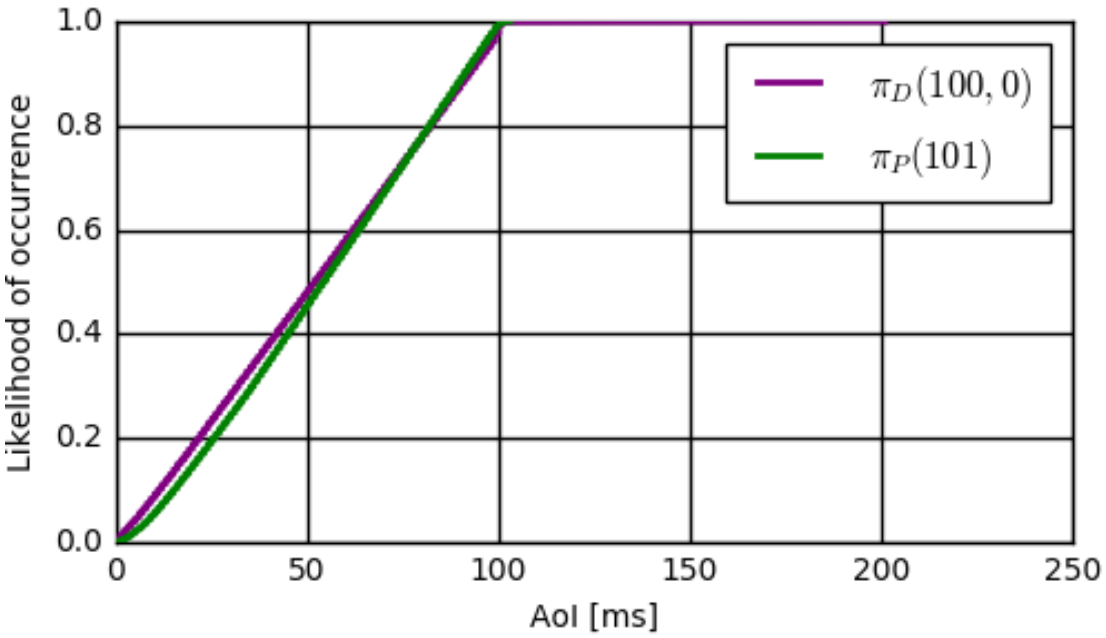

Figure 16: CDF comparison between a periodic schedule π_P and a grouping window schedule π_D on AoI.

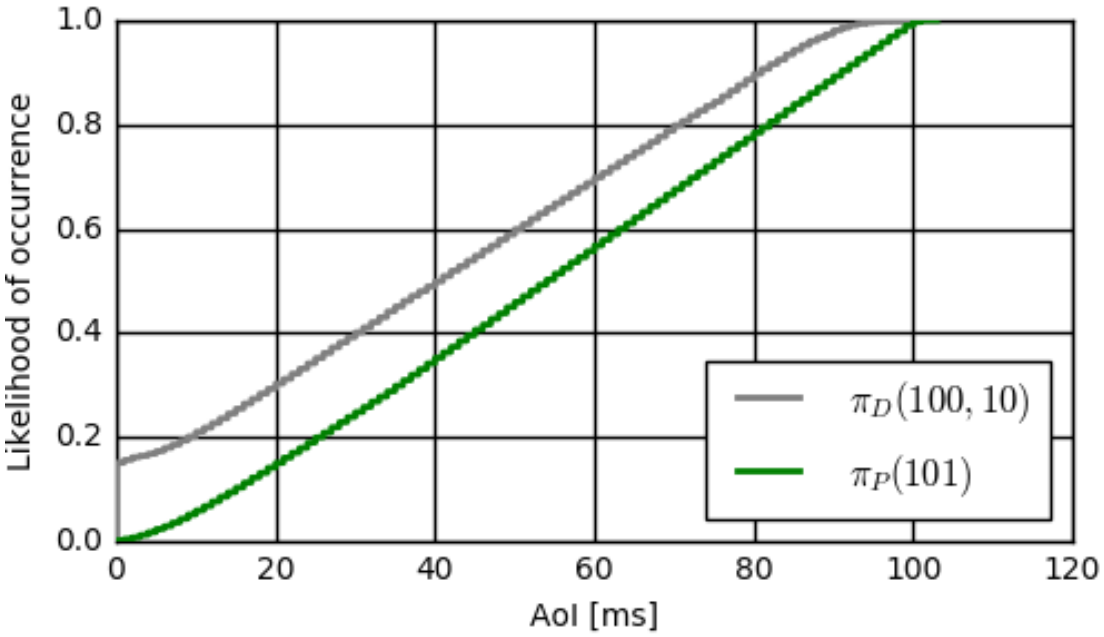

Figure 17: CDF comparison between a periodic schedule π_P and a grouping window schedule π_D on AoI.

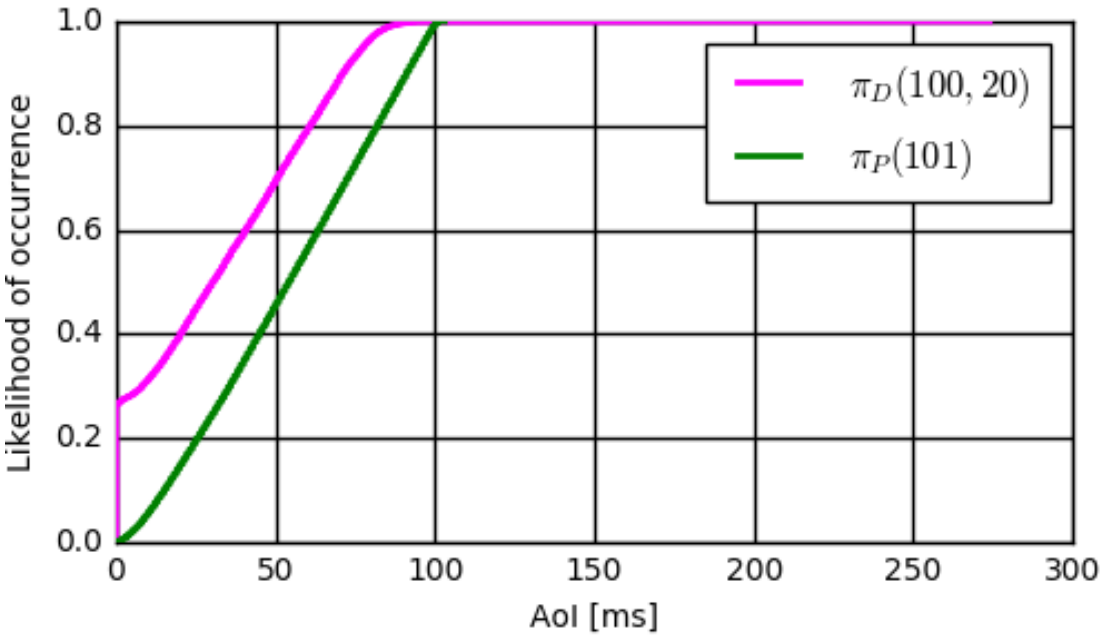

Figure 18: CDF comparison between a periodic schedule π_P and a grouping window schedule π_D on AoI.

5.1.2 Error rate and packet loss

Table 5 and Table 6 show us the error rate and the loss percentage for test group 1 and test group 2. As we can see, there's no correlation whatsoever between error rate and the different type of schedules. The expected error rate for the CC2500 is 1% [9, p. 1], which is over three time higher than the highest error rate in both tables, which is for the periodic schedule in test group 2. The packet loss rate on the other hand only occurs for the π_D schedules in test group 1. Packet loss can occur because of bad signal strength, but this packet loss has another explanation. correction packets is only used for grouping window schedules. A correction packet may be scheduled between two timestamps where the inter-delay is less than what's required to not cause a delay. In this case a correction packet sent between two timestamps with a inter-delay of less than α(79ms) will cause the next update packet to be dropped. As the average inter-delay goes up when τ increases, we expect less packet losses. Also, a higher ϵ will group more application timestamps together when generating a π_D schedule, which will leave more availability for algorithm 1 to find timestamps which is above the 79ms threshold.

	$\pi_D(60,0)$	$\pi_D(60,10)$	$\pi_D(60,20)$	$\pi_P(67)$
Loss [%]	0.27	0.23	0.11	0.00
Error [%]	0.07	0.02	0.07	0.09

Table 5: Packet loss percentage and packet error percentage for group 1.

	$\pi_D(100,0)$	$\pi_D(100,10)$	$\pi_D(100,20)$	$\pi_P(101)$
Loss [%]	0.00	0.00	0.00	0.00
Error [%]	0.20	0.20	0.17	0.30

Table 6: Packet loss percentage and packet error percentage for group 2.

5.2 Experiment 2

Another experiment was done to see how the use of correction packets will affect the δ of the updates fed by the sensor node. This time we let the sensor node send 3000 update packets periodically with a period of 100ms. Since 100ms is larger than the α we used (79ms), none of the correction packets will cause any packet loss. We ran this experiment with with three different values of β, 10, 100 and 1000. Figure 19, 20 and 21 shows the occurrence of each δ for each β. An ideal result would be zero δ for all packets. Closest to ideal result was gotten when using $\beta = 10$. Here the δ ranges between [-2, 2] and over 50% arrived with 0ms diff. Using $\beta = 100$ resulted in the δ range of [-1,7], and only a bit less than 10% of the packets to arrive perfectly. The δ distribution of when using $\beta = 100$ is the widest, ranging between [0, 42]. Only 20 packets out of the 3000 sent arrived with a 0ms diff. Using $\beta = 10$ gives the best result, but also the largest number of extra receptions of correction packets for the sensor node. $\beta = 10$ will cause an extra of 300 receptions, compared to $\beta = 100$ which would cause 30 extra, and $\beta = 1000$ with only 3 extra receptions.

Sending correction packets more frequent (smaller β) results in a lower δ. For the experiment with $\beta = 10$, one correction packet is sent at least every 10th update which means that the timers are synchronized at least once every second (10 update packets 100ms apart). $\beta = 1000$ results in timer synchronization every 100 seconds. The sensor node timer will deviate more after 100 seconds without synchronization than after one seconds without.

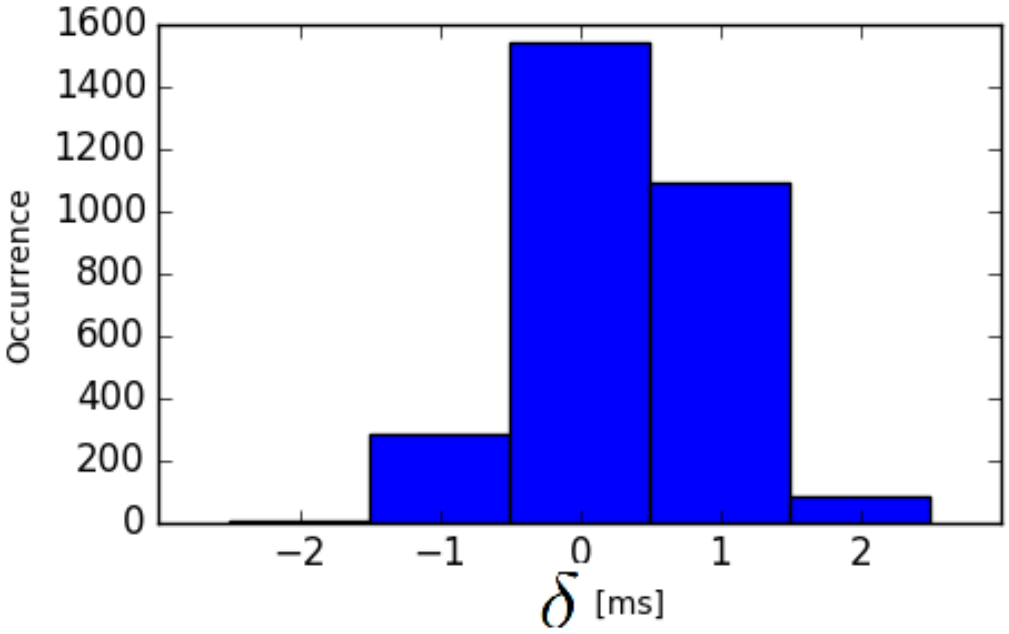

Figure 19: δ distribution for 3000 update packets sent periodically every 100ms, when using $\beta = 10$.

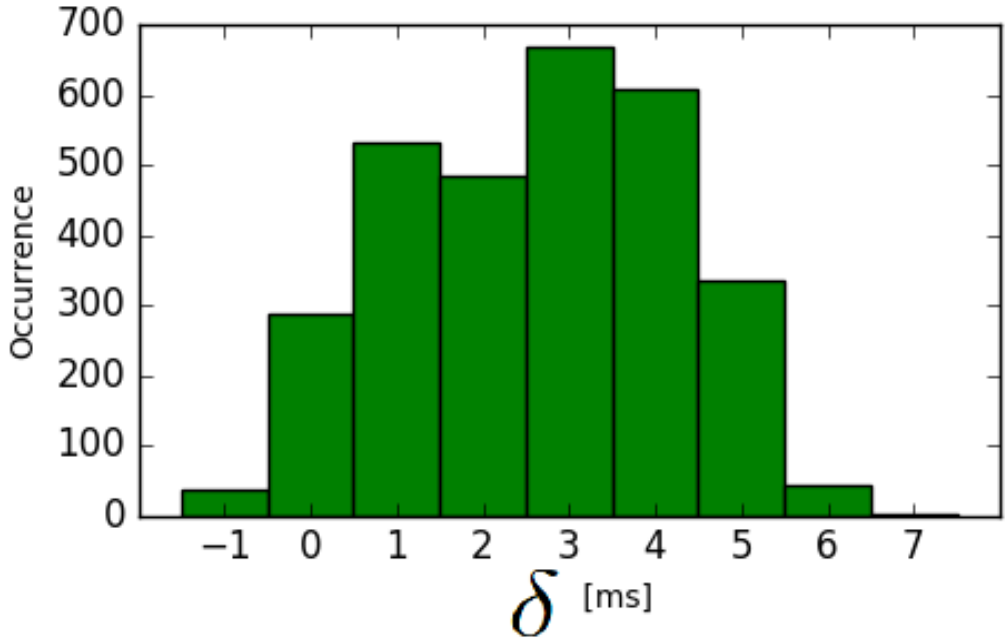

Figure 20: δ distribution for 3000 update packets sent periodically every 100ms, when using $\beta = 100$.

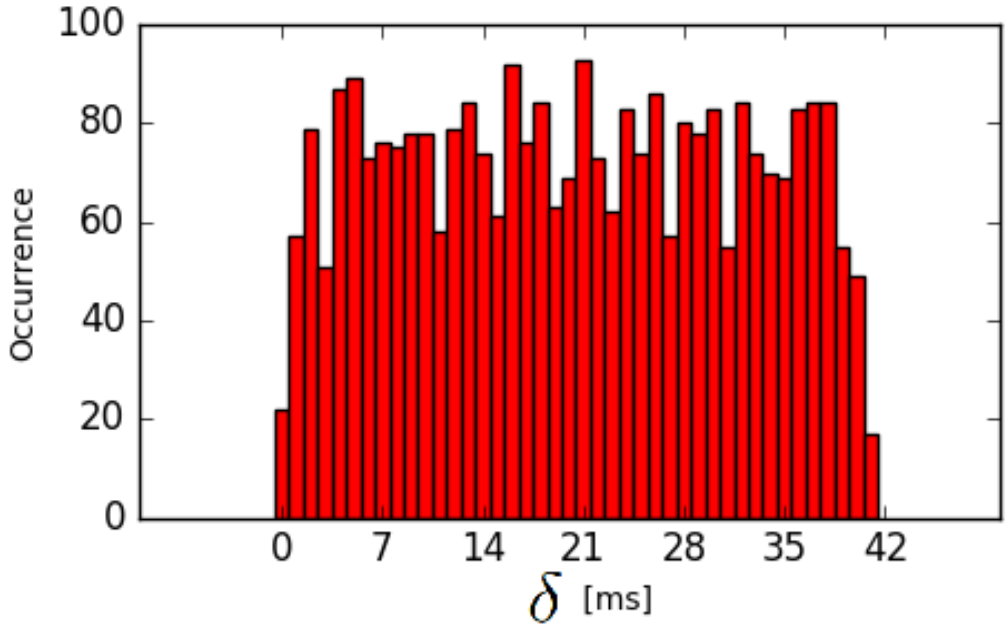

Figure 21: δ distribution for 3000 update packets sent periodically every 100ms, when using $\beta = 1000$.

6 Conclusion and future work

6.1 Conclusion

In this paper we described the implementation of a testbed for the grouping window scheduling policy. The testbed consists of an edge device build from a BeagleBone black, A sensor node using a MSP430fr5969, both communicating using a CC2500 radio transceiver. The edges was required to send a time schedule to the sensor node, after which the sensor node starting sending back update packets according to this schedule. Solutions to deal with memory, energy, communication and timing constraints was developed and explained with help of state chart diagrams. The effectiveness of the timing solution was evaluated, and the results showed that we could

acquire over 50% perfect timing with δ ranging between [-2, 2]ms. We also saw that good timing results required more energy consumption from a more frequent communication between the two devices. We also evaluated how the results from an experiment compared to the results from the theoretical literature in [6]. We saw that our result in the experiment confirms the results that the schedule policy could decrease AoI by as much as 60%.

6.2 Future work

We saw from the results of the experiments that the testbed has some flaws concerning packet loss and timing. A solution to this would be to install external crystals that are less prone to drift. This would decrease the need of synchronizing the software timers of the system. The packet loss could also be dealt with by increasing the bit rate of the radio transceivers. This wasn't done before the experiments because of trouble configuring the devices. A higher bit rate would also open up for other solutions, such as feeding a new schedule when the sensor node is done with the current schedule. This wouldn't limit the schedule sizes to the memory of the sensor node. Another way this testbed could be improved is by supporting multiple sensor nodes. This would open up for the possibilities of inter sensor node communication. It would also let us try the testbed in an environment where it may experience more communication problems. Last but not least, representing the δ in the correction packets with a higher resolution than milliseconds would make the sensor node adjust more precisely.

References

[1] S. Kaul, M. Gruteser, V. Rai, and J. Kenney. Minimizing age of information in vehicular networks. pages 350–358, June 2011. 2011 8th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks.

[2] S. Kaul, R. Yates, and M. Gruteser. Real-time status: How often should one update? pages 2731–2735, March 2012. 2012 Proceedings IEEE INFOCOM.

[3] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html`, Feb 2009. [Online].

[4] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. Fog computing and its role in the internet of things. `http://doi.acm.org/10.1145/2342509.2342513`, 2012. [Online].

[5] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. pages 14–23, Oct 2009. IEEE Pervasive Computing, vol. 8, no. 4.

[6] L. Corneo, C. Rohner, and P. Gunningberg. Age of information-aware scheduling for timely and scalable internet of things applications. pages 2476–2484, April 2019. IEEE INFOCOM 2019 - IEEE Conference on Computer Communications.

[7] K. M. Alam and A. E. Saddik. C2ps: A digital twin architecture reference model for the cloud-based cyber-physical systems. pages 2050–2062, 2017. IEEE Access, vol. 5.

[8] *MSP430FR5969 LaunchPad Development Kit (MSP-EXP430FR5969). SLAU535B.* Texas Instruments, Feb 2014. Rev. July 2015.

[9] *CC2500 Low-Cost Low-Power 2.4GHz RF Transceiver. SWRS040C.* Texas Instruments, May 2009.

[10] G. Coley. *BeagleBone Black System Reference Manual. BBONEBLK_SRM.* BeagleBoard, April 2013. Rev. A5.2.

[11] P. Thanigai and W. Goh. *Maximizing Write Speed on the MSP430 FRAM. SLAA498B.* Texas Instruments, June 2011. Rev. Feb 2015.

[12] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual.* Addison-Wesley Professional, 2nd edition, 2010. ISBN 032171895X 9780321718952.

A Generate Schedule

Let $A = (a_1, a_2, .., a_{n-1}, a_n)$ be a set of n application requests towards a specific sensor node. Now let $P = (p_1, p_2, .., p_{n-1}, p_n)$ be a set of periods where period $p_i \in P$ correlates to application $a_i \in A$. From this we can derive a set of execution times $x_a = p_a \cdot k, \forall k \in N_+$, where $a \in A$. Now we can define X which is all sets of execution times combined, X can be defined as.

$$X = \cup_{\forall a \in A} x_a$$

. We sort X in ascending order and use it in algorithm described in 2 to produce a grouping window schedule for all requests in A.

Algorithm 2 Grouping Window with Delay Budget [6, p. 2479]

X, the schedule of applications executions
τ, applications timing requirement
ϵ, applications delay budget
$S \longleftarrow \emptyset$
$i \longleftarrow 0$

while $i < |X|$ **do**
$j \longleftarrow 1$
    while $X_{i+j} \leq X_i + \tau \wedge i + j < |X|$ **do**
$j \longleftarrow j + 1$
        if $\epsilon > 0 \wedge \exists X_a \in [X_i, X_i + \epsilon]$ **then**
$S \longleftarrow S \cup \{X_{i+|X_a|}\}$
        else $S \longleftarrow S \cup \{X_i\}$
$i \longleftarrow i + j$
return S

x_a will be a set of infinite number of execution times. The resulting schedule using algorithm 2 will also have infinite number of execution times. Obviously this would not be feasible to fit into any device. Therefore only a part of this schedule will be used.

B UML diagrams

Here follows some diagrams made with help of the Unified Modelling Language(UML) [12]. One state diagram for each device explains the stable states and how external events move the system into another state. To be able to keep the details, each diagram is split up into several containers, each with its own logic and state(s).

B.1 State diagrams - Edge

Each container is a part of the bigger system. How everything is connected is shown in Figure 22. Each time the system is started it's file containing a periodic or grouping window schedule. If it's fed with the periodic schedule it will continue down the left branch of the state diagram into the container described in Figure 23. The edge device first sends a Period packet with information about the period and the number of periods (in Figure 23 referred as N_GOAL). After an ack packet is successfully received the program continues down to container described in Figure 24.

Here the edge will listen for incoming update packets, process information about error rate, lost packets and arrival time. This continues until the last packet has been received or when a system timeout occurs.

If the edge is fed with a grouping window schedule instead, it goes down the right branch of the state diagram and into the container in Figure 25. First it reads the grouping window schedule and then calculates the indices for the correction packets. The edge then sends the schedule chunk by chunk, until the corresponding acknowledgement packet has been successfully received for each sent packet. All the indices of the correction packets are then sent in a IDs Packet until it receives all the acknowledgements. This will make the edge continue to the next container, described in Figure 26. Here the edge will listen to incoming update packets, receive them, process the data about error rate, lost packets and arrival time. This continues until the last packet has been received or when a system timeout occurs. For each update packet received, it checks if the next expected packet id is one of the correction packet indices. If it is, the edge sends a correction packet holding the δ of the last update packet received.

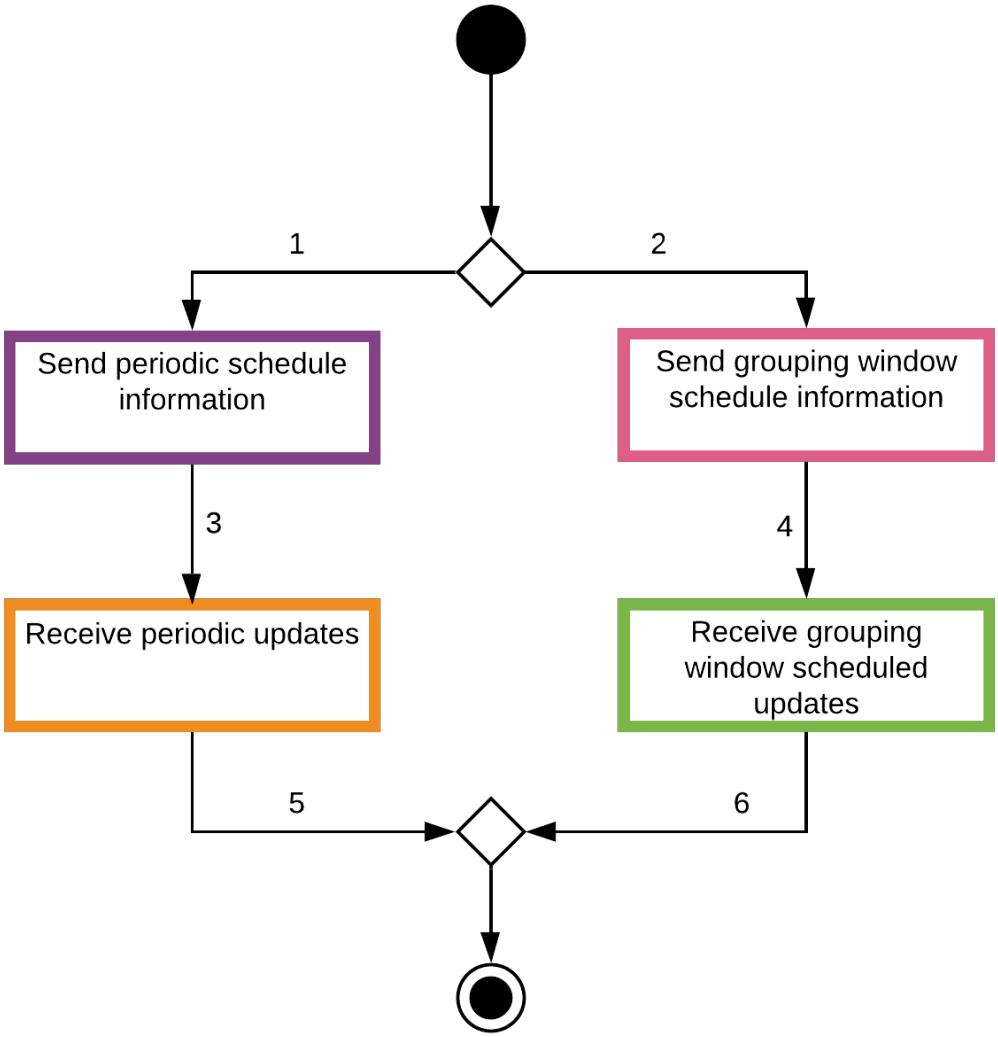

Figure 22: A simplified state diagram of the edge device.

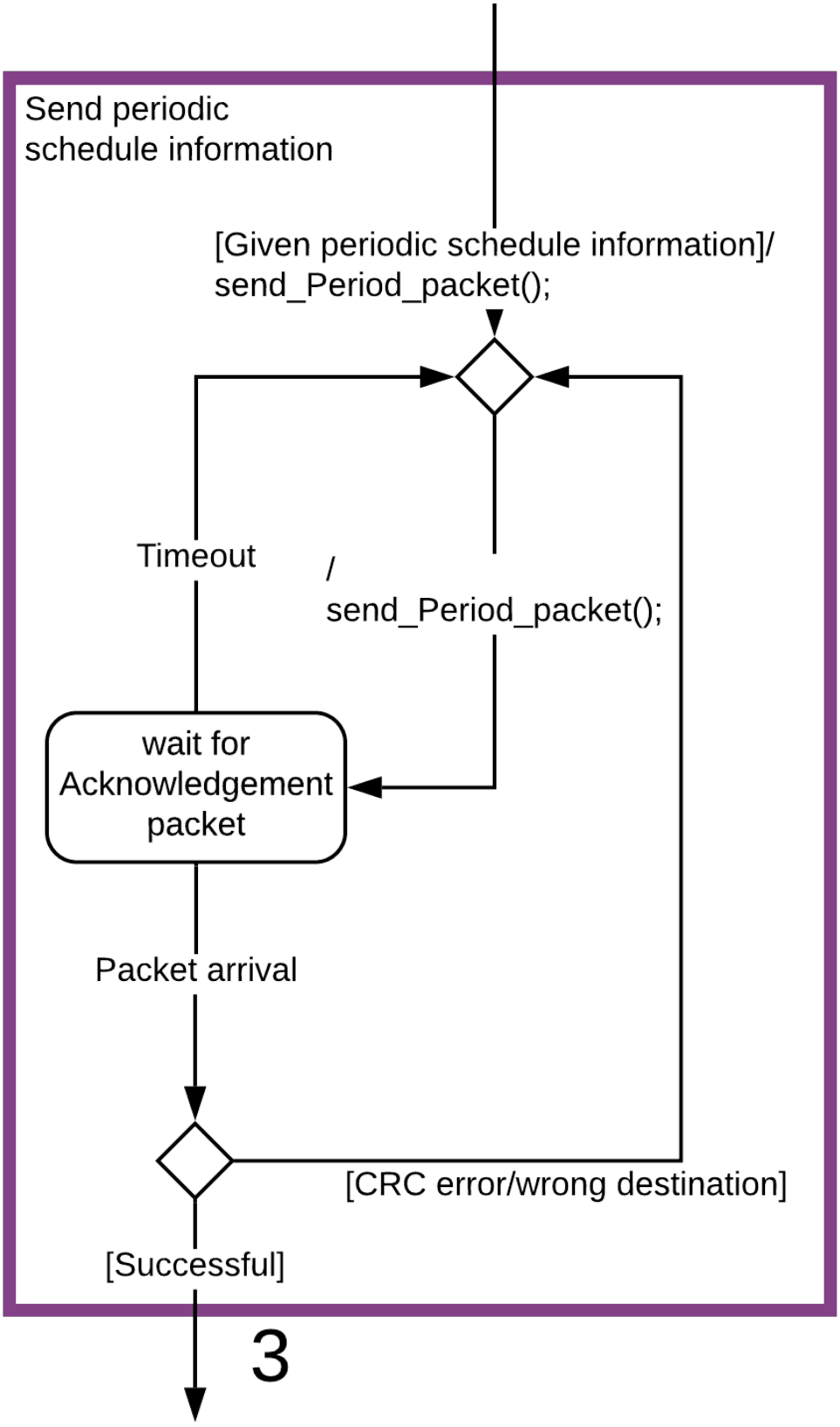

Figure 23: A part of the state diagram in figure 22 shown in more detail

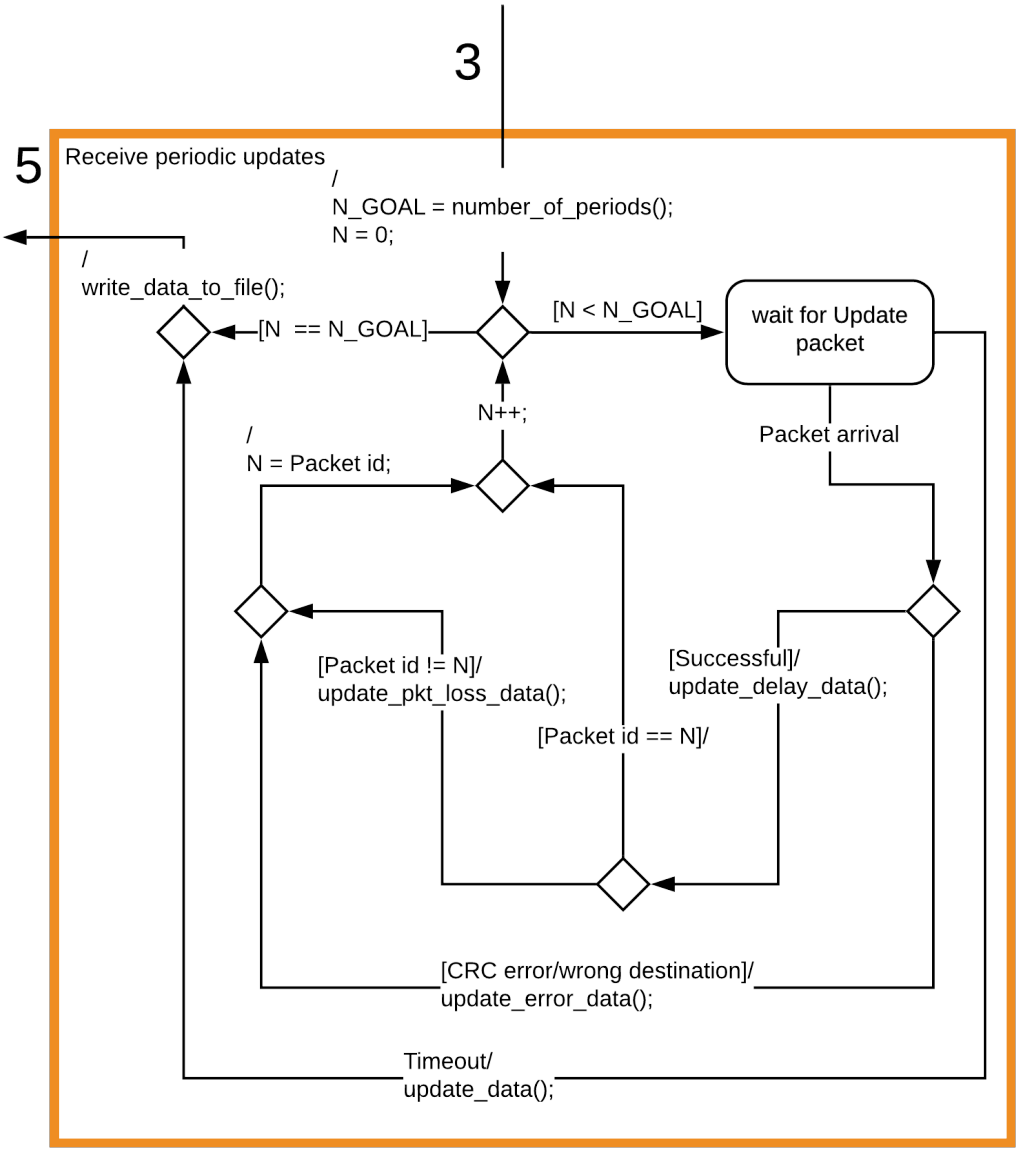

Figure 24: A part of the state diagram in figure 22 shown in more detail

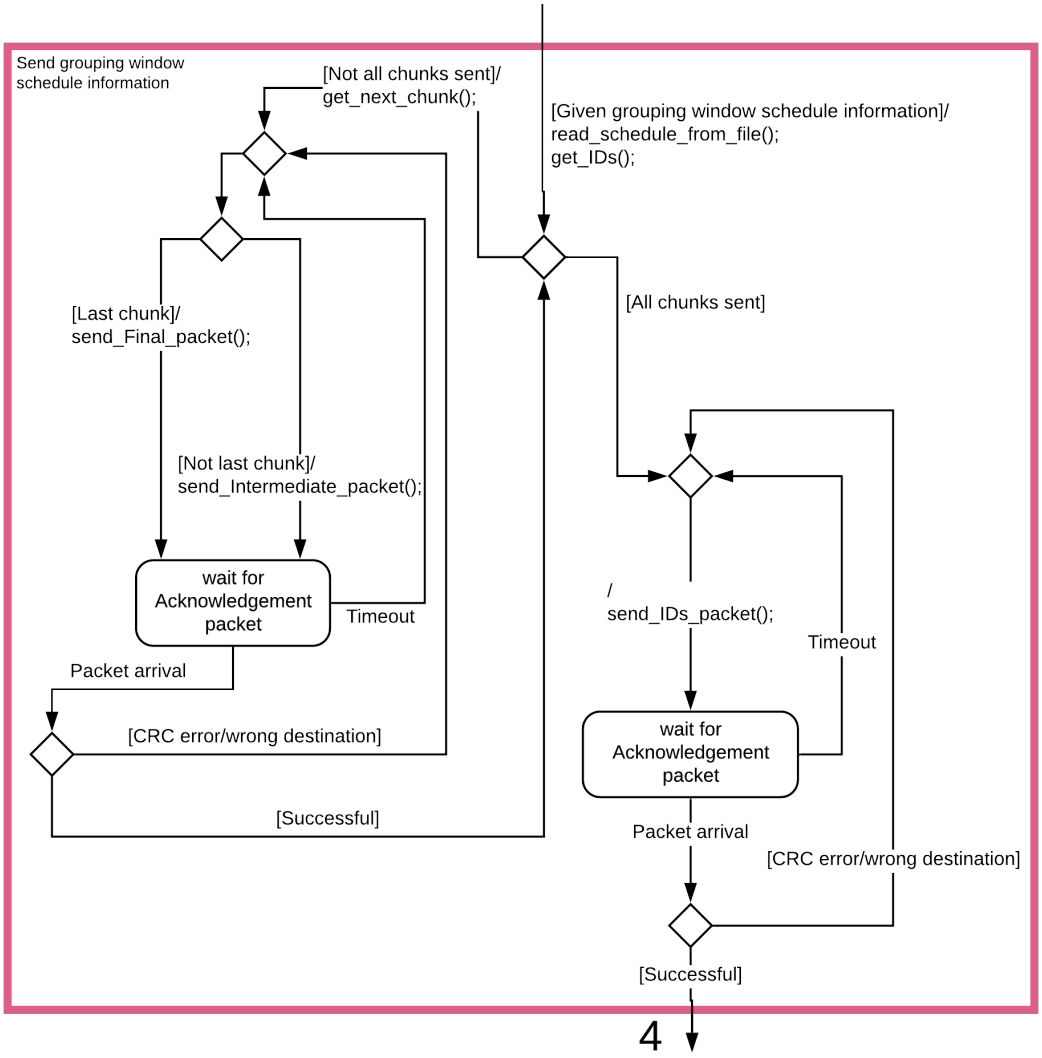

Figure 25: A part of the state diagram in figure 22 shown in more detail

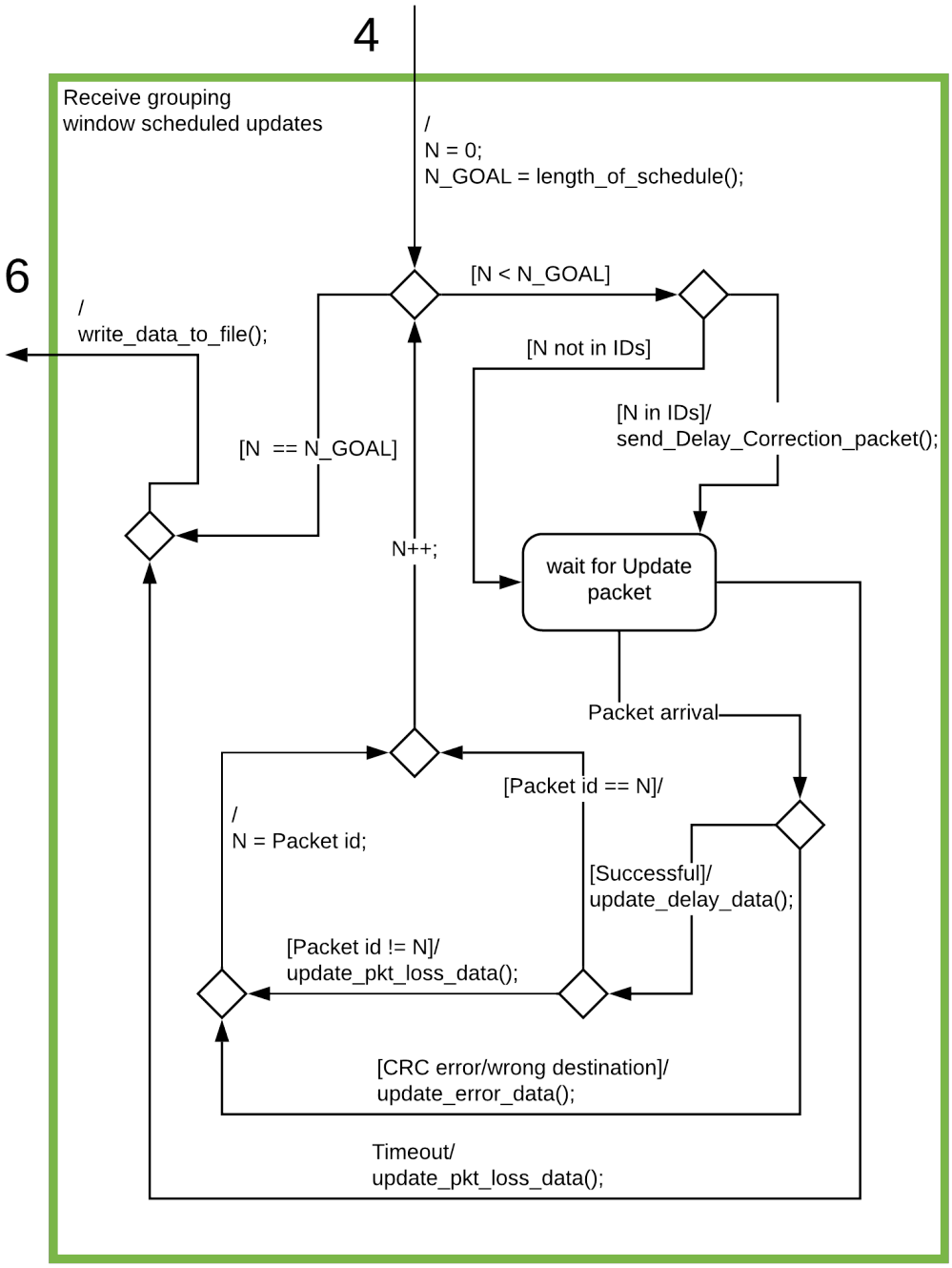

Figure 26: A part of the state diagram in figure 22 shown in more detail

B.2 State diagrams - Sensor node

Each container is a part of the bigger system. How everything is connected is shown in Figure 27. The sensor node will arrive at the first container described in Figure 28 at start up. It actively listens for incoming packets that holds information about how it should feed the edge device updates. This information is important and therefore the sensor node will always answer every successfully received packet with an acknowledgement packet to let the edge device know that it's ready for the next step. If a Period packet is received, the sensor node will continue to container described in Figure 29. The sensor node will start a timer that will trigger an interrupt every millisecond. Then it will start to feed update packets to the edge periodically. In Figure 29 the period is called PERIOD_NEXT_UPDATE is the time the sensor node should send the next update packet, N_GOAL is referring to the index of the last packet that should be sent. N is referring to the next update packet ID, and MS is tracking how many milliseconds that has passed since the timer was started. While in the "Wait for NEXT_UPDATE" state the device will be in low power mode. Each millisecond the timer will wake the sensor node up the sensor node will check if NEXT_UPDATE has been reached. If so, the packet is sent. N is after that calculated with the following formula.

$$N = \left\lfloor \frac{MS}{PERIOD} \right\rfloor$$

If the transmission time of last update packet didn't exceed the PERIOD, the result of the formula is that N is incremented by one. NEXT_UPDATE is then set to $PERIOD * (N + 1)$. This loops until $N >= N\_GOAL$.

Now to container described in Figure 28. Instead of the edge sending a Period schedule it may send a grouping window schedule. If the edge tries to send the sensor node a schedule that doesn't fit into one packet, it will first feed the sensor node with Immediate packets holding chunks of the schedule. These packets has a sequence number to let the sensor node know if a packet is missing. After all Immediate packets has been received, the sensor node is expecting a final packet holding the last information about the schedule. When that packet is successfully received, it waits for an IDs packet that carries the timestamp ids of when it should expect a correction packet. When this information is received it continues down to the next container described in Figure 30. Now the sensor node will feed the update packets to the edge device, according to the schedule. This is done in a similar way as sending periodically, but with some extensions. NEXT_UPDATE still holds the next time an update packet should be sent. Every time an update packet is sent, the device will check if it's time to wait for the next correction packet. This is performed by deciding if the id of the last update packet that was either sent or planned to be sent, N, exists in the correction IDs (IDs). If this is the case, the sensor node will actively listen for an incoming packet. Here a variable called TIMEOUT is introduced

$$TIMEOUT = MS + \frac{\alpha}{2}$$

This variable lets the sensor node wait for at most half of α for the expected correction packet, in case it was lost. If a correction packet is successfully received, the millisecond counter, MS, is changed according to the δ received. NEXT_UPDATE is now updated.

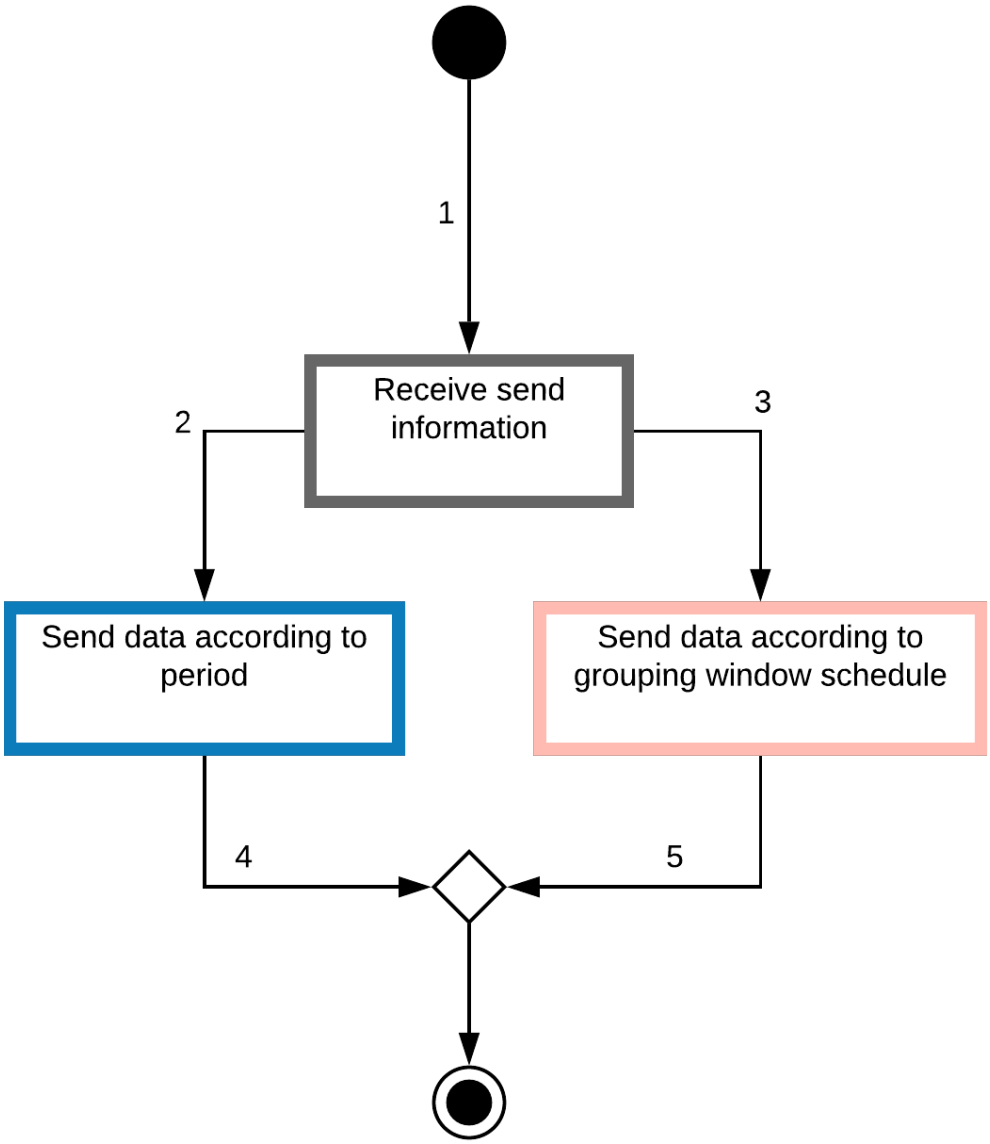

Figure 27: A simplified state diagram of the sensor node.

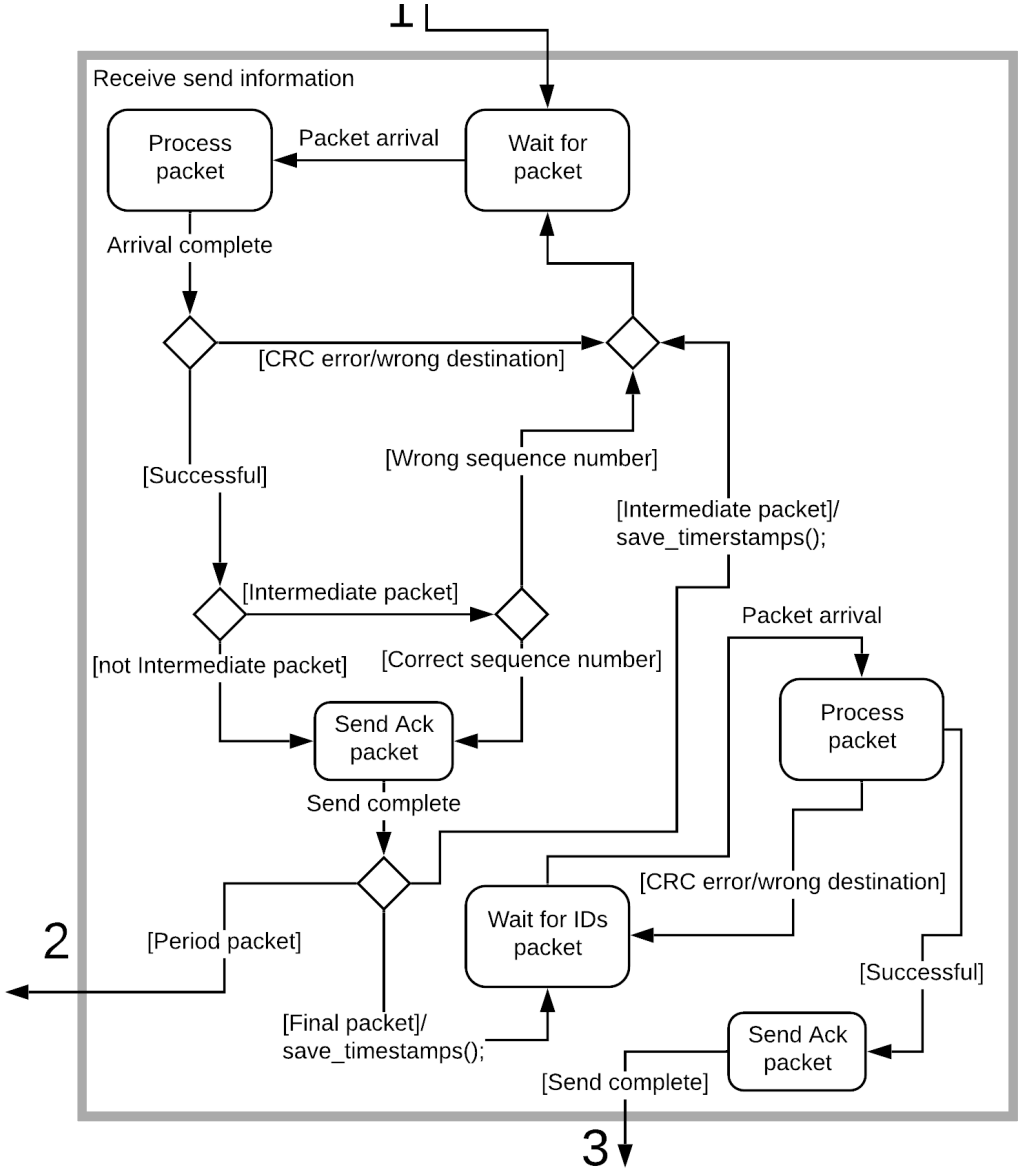

Figure 28: A part of the state diagram in figure 27 shown in more detail

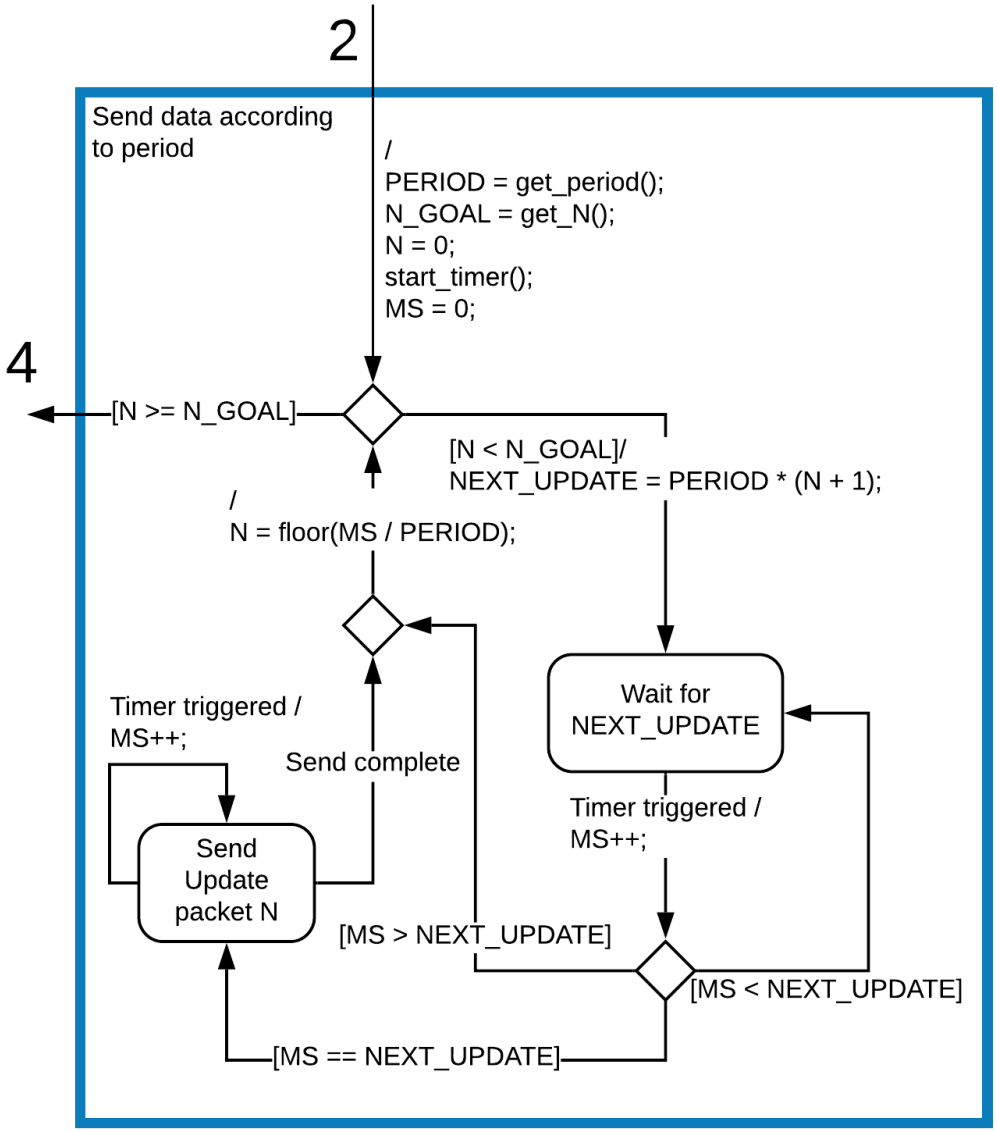

Figure 29: A part of the state diagram in figure 27 shown in more detail

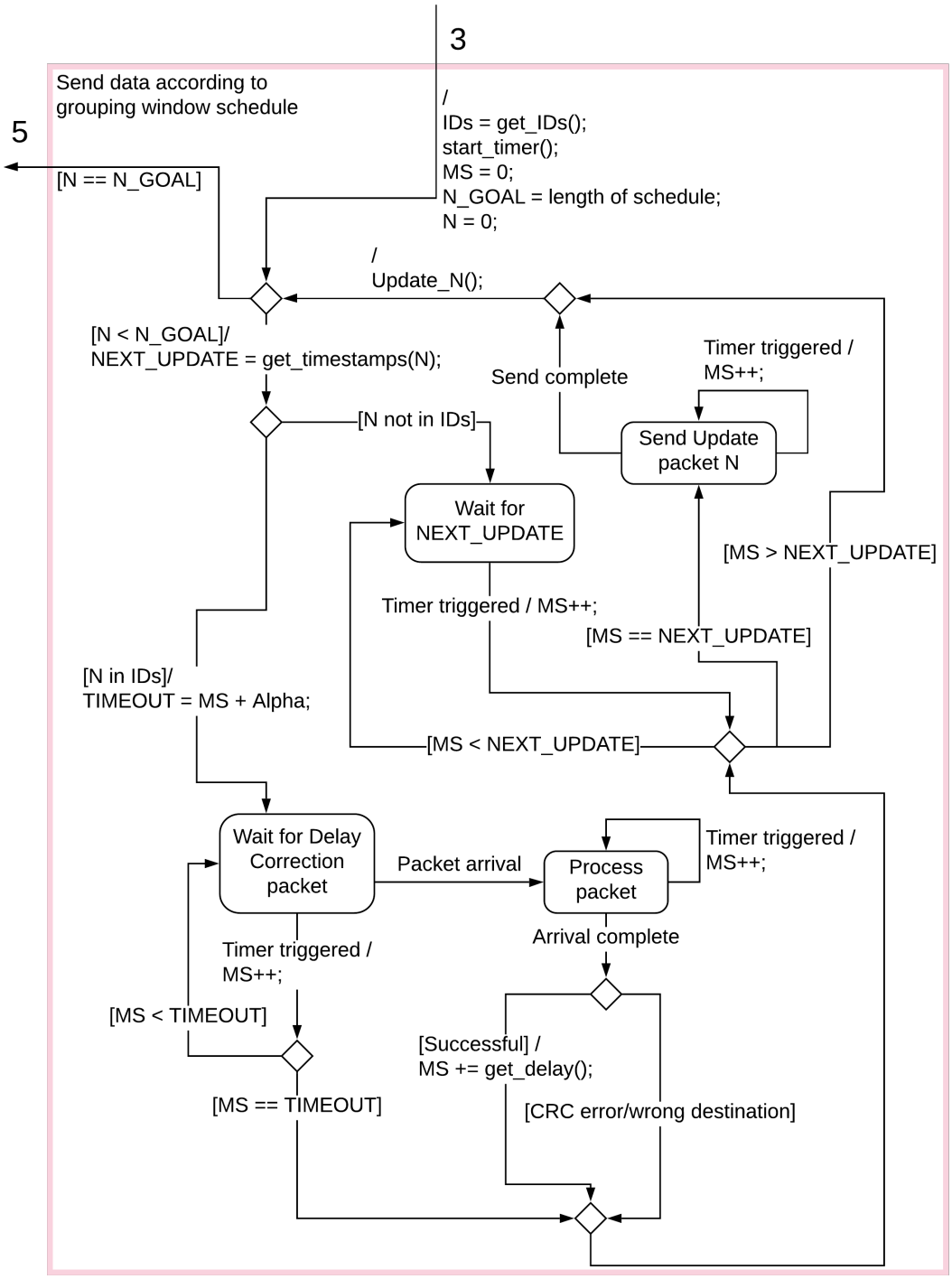

Figure 30: A part of the state diagram in figure 27 shown in more detail