Energy Efficient Code Updates in Wireless Sensor Networks

Validation and enhancement of the GCP protocol

Peter Finderup
Robertas Backys
Thomas Birk Abildgaard

AALBORG UNIVERSITET

Department of Computer Science
Aalborg University
Selma Lagerløfs Vej 300
9220 Aalborg
Phone 96 35 80 80
Fax 96 35 97 98
http://www.cs.aau.dk

Title:
Energy Efficient Code Updates in Wireless Sensor Networks
- Validation and enhancement of the GCP protocol

Project Period:
DAT6, spring semester 2010

Project Group:
d617a

Participants:
Peter Finderup
Robertas Backys
Thomas Birk Abildgaard

Supervisor:
Brian Nielsen

Circulation: 5

Number of pages: 98

Number of appendixes: 6 and 1 CD.

Finished on: 31th of May, 2010

Abstract:

Wireless Sensor Networks (WSNs) are typically deployed in scenarios where sensor measurements from several distinct locations are needed. WSNs usually consist of many nodes scattered around an environment, often one humans tend to avoid. Example of hard to reach or hostile environments could be inside a glacier or upon a battlefield. A problem occurs once a WSN has been deployed and the running application needs a software update, either because errors has been found, parameters are tweaked or application improvements are needed. Collecting each sensor node by hand would be a tedious task, impractical if not impossible. Therefore, when such sensors need to be updated another approach is required.
The lifetime of a WSN highly depends of the utilisation of the energy resource available. The energy supply of the individual sensor node is a very scarce resource, and the source of energy can seldom be replaced after deployment of the WSN. In this report we focus on energy efficient code updates in a WSN through wireless communication. We systematic describe the *Gossip-based Code Propagation* protocol and implement it.
Through the specification of the GCP protocol we discover ambiguous protocol description, as well as limitations to the protocol.
We present an extension which incorporates a reliability mechanism which ensures updates can happen though communication is faulty. We also improve the utilisation of tokens which ensures a better overall load balancing in dense networks.
As a conclusion, we compare our results from the GCP and our extension to reason about their performance while distributing code updates in a WSN.

Preface

This report is written by group d617a as a DAT6 (master) project from February 1^{st} 2010 to May 31^{th} 2010 at the Department of Computer Science at Aalborg University. This report is addressed to other students, supervisors and anyone else who might be interested in the subject. To read and understand the report correctly, it is necessary to have knowledge about the most basic computer related terms.
The entire report is written in English and no translation will be accessible, it is therefore required to understand English. Abbreviations and acronyms will at first appearance be written in parenthesis, to avoid breaking the reading stream. Specification of gender in this report is not to be understood as suppression or any other form of political/religious position. The gender is only specified to simplify the process of writing for the authors.
References to sources are marked by [#], where # refers to the related literature in the bibliography at the end of the report.
The Appendixes to this report is found in the last chapter of the report.
The report is written in LaTeX and are accessible as a PDF-document, which can be read with Adobe Acrobat Reader.
Finally, the group will like to thank our supervisor Brian Nielsen who has supported us throughout this project.

Content of the CD

- Graph folder with all graphs.
- Source folder with our source files.
- Simulation date.

Thomas Abildgaard Robertas Backys

Peter Finderup

Contents

1 Introduction **1**
- 1.1 Challenges . . . 1
- 1.2 Motivation . . . 2
- 1.3 Problem Statement . . . 3
- 1.4 Report structure . . . 3

2 Background on wireless sensor networks **4**
- 2.1 Typical hardware platforms . . . 5
- 2.2 Update management model . . . 8
- 2.3 Summary . . . 10

3 GCP protocol specification **11**
- 3.1 GCP Protocol . . . 11

4 Verification of the GCP protocol **16**
- 4.1 Initial GCP model . . . 17
- 4.2 Timing analysis . . . 20
- 4.3 Energy analysis . . . 26
- 4.4 Summary . . . 29

5 Simulation of GCP **30**
- 5.1 Introduction to NS2 . . . 31
- 5.2 Implementation of GCP . . . 34
- 5.3 Basic GCP tests . . . 38
- 5.4 Simulation scenarios . . . 39
- 5.5 GCP test results . . . 44
- 5.6 Explanation for deviation . . . 48
- 5.7 Replicate results of GCP . . . 50
- 5.8 Summary . . . 52

6 Extension of the GCP protocol **53**
- 6.1 Problems with the GCP protocol . . . 53
- 6.2 Specification of extension . . . 58
- 6.3 Implementation of eGCP . . . 62
- 6.4 Test of the extension . . . 68
- 6.5 eGCP test results . . . 70
- 6.6 Summary . . . 73

7 Conclusion **74**
- 7.1 Future work . . . 77

A Calculating average density **80**

B UPPAAL models **82**
- B.1 UPPAAL model of the original GCP protocol . . . 83
- B.2 UPPAAL model of the original GCP protocol with a half duplex radio . . . 84

B.3 UPPAAL model of the original GCP protocol with time and node connectivity . . . 85
B.4 UPPAAL model of the original GCP protocol with energy, time and node connectivity . . . 86
B.5 Maple Time and Energy calculations . . . 87

C GCP results **88**
C.1 GCP simulation results - Scenario 1 . . . 89

D GCP Replication results **93**

E eGCP results **95**

F Report summary **97**

Chapter 1

Introduction

Today modern technology improves faster than ever before. The numbers of transistors that can fit a square inch increases each year and thus electronic devices get smaller and smaller. This development is of great importance for sensor nodes since their actual size can determine if they can be applied in a specific area or not. An example could be a nature habitat, where the miniaturisation of the sensor nodes allow us to collect data in a non intrusive manner. In this scenario tiny, almost invisible sensors become essential in terms of getting accurate data without interfering with the normal life of the animals or polluting the area.

Wireless sensor networks (WSNs) are typically deployed in scenarios where sensor measurements from several distinct locations are needed. They usually consist of many nodes scattered around an environment, often one humans tend to avoid; either because its costly or too dangerous. Examples of such areas are at the bottom of the ocean, in space, on a battlefield[8] or deep inside a glacier[15].

A problem occurs once a WSN has been deployed and the running application needs an software update, either because errors has been found, parameters are tweaked or application improvements are needed. Collecting each sensor node by hand would be a tedious task, impractical if not impossible. Therefore, when such sensors need to be updated another approach is required.

In this report the focus will be on energy efficient code updates in a WSN through wireless communication. However, several challenges arises when updating a sensor network wirelessly.

1.1 Challenges

There are many challenges when updating a wireless sensor network. Physical reachability can post a problem. Transmitting the update over an unstable media such as a radio is troublesome. The main problem with wireless sensor networks is *power consumption*. When a WSN is deployed the sensor nodes only energy source is the battery. Although methods for energy harvesting has being examined through solar panels, minimizing power consumption is highly prioritized.

An effective way to minimise power consumption during a wireless transmission of an update is to use binary differential patching. Differential patching produces a patch (delta patch) that is smaller than the original code update. The delta patch contains the differences between the old- and the new software image. Differential patching is particular effective when distributing smaller incremental code updates. The power consumed by applying the patch at the target will increase as the delta patch complexity increases and will ultimately outweigh the benefit of conserving power while transmitting the smaller delta patch.

Another area of interest is dissemination. It is possible to minimize the power consumption at the expense of latency in the network. This is done by decreasing the use of *flooding* protocols in order to reduce the chance for duplicated messages and collision. The trade-off here is increased inconsistency because some of the nodes are updated while others are not for a period of time due to the slow diffusion of code updates. For some applications latency is not important and can be sacrificed, for others it is crucial. It all depends on the WSN and the running application. Common challenges in a wireless sensor network are message *implosion* and *overlapping of interest* as described

in [11].

Another challenge updating a WSN is the topology of the network. Within every ad hoc network a set of nodes will typically become hot spots due to their physical placement. A general problem is that those hot spot nodes tend to use their battery power faster than the other nodes in the network. An example could be an ad hoc network which was grouped into clusters with only a few nodes to connect them. Every time information has to be spread to the other clusters those nodes will have to relay the data, which will increase their power consumption and will eventually lead to a disjoint network.

A way to diminish the workload of those centralized nodes is to incorporate the ideas of evenly load balance as described in [9]. The purpose of this protocol is to propagate code updates throughout an ad hoc wireless sensor network in an energy efficient manner. The protocol *Gossip-based Code Propagation* (GCP) in [9], uses tokens to balance the workload, when updates are disseminated throughout the network. This is a simple mechanism where each node gets a number of tokens which denotes the number of times a given node can transmit an update. For every transmitted software update the number of tokens decreases until it reaches zero. When the number of tokens reaches zero, the node will conserve its energy by refusing to transmit more code updates of this particular version. The forwarding control mechanism will balance the workload of updating the entire network. Whenever a node receives a new update, the number of tokens are reset to the initial number of tokens, regardless of the numbers of tokens prior to the update.

1.2 Motivation

We want to improve the energy consumption while diffusion updates in a wireless sensor network. When distributing an update, the load balance of the nodes will vary. If the balance is too uneven the network becomes fragmented which in worst case can prevent the running application to fulfil its purpose. A way to solve this problem is to use the ideas of even load balance as described in [9]. The results presented by the authors in [9] look very promising. Going into greater detail of the GCP protocol reveals some flaws though. The specification of the GCP protocol in [9] is insufficient as important details are omitted. For instance, the movement pattern of the nodes is not clearly specified. Another example is the description of the periodically transmission of the beacon message. According to the authors this message should be transmitted after a given time out throughout the lifetime of the sensor node. But a radio can only either transmit or receive at one time. This means that if the radio was in a middle of receiving an update message and the time out happens, the radio apparently has to switch and send the beacon before it continues to receive the update, unless it postpone the message till after the reception has been completed. Whatever happens is unclear since the precedent rules have not been clearly specified. The authors assume that no collision / packet loss happens in their wireless setting, thus they do not need to specify recovery management, a vague assumption in our opinion since we operate with wireless communication. Also, the way tokens are used when sending update messages has not been specified properly. This means that under certain circumstances multiple tokens can be consumed whereas one was enough.

Lastly, the authors have developed their own network simulator called *SeNSim*, instead of using a widely accepted and proven tool like the *Network Simulator 2* (NS2). We find it interesting to see, if the GCP protocol provided the same results in a "neutral" simulator.

1.3 Problem Statement

In this report we look into the work in [9]. We give a systematic specification of the protocol using the protocol specification methodology of [12]. With this specification we create a formal model of the protocol in UPPAAL[23] and verify sanity checks while performing time- and energy analysis on selected topologies. We implement the GCP protocol, verify it and run the same test scenarios as in [9]. This is done with the NS2 tool. If our results does not match those of the authors of the GCP protocol replications of their results will be made to determine the possible reasons for the mismatch.

As the specification of the GCP protocol was made several limitations of the protocol became clear. We therefore present an extension to the GCP protocol called enhanced GCP (eGCP). A systematic specification of our protocol will be specified. In eGCP protocol we add a reliability mechanism, so the network can be subject to failures. We also change the way tokens are used in order to improve the load balancing. An implementation of the eGCP protocol will be made and tested to see if the behaviour matches our expectations. Finally, the GCP and eGCP protocol will be compared in the selected test scenarios.

1.4 Report structure

In chapter 2 we give a characterisation of typically hardware used to create a wireless sensor network. In this chapter We also present an update management model which describes the different important parts used when diffusing an update in a wireless sensor network.

In Chapter 3 we introduce the GCP protocol. We use the methodology of [12] to create a thorough specification of the GCP protocol. We finish the chapter with a discussion regarding our view of the protocol.

The next chapter (Chapter 4) contains a validation of the GCP protocol. The tool UPPAAL is used to verify that the GCP protocol is free of deadlock, and that the intended update behavior is correct. We further extend the GCP UPPAAL model in order to conduct time and energy analysis.

Chapter 5 starts by introducing the simulation tool we used to conduct our tests. Then a description of our GCP implementation will be given followed by verification of it. Then a test of the GCP protocol will be conducted and the results will be matched with the original obtained. In Chapter 6 we define the problems with the GCP protocol and give ideas to how they can be corrected. This leads to a systematic specification of our extension to the GCP protocol. Afterwards we implement and test our extension which ends in a presentation of the results found when comparing GCP with eGCP. Finally, a conclusion is given. In this, we summarise the different results found in the previous chapters. Lastly, different thoughts on future work will be described.

Chapter 2

Background on wireless sensor networks

This chapter describes background knowledge regarding updating a wireless sensor network. As briefly described in the Introduction chapter, wireless sensor networks may be applied in environments which humans tend to avoid. An example is at the bottom of a glacier where the network monitors the movement [15]. Nodes are placed in layers through the ice and relay information back to the surface. From the surface the base station has the resources to transmit the information back to the research centre.

Due to the increasing fault-tolerance, robustness and the ad hoc nature, WSNs becomes attractive for military application and other risk-associated applications. For instance, in [8] aerial deployment is used to create a WSN for detection and tracking of vehicles.

In this chapter we give a characteristic of the hardware in a wireless sensor network. This is done to emphasize the resource constraints in these networks and the importance of optimising the energy usage of the running application to prolong the networks lifetime. To conserve energy, a number of possible fields can be addressed. One of these fields is the physical hardware of the network, another is to minimise the use of communication.

In the following, we look into the physical hardware of wireless sensor network. Firstly, we look into two possible hardware platforms and describe their available resources to emphasise the need for awareness. Afterwards we describe an available radio for a wireless sensor network together with the different memory available. Then we introduce an update management model which describes the various parts included when an update is created and diffused. Choosing the right configuration in each of these parts have an influence on whether a WSN application will become a success or not.

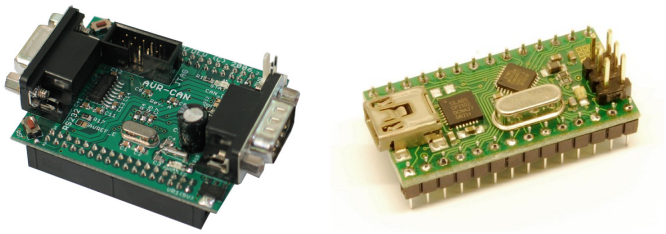
2.1 Typical hardware platforms

A large range of hardware platforms exist depending on the purpose of the WSN. Therefore, in order to chose the right platform, a resources prioritisation is needed. Some applications require a lot of memory, others needs a special radio, while some is more concerned with the physical size of the platform. In this section we look into different hardware and the apertaining memory types.

Development boards

Examples of typical hardware platforms for a wireless sensor network are the AT90CAN128 or the CRUMB168-USB. Both of these platforms have restrained resources and the main difference between them are their interface and memory. The AT90CAN128 uses JTAG (Joint Test Action Group) and CRUMB168-USB uses SPI (Serial Peripheral Interface). A comparison between the two development platforms are illustrated in Table 2.1. Looking at the data it becomes obvious that the AT90CAN128 is the most powerful of the two, which makes sense since its almost twice the size. But node resources and power consumption are linked together, and with the same battery power the life time of the CRUMB168-USB far surpass that of the AT90CAN128. This emphasize the need of a

thorough analysis in order to select a board which has just enough resources available for the needed task.



(a) (b)

Figure 2.1: (a) The AT90CAN128. (b) The CRUMB168-USB.

	AT90CAN128	**CRUMB168-USB**
FLASH	128kB	16kB
RAM	4kB SRAM	1kB SRAM
EEPROM	4kB	512B
PIN connection	JTAG	SPI
IO PINs	53	23
CPU	0-16 MHz	0-20 MHz
Supply @ $25\,^{\circ}\mathrm{C}, 4.5V_{cc}$**, (active)**	(16MHz) 27 mA	(1Mhz) 0,75 mA
Supply @ $25\,^{\circ}\mathrm{C}, 4.5V_{cc}$**, (idle)**	(16MHz) 16,5 mA	(1Mhz) 0,125 mA

Table 2.1: Comparison between the AT90CAN128 and CRUMB168-USB. The data from is from [2][1].

Radio

Another important element is the radio since communication is one of the most power consuming parts of the WSN. As stated in [20], sending one byte over the radio requires the same amount of energy as 1000 CPU instructions, which illustrates just how expensive radio communication is. Some radios provides the developer with the opportunity for customising several parameter. Examples of parameters could be the length of the cyclic redundancy check (CRC), the number of hardware retransmissions, which interrupt should trigger which event and so on.

Each of these parameters gives the developer an opportunity for customising his implementation and although it induce complexity it also produce the most energy preserving approach since it can be optimised to fit the needs of the WSN application. A potential radio could be the radio nRF24L01 (depict in Figure 2.3). The specfic range of the radio is left out since it varies depending on the weather and landscape. Typically range would be 10 - 20 meters inside with a throughput on 1 Mbps. Tests with the radio shows that transmission rates of 256 kbit/sec at a distances of 100 meter is possible given optimal conditions[5]. The radiation pattern around the antenna resembles two overlapping circles circling in 360ˆ∘, the actual transmission pattern resembles a doughnut without a whole in the middle as illustrated in Figure 2.2

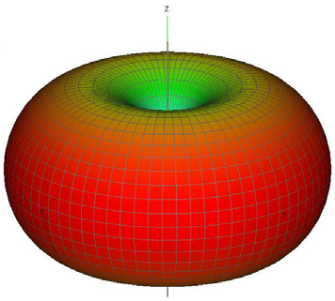


Figure 2.2: The radiation pattern around an antenna.

This radio includes an embedded packet handler called *ShockBurst*, which makes bi-directional communication protocols easier to implement. The radio has six data pipes with a data rate of either 1 or 2 Mbps which makes it possible to run as both half-duplex or full-duplex. Assuming only one pipe is used, the data rate is set to 1 Mbps and output power to 0 dBM (highest) the radio uses 22 μA in standby, 11,3 mA in transmit mode and 12,3 mA in receive mode [5]. Using all six pipes ("MultiCeiver") will allow the radio to communicate with six different sources, but because only one data pipe can receive a packet at a time, the throughput is not linear with the number is used pipes.

Figure 2.3: MOD-NRF24Lx RF 2.4GHz transceiver module with NRF24L01 chip.

Memory

As memory is one of the areas in which wireless sensor networks are severely constrained, efficient utilisation of the memory is critical. Typically, sensor nodes have two types of memory - flash and EEPROM[1]. Sensor nodes also contains some amount of SRAM[2] but since they are not used related to updates we omit them.

Flash is non-volatile memory, which means no power is needed to maintain the information stored in the chip. Additionally, flash is very robust and can withstand high pressure, extremes of temperature and immersion in water - all weather phenomenons which a WSN has to endure. When a sensor node receives packets for a new update it needs a place to store the packets until all packages have been received. Usually packets are stored in a part of the internal flash memory or by utilizing external flash memory. Lastly, the packets are assembled and moved to the internal flash application area. The task of writing to flash is very time and energy consuming. Writing one 1kB to flash consumes approximately $2.2mJ$, compared to $62\mu J$ for reading a 1kB from flash. More on flash charaterstics in Section 4.2 and 4.3.

[1]Electronically Erasable Programmable Read-Only Memory
[2]Static Random Access Memory

The EEPROM is also non-volatile memory and was original designed to store configurations information when the device was turned off. The EEPROM can only write / erase 1 byte at the time. Both EEPROM and flash have a finite number of erase/write cycles. The advantage of the EEPROM over the flash is that it can withstand 100.000 erase/write cycles compared the the flash that can do only 10.000 writes/erase cycles.

2.2 Update management model

Different approaches can be taken in order to influence the efficiency of a software update within a WSN. One way to get an overview is by using the WSN update management model described in [10].

This model divide a WSN into different components, namely a non resource constrained environment called the *base station*, and the very resource constrained *sensor network*. An illustration of the two environments and their appertaining parts can be seen in Figure 2.4.

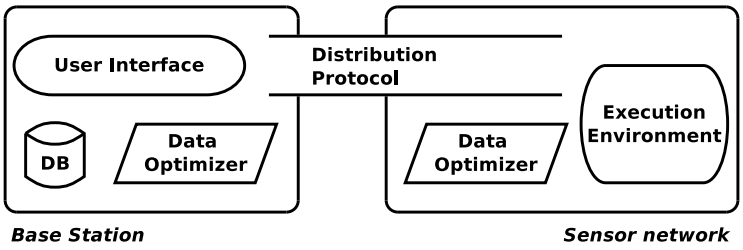


Figure 2.4: The update management model showing the two different environments - the base station and the sensor network.

User interface

The user interface can be used to schedule updates to a wireless sensor network. Through a browser, system administrators can log into the database and tweak the needed parameters or correct errors to create the update. Afterwards, remote monitoring of the progress of the update is (typically) possible.

Database

The database is the backbone of a WSN. Through it, system administrators can create, schedule and store updates. A database is used when the purpose of the WSN is to gather data compared to other applications which purpose is to detect events such as a fire alarm. Finally, The database also enables administrators access to the collected data which comes in hand if the WSN is deployed in a hostile or harsh environment.

Execution environment

The execution environment (EE) represent the way code is executed on the sensor nodes. The EE can be subdivided into *Monolithic*, *Modular* and *Virtual machine* environments. Each of them with their own characteristics regarding execution efficiency and energy consumption. In the following, we briefly summarize the pros and cons of the three sub-environments.

Monolithic

In a monolithic EE, the applications and kernel are closely intertwined. This means that compile time optimisation provides very efficient execution and memory utilization [10]. Because the core and applications are strongly tied together the typical way to update a monolithic EE is to update the entire software image all together.

This update approach is quite expensive energy wise, since it both requires a lot of radio communication to transmit / receive the software image, and then often requires external flash memory to temporarily store it. While updating a sensor node, it may be desirable to preserve the data and states associated with the program. Unfortunately, memory allocation for a monolithic EE is resolved at compile time and memory is accessed directly. This means that a "fresh" software image will have no knowledge of the memory layout of previously installed software. Therefore, preserving data and states in this environment can pose a problem. In general, the monolithic execution environment is expensive to update while having excellent utilization of the sensor node's resources.

Modular

A modular execution environment consists of two parts - a static part being the core / kernel and loadable modules. Memory is allocated during the loading of the modules (instead of compile time) and therefore, the modular EE must use indirect addressing e.g. through lookup tables. The use of indirect addressing gives the Modular EE, an execution overhead compared to the Monolithic EE. On the other hand, its possible to update a single module compared to updating the entire software image. For frequent small software updates the modular EE performs better energy wise compared with the monolithic EE.

Virtual machine

The virtual machine EE provides a virtual instruction set through its own high-level hardware abstraction. Compact scripts can be executed through an interpreter, thus perform complex tasks with very few instructions. When updating a sensor node, the running script on the virtual machine is replaced by another. As the scripts are relatively small, updating a node consumes very little power. Interpretation of the scripts are done within a sandbox environment with no direct hardware access. This gives the virtual machine EE the beneficial property of code safety [10]. The penalty of of using the virtual machine EE is execution overhead caused by the interpretation of the scripts. Despite this overhead, the virtual machine EE becomes a better alternative to both modular and monolithic EE as the frequency of system updates increases [10].

Diffusion protocol

When an update is pending it needs to be distributed efficiently. There exist a lot of different dissemination protocols, each targeted for a specific wireless sensor network topology. Some wireless sensor networks consists solely of sensor nodes. This simplifies the choice of diffusion protocol since extensive use of multicast will be discarded, thus ruling out classic flooding alike protocols. Another option could be classic gossiping. With this approach the energy consumption will be kept on a minimum and if the running application is not relying heavily on consistency the latency induced by gossiping could be acceptable.

A subscription based approached [20][11] is often preferred since this gives the fastest distribution while still preserving energy efficiently. A comparison between ACK- and NACK-based acknowledge could be interesting in determine the amount of overhead.

Another aspect which has a lot of impact on the performance of the diffusion protocol is the topology of the wireless sensor network. If the average density is high, the speed boost gained from flooding's multicast approach will diminish significantly because of the occurring implosion.

Data optimization

The role of data optimizatino is to minimize the size of updates before it is distributed in the network. One way of data optimizing an update is by data compression. Data compression of the software update is performed on the basestation and later uncompressed on the sensor node. The compression algorithm that produce the best compression ratio, is not necessarily the best choice. The energy used by the EE while decompression the software update on the sensor node may outweight the benefit of transmitting the smaller update. Another approach is to utilize delta patching. Instead of transmitting a completely new image, or compressed image containing the update, a differential script can be used instead. This script contains the changes between the old and new image (thus the name differential).

2.3 Summary

In this chapter we have described the hardware setup for a typical wireless sensor network with focus on their resources. This was done to emphasise the need for resource awareness when working with these networks. We also present an update management model which in detail describe the different parts needed to diffuse an update in a wireless sensor network. Each part has a huge influence on the efficiency of spreading an update within a wireless sensor network.

GCP protocol specification

In this section we give our own detailed and systematic specification of the work presented in [9] and outline their protocol *Gossip-based Code Propagation (GCP)*.

3.1 GCP Protocol

To specify the GCP protocol we follow the methodology of describing protocol structure given in [12]. Five distinct parts are described in order to complete the specification of a protocol. The first part is the *service* provided by the protocol. The second part contains the *assumptions* about the environment in which the protocol is executed. The third part contains the *vocabulary* of messages used to implement the protocol. The forth part describes the *encoding* of each messages within the vocabulary. The final part contains the *procedure rules* regarding the consistency of message exchanges. The fore mentioned parts are discussed separately in the following subsections.

Service

The protocol provides code updates between sensor nodes while incorporating *Piggy-Backing* and *Forwarding Control*, two control mechanisms both inspired by the flooding paradigm. The code updating is achieved through neighbourhood awareness in form of `Hello` messages called *beacons*. The beacons are transmitted at predefined intervals continuously throughout the lifetime of the sensor network in order to continuously alert/discover new software versions.

The workload of performing updates in the network is distributed between the nodes. Each node starts with a predefined number of *tokens*. Whenever a node has transmitted an update, its token count is decremented, and when it reaches zero, further updates cannot be transmitted until the token count has been reset.

Environment assumptions

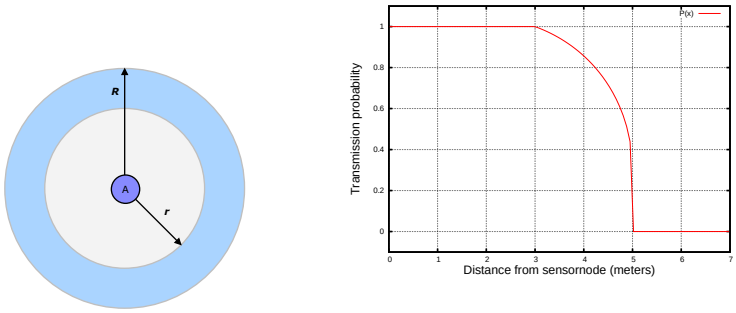
The environment of the GCP protocol is a distributed system with a large finite number of wireless sensor nodes which are unaware of their geographical position. No explicit assumptions are clear from [9], but test scenarios ranges from 1900 (plus 100 *transmitters*)[1] to 2250 nodes distributed according to different test scenarios. The sizes of the deployment areas goes from $0,625km^2$ to $4,0km^2$ which gives an average density of 1660 nodes per km^2 [2].

The original description in [9] assumes 1-hop broadcast between nodes in transmission range (no collision). Two ranges of transmission is considered r and R. The range r, is the range where target nodes will receive transmissions with 100 % certainty hence probability P=1. The range $\geq r$ and $\leq R$ represents the range where the transmission probability is not uniform. No nodes with a distance greater than R can receive transmissions [3]. It is illustrated in Figure 3.1.

[1] A transmitter is a node which movement is not restricted to the specific cluster and thus is used to assure connectivity between the clusters in the deployment area.

[2] The calculation is included in Appendix A.

[3] All nodes are considered to have the same communication range.



(a) Transmission ranges (b) Transmission probability

Figure 3.1: Transmission ranges and probability

Target nodes within the transmission distance $r < d < R$, may or may not receive a transmission. The probability of a node receiving a transmission is calculated according to Equation 3.1 below. The authors indicates $r = 3$ and $R = 5$

$$[9]P(d) = \begin{cases} 1 & \text{if } d < r \\ P_{min} - \sqrt{\frac{R-d}{R-r}} \times \left(\frac{R-d}{R-r} - 5\right) \times \frac{1-P_{min}}{4} & \text{if r} \leq d \leq R \\ 0 & \text{if } d > R \end{cases} \tag{3.1}$$

where P(d) is a function that describes the probability for packet transmission between two nodes separated with a distance of d meters. P_{min} is the minimum reception probability for two nodes separated by distance R. In [9] P_{min} is equal to 0,3.

The network is dynamic. By introducing the *Random Waypoint* model [13] nodes will change a movement direction as soon as the reach a way point, thus every node will eventually be in range of another node to receive the update. It should be noted that depending on the test scenario, different restrictions are enforced on the nodes movement (detailed description of the different test scenarios will be given in Chapter 5.)

Message vocabulary

The vocabulary of the GCP protocol can be expressed with two distinct types of messages: *beacon* messages with the purpose of broadcasting the nodes current version number and the *update* messages which contains the update. Another way to describe the vocabulary is with a set:

$$V = \{beacon, update\}$$

It should be noted that the GCP protocol assumes no collision or data loss would occur. Therefore, the "acknowledgement" and "not acknowledgement" messages are not needed.

Message encoding

Each message consists of a control field identifying the message type succeeded by a version number and lastly a data field.

As in [12] the general form of each message can be expressed in more details as a structure of three fields: { *control tag, version, data* }. Using this structure a C-like description could be written as:

```
enum control { beacon , update };

struct message {
        enum control        tag;
        unsigned int        version;
        unsigned char       data [30];
};
```

It should be noted that the beacon message does not contain any update data and thus the data field is omitted. The detailed specification of each message is omitted in the work of [9], thus a likely representation is given in Figure 3.2. Each field has the most significant bit (msb) as its outer left bit, and the outer right bit is the least significant bit (lsb).

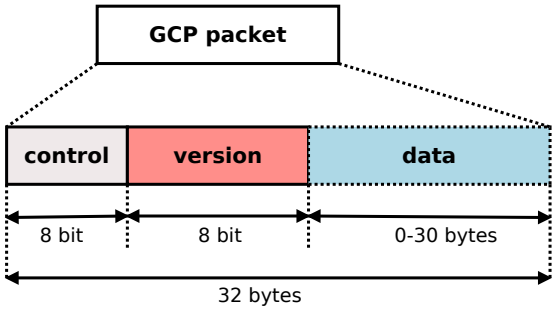


Figure 3.2: Presumably the GCP packet layout.

Procedure rules

The procedure rules of the GCP protocol is informally described trough pseudo code in [9]. We use a flow chart diagram to describe the protocol. To distinguish between an nodes own variable and one it receives with a message we adopt the notation of *_r*. An example is *version* > *version_r*, which means that a nodes version is greater than the transmitting node. Also, states are marked with **bold**, message types in `verbatim` and variables with *italic*. It should also be noted that in every state the radio is able to receive any kind of messages. To simplify our flow chart, for every state we have omitted those message types which will be discarded upon reception. In the flow chart the following labels are used:

- *version* represents the local version number of the software.
- *software* represents the local software binary.
- *tokens* represents the current amount of tokens available on the sensor node.
- *initNrOfTokens* is the predefined number of tokens that initially are available on the sensor node.
- *timeout* is the amount of milliseconds before a beacon has to be transmitted.

Figure 3.3 shows the flow chart of the GCP protocol. The GCP protocol starts with an initialization of the local variables *version*, *initNrOfTokens*, *token* and *timeout*. With the local variables initialized the protocol continues to the **Idle** state. This is the main state of the protocol where the node listens for incoming beacons and update

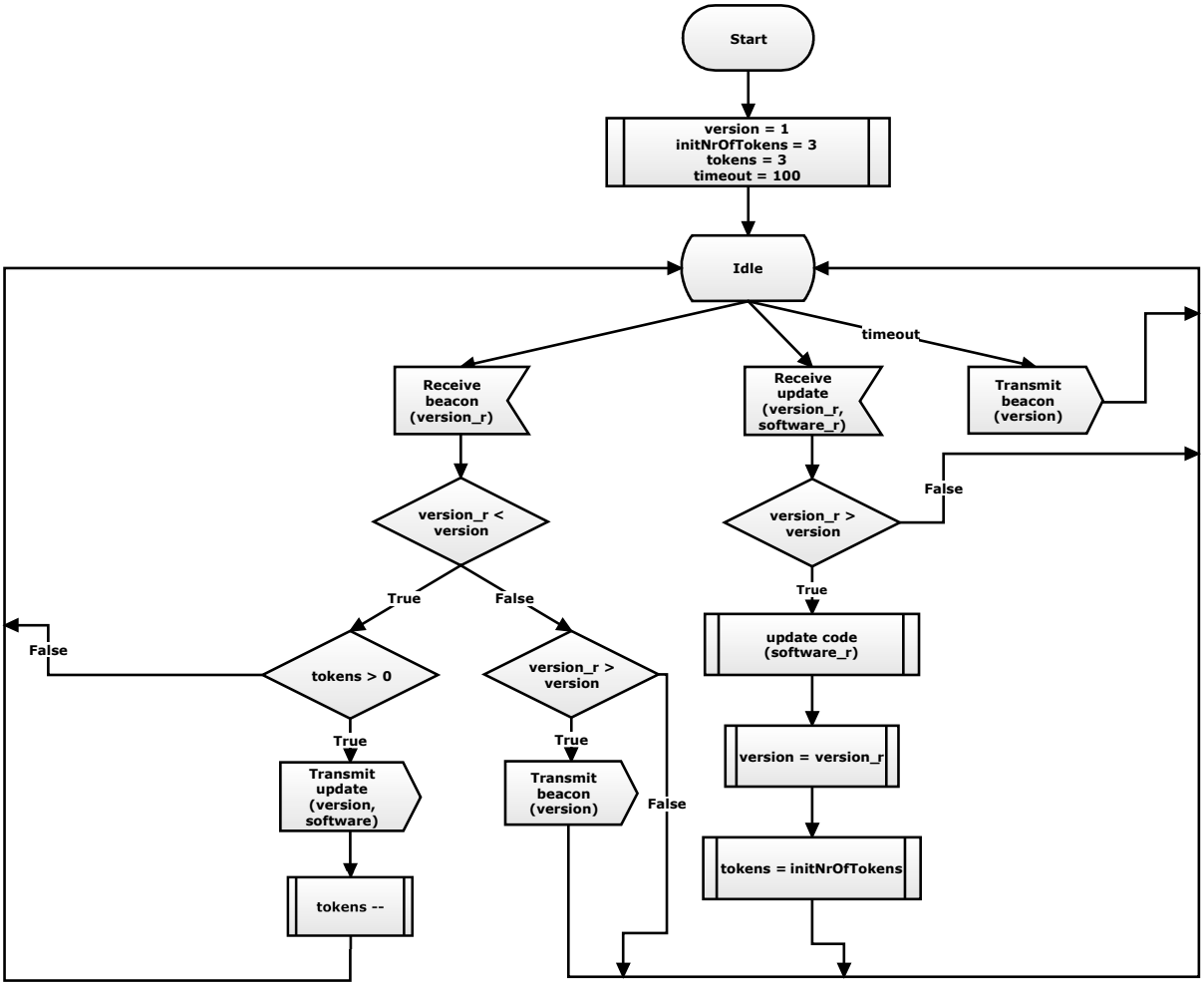


Figure 3.3: The flow chart of the GCP protocol.

messages while periodically transmitting a beacon (this is illustrates with a time-out period on the arrow from **Idle** to **Transmit beacon**). While the node is in the **Idle** state and periodically sending beacons, the protocol listens for two kinds of messages - **Receive beacon** and **Receive update**. Starting with **Receive beacon** the protocol first compares the received version with the one running on the node. If the received version is of a newer edition, the node will do a **Transmit beacon** before going back the the **Idle** state. The transmission of the beacon ensures that update messages will be sent to the node. On the other hand, if the software version on the node is newer than the received, the node will look at its *tokens* count. If that value is greater than zero, it will do a **Transmit update**, decrement the *tokens* variable with one and return to the **Idle** state. On the other hand, if the *tokens* variable is zero, the node is not able to start an update and thus return to the **Idle** state. The second message type the node could receive in the **Idle** state was an **Receive update** message. When the node receives this message it will compare the version with its own. If the received version is older , the node will send an update message, otherwise it will return to the **Idle** state.

Reflection on the GCP protocol

From the construction of the flowchart different constraints of the GCP protocol becomes more clear. Regarding the features of the radio no explicit assumptions are given in [9]. But unless the radio is full duplex situations will occur where the radio is busy receiving/transmitting updates and thus cannot transmit the periodical beacon. Which action the authors takes in this situation is unclear. Another unclear situation is what happens if a node crashes in the middle of a transmission/reception or while updating the running software. No explicit assumptions are found within [9]. The same applies if the node should recover after a crash. Lastly, assuming no packet loss / collisions in a wireless network is very limited and does not reflect the reality which makes this protocol less suitable for real implementation.

Summary

In this chapter we give a systematic specification of the GCP protocol. We start by defining the service the protocol delivers. We then describe the different assumptions upon which the protocol is built. After this a vocabulary was specified and the message encoding was presented. Lastly, the procedure rules were created with help from flow charts. The last section contains a short discussion about different ambiguous specifications and certain limitations in the protocol.

Verification of the GCP protocol

In this chapter we model the GCP protocol shown in Figure 3.3 as a timed automaton in UPPAAL. UPPAAL is a tool for modeling, validation and verification of real-time systems modeled as networks of timed automata [23]. Using UPPAAL we verify that our GCP model satisfy selected correctness properties. We ensure that the GCP protocol is free of deadlocks. We also verify the general update behavior of the GCP model, including correct tokens usage e.g the possibility of updating multiple nodes using only one token as specified in [9] We further extend the UPPAAL model with time, and energy, allowing us to reason about best- and worst case code update propagation speed as well as energy consumption for selected topologies.

4.1 Initial GCP model

The initial model of a UPPAAL node process implementing the GCP protocol is illustrated in Figure 4.1. The model reflects the mechanics of the GCP protocol modeled as a timed automaton, with no assumptions about the underlying hardware.

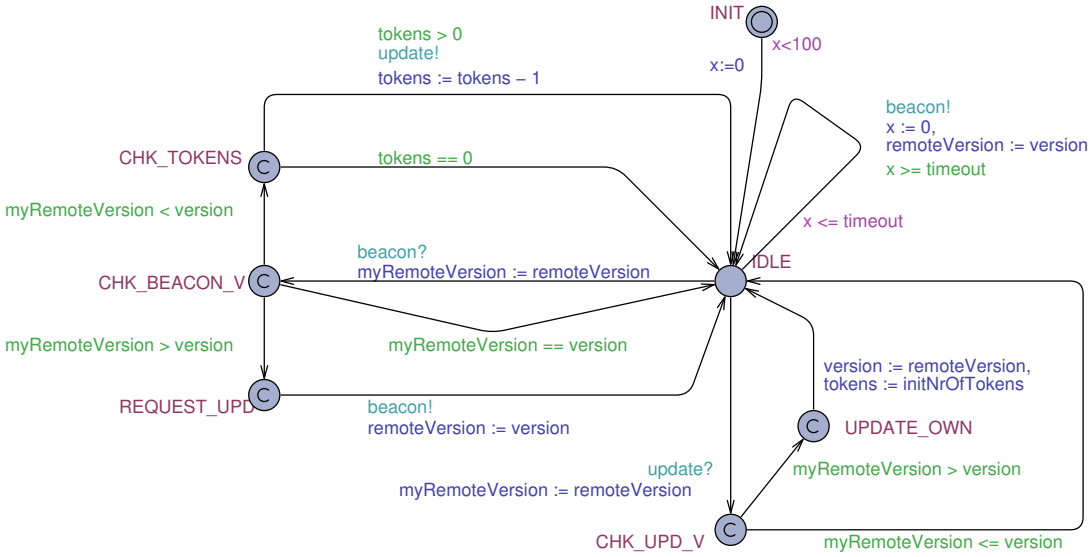


Figure 4.1: Initial GCP UPPAAL model

To explain the behavior of the initial GCP UPPAAL model we describe the individual states of the model.

INIT - The node process starts in INIT state, here it will remain between 0 to 100 time units, before it enters the IDLE state. We use the local clock x to keep track of time. Having the invariant $x < 100$ in the initial state forces the process to leave the initial state after 100 time units. By delaying 0 to 100 time units from the initial state to the IDLE state, we model that nodes have randomly starting beacon intervals.

IDLE In the IDLE state beacons will be broadcast periodically every 100 time units. We enforce this by having the invariant $x \leq timeout$ in the IDLE state, and by placing the guard $x \geq timeout$ on the *beacon!* transition, where $timeout = 100$. We reset clock x when a beacon is broadcast from IDLE state. When broadcasting the *beacon!* the node perform a *one-way* value passing of the update *version* that this node is currently holding. The *version* is passed to trough the shared variable *remoteVersion*

In the IDLE state it is possible to receive beacons and updates. The node can receive a beacon by synchronizing upon the *beacon?* broadcast channel and receive an update by synchronizing upon the *update?* broadcast channel. The update version for both a beacon and an update is passed trough the shared variable *remoteVersion* and stored in the local variable *myRemoteVersion* upon synchronization. When receiving a beacon the node will enter the CHK_BEACON_V state. When receiving an update the node will enter the CHK_UPD_V state.

CHK_BEACON_V The node check if the remote update is newer, older or the same version as the update the node is currently holding. If the remote version is newer than the current version, the node proceeds to the REQUEST_UPD state. If the remote version is the same as the current version the node returns to IDLE state. If the remote version is older than the current version the node will enter the CHK_TOKENS state.

CHK_TOKENS The node checks if it has enough tokens to broadcast an update. If the node has more than zero tokens it will transmit the update *update!*, decrease its number of tokens and enter IDLE state.

REQUEST_UPD The node request a new update by broadcasting a beacon with the local update version.

CHK_UPD_V The node checks the version of the received update. If the version is older or same as the current version we ignore it and enter the IDLE state. If the update version is newer than the current version the node enter the UPDATE_OWN state

UPD_OWN The node stores the received update by assigning the *remoteVersion* to the local variable *version*. The number of tokens are assigned to the initial token value.

The initial model introduced in 4.1 does not reflect the underlying hardware we there for proceed to extend the model with some assumptions about the radio used for communication.

model assumptions

We assume that we use a half-duplex radio to send and receive messages, and assume that the radio is always listening unless it is transmitting.

modeling

To model that we use a half-duplex radio, and that the radio is always listening unless its transmitting we need to ensure that after the node process performs a transmits

operation, *beacon!* or *update!* the node process is able to receive incoming packets, *beacons?* or *updates?*.

The intermediate states CHK_TOKENS and REQUEST_UP are removed. Transitions originating from these two state are accessible from the CHK_BEACON state. The check of update version of the beacon is enforced by placing guards on the transitions exiting the CHK_BEACON state. To enable update reception after a transmitted beacon, we add a transition from the CHK_BEACON to the CHECK_UPD_V state that syncronize upon *update?*. The changes made to the initial model is illustrated in Figure 4.2

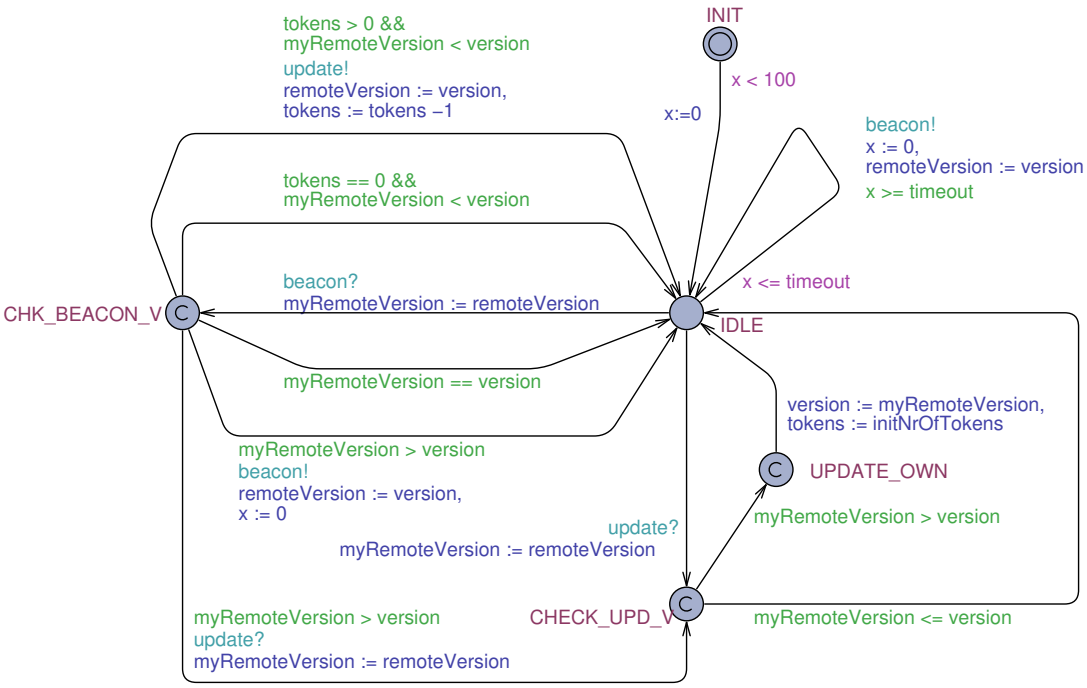


Figure 4.2: Basic GCP UPPAAL model

Verification results

The first property of the initial GCP model we verify is a safety property ensuring that the GCP protocol is free of deadlocks. We setup a system of system of three nodes with three tokens each. One of the nodes contain a new version of a code update. We ask the the UPPAAL verifier if a node will ever enter a state with no outgoing action transitions from the state itself, or any delayed successor states [23]. In UPPAAL the query i formulated as the Query in 4.1 Will it hold invariantly that we will not reach a state that is deadlocked.

$$A[]\ not\ deadlock \tag{4.1}$$

Another property we want to verify is that when ever we use a token, we are able to update at least one other node. Will it always hold that when N0 use a token, at least one other node is updated.

$$A <> (N0.tokens == 2)\ imply\ (N1.version == 2\ or\ N2.version == 2) \tag{4.2}$$

Another property of the GCP protocol that we wish to verify is that we are able to perform updates of multiple nodes using only one token. We query the UPPAAL verifier if it there exist a trace where node N0 use only one token to update the two nodes N1 and N2. As part of the query we ensure that N1 and N2 both got their update from N0 by requiring that the token number of both N1 and N3 is at the initial token value 3.

$$E <> (N0.tokens == 2) \ imply \ (N1.version == 2 \ and \ N1.tokens == 3) \ and \ (N2.version == 2 \ and \ N2.tokens == 3) \tag{4.3}$$

The Query 4.1, 4.2 and 4.3 all pass verification, and we ensured that our system is deadlock free, and performs updates according to specification.

4.2 Timing analysis

model assumptions

In our previous work [16] we treated three different components of a WSN software update scenario, that influence the code update propagation time and energy consumption, namely data optimization (DO), distribution protocol(DP) and execution environment (EE). We wish to examine the time and energy properties of the GCP protocol as a code update distribution protocol. Before defining a time and energy model we first make assumptions about both data optimization and the execution environment, as well as the underlying hardware.

Data optimization

For the time and energy models we use no data optimization. This means that the total software update is sent as a binary, as performed by the Deluge update mechanism [16][7] It can be argued that using differential patching (delta patching) [16][19] for data optimization will minimize the code footprint for simple updates, hereby conserving energy. As seen in [14] [19] the total amount of energy used applying a delta patch may vary greatly. The amount of energy used to apply a delta patch is not linear to the size of the delta patch, but also highly depends on the binary layout of the executable update as well as the memory type and structure on the target node [14]. The full binary update is chosen as it provides a more transparent code update footprint, to energy consumption ratio, when applying the patch at the target node.

Execution environment

For the models we assume that the EE is monolithic. Having a monolithic EE gives us efficient code execution that provides a strong correlation between the tasks involved in the update and the actual MCU time spend while updating. This ratio is otherwise distorted by the execution overhead of modular environments caused by the levels of indirection, or hardware abstraction in the virtual machine EE.

Code update size

As a basis for the UPPAAL model we use the platform previously explored in [16], comprising of the AT90CAN128 MCU [1] and the nRF24L01 radio [5]. In the AT90CAN128 there is 128 kB of programmable flash memory available, structured in 512 pages of

each 256 bytes. The flash memory space is divided into two sections, the boot loader section and application program section. The size of the bootloader section can be defined to either 4, 8, 16 or 32 pages, trough the BOOTZS1 and BOOTZS2 register of the AT90CAN128 MCU [1]. We reserve 16 pages of flash memory for the boot loader application, hereby leaving 496 pages of unused flash program memory. As we perform total binary updates, using no external ram source, we need to reserve half of the flash program memory as a buffer for the update before it can be written to the application area. The size of the buffer- and application area are both 248 pages allowing code updates of a maximum of $248 \times 256\ bytes = 62\ kB$. For the time and energy model we make the worst case assumption that the size of all code updates are 62 kB [2]. The flash memory layout is illustrated in Figure 4.3

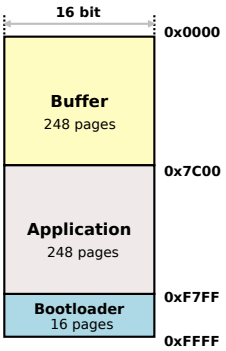


Figure 4.3: Flash memory layout
has

Update TX/RX time

We first calculate the time a node spends when transmitting or receiving an update. The radio is capable of transmitting $2 \times 10^6 bps$. We assume that we have no collisions and no lost packages as in [9]. To send a 62 kB update, we also assume that the MCU time reading data from memory is insignificant compared to the throughput of the radio, hence this will not influence the overall transmit time of the update. With the assumption that we have no collisions and no lost packages we calculate the time to send a 62 kB update in Equation 4.4.

$$TX\_Time = \frac{62\ kB \times 8\ bit \times 1024}{2 \times 10^6\ bits/s} = 0.253952 \approx \underline{0.254}\ seconds \tag{4.4}$$

With the aforementioned assumptions we can transfer a 62 kB update wirelessly in 0.254 seconds.

The time spend sending and receiving an update is not the same. The receiving node has additional processing to do by storing the update in buffer, and subsequently moving the update from buffer to the application area. In the follow we calculate the time it takes to store an update in buffer. The flash memory of the AT90CAN128 is structure in units of 256 byte pages [1]

Writing a single page in flash memory takes $4.5ms$ [1], to store a 62 kB update in buffer we need to write 248 pages. We calculate the time it takes to store the update in the buffer in Equation 4.5.

$$hasBufferWriteTime = 248\ pages \times 0.0045\ seconds = \underline{1.1160}\ seconds \tag{4.5}$$

The receiving node use 1.116 seconds to store the update in buffer. As the receiver is not able to store the data as fast as the transmitter can send it, the transmitting node is also involved in the update for this period of time. Even though the transmitting node is involved in this period of time, the radio is not always fully active during the update. We use the 1.116 seconds as the time a transmitting node use to send an update. As a basis for energy consumption we later use the 0.254 seconds as a rough estimate for the active period the sending node is actually transmitting. We assume that the MCU is active during the entire update for both the sending and the receiving node.

When the receiving node has stored the update in the buffer, it needs to copy the update from the buffer to the application area. This write operation will consume an additional 1.116 seconds and in addition to this we must add the time it takes to read the 248 pages from the buffer.
The read time of a full 256 byte page is not explicit in the AT90CAN128 data sheet, we therefor derive an estimate on the total buffer read time, based on the AT90CAN128 MCU architecture, the MCU frequency and a pseudo code example found in the AT90CAN128 data sheet, dictating a 'byte by byte' flash read operation.

The flash memory is read one byte at a time [3]. The pseudo code in Listings 4.1 describes how to read first the low byte, then the high byte of a specific 16 bit flash address. This read operation consist of six instructions. As both high- and low bytes are stored in DATA register we add two move instructions, for at total of 8 instructions as an estimate for the operations performed reading two bytes from the buffer.

```
1 Load Command 0x02. // ReadFlash command
  Load Address High Byte (0x00   0xFF).
  Load Address Low Byte (0x00   0xFF).
  // Output Enable =0, ByteSelect1 =0 (Addr. word low byte now in DATA.)
5 Set !OE to 0, and BS1 to 0
  //  ByteSelect1 = 1 (Addr. word high byte now in DATA.)
  Set BS1 to 1
  Set !OE to 1. // Output Enable = 1
```

Listing 4.1: Buffer read operation

The AT90CAN128 MCU is of Harvard architecture, this means, that the MCU utilizes a separate instruction- and data bus, allowing it fetch instruction, while executing. The AT90CAN128 MCU can hereby achieve speeds of up to 1 MIPS per MHz on operations on the registers R0 - R31. Loads and store operations to and from memory use 1-2 clock-cycles and branch operations use 1-4 clock cycles [1]. The AT90CAN128 is running at 16MHz. All eight instructions use 2 clock cycles each [1][3], and we can calculate the time to read a 248 pages of 256 bytes in Equation 4.6

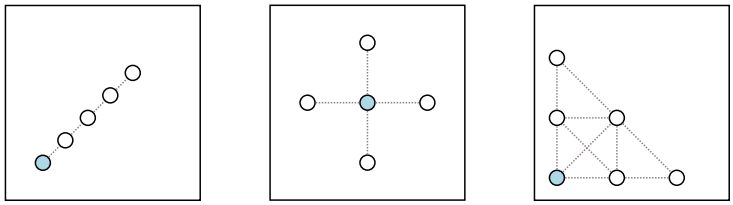
$$BufferReadTime = \frac{8 \times 2 \ cycles}{16 \times 10^6 \ cycles/sec} \times \frac{248 \ pages \times 256 \ bytes}{2 \ bytes} = \underline{0.031744} \ seconds \tag{4.6}$$

The total update time for the receiving node is calculated in Equation 4.7.

$$UpdateTime_{Receiver} = 2 \times 1.116 \ sec. + 0.031744 \ sec. = \underline{2.2637} \ seconds \tag{4.7}$$

Modeling

For time analysis we extend the basis GCP UPPAAL model in Appendix B.2 with the notion of transmission time, and connectivity. Using the extended GCP time analysis model Appendix B.2 we wish to show the best- and worst case code propagation speed for the synthetic topologies illustrated in Figure 4.4



(a) Straight line. (b) Center node. (c) Corner node.

Figure 4.4: Synthetic topologies

The synthetic topologies in 4.4 each reflect a specific placement of an updated node in relational to a multiple number of non-updated nodes. Each topology will influence the code update propagation speed as well as the use of tokens in its own way. Using UPPAAL we reason about the best- and worst case code update propagations speed.

Connectivity

To do time analysis using the synthetic topologies in Figure 4.4 , we introduce the notion of connectivity to the GCP UPPAAL model.

We use a simple Beacon process illustrated in Figure 4.5 to explain how connectivity is modeled. The simple Beacon process can send and receive beacons. The Beacon process counts the number of sent and received beacons using the variables *beaconsSent* and *beaconsRecv*. The Beacon process will send a beacon every 100 time units.

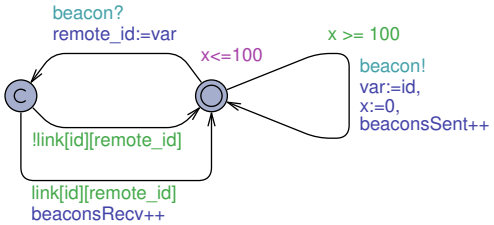


Figure 4.5: Beacon process

When synchronizing on the *beacon* channel we perform a one-way value passing of the sending node *id*. The sending node passes its *id* to the receiving node trough the shared variable *var*. Synchronization at the receiving node will lead to a new intermediate committed state, from where we will decide if there is connectivity between the sending an receiving node. The topology of the network is modeled as the adjacency matrix *Link* in Table 4.1. To determine if the sending- and receiving nodes can communicate, the receiving node perform a lookup in the *Link* matrix using its own *id*, and the id of the sending node, *remote_id*. If $Link[id][remoteId] = 1$, the node identified by *id* receives messages sent by the node with *remote_id*. The *Link* evaluation is placed as a guard upon the transition indicating that the nodes are connected, in this case incrementing the number of beacons received, *beaconCount*. The negated *Link* evaluation is placed

as a guard on the transition leading back to the initial state hereby ignoring the effects of the *beacon* synchronization as is if the two nodes were not connected. We test the connectivity by verifying a UPPAAL *system* of two Beacon processes $P0_{id=0}$ and $P1_{id=1}$, using the connectivity array *Link* in Table 4.1 first in connected configuration, and next in a non-connected configuration.

Connected	Non-Connected
$Link = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$	$Link = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$

Table 4.1: Connected vs non-connected *Link* configuration

To verify that process P0 and P1 can communicate in the connected *Link* configuration we first query the UPPAAL verifier if it holds that when P0 sends a beacon, it will imply that this beacon is received by P1.

$$E <> (P0.beaconsSent > 0) \ imply \ (P0.beaconsSent == P1.beaconsRecv) \tag{4.8}$$

To ensure two way communication it must also hold that if P1 sends beacons, the number beacons sent by P1 must be equal to the number of beacons received by P0.

$$E <> (P1.beaconsSent > 0) \ imply \ (P1.beaconsSent == P0.beaconsRecv) \tag{4.9}$$

To verify that process P0 and P1 can not communicate in the non-connected *Link* configuration, we ask the UPPAAL verifier if it holds invariantly that the *beaconsRecvt* variable of both P0 an P1 always will be zero, meaning that communication never occurs.

$$A[](P0.beaconRecv == 0) \ and \ (P1.beaconRecv == 0) \tag{4.10}$$

All 3 queries pass verification, and we can apply connectivity as illustrated in Figure 4.5 to the GCP model in Figure 4.6. (A large scale Figure can be found in Appendix B.3)

Time

To reflect the passing of time in the model we introduce two new states SEND_UPD and RECV_UDP.

SEND_UPD In this state the node will delay for the period of time it takes to send an update. The time it takes to send an update is stored in the constant SEND_TIME. The invariant $x \leq SEND\_TIME$ in the SEND_UPD state enforces the process to leave the state after SEND_TIME time units. The guard $x \geq SEND\_TIME$ placed on the transition leading to the IDLE state enforces that the node cannot take this transition sooner than $SEND\_TIME$ time units. The node process remains in the SEND_UPD state exactly $SEND\_TIME$ time units, and proceeds to the IDLE state.

RECV_UDP In this state the node will delay for the period of time it takes to receive an update. The time it takes to receive an update is stored in the constant RECV_TIME. Using a combination of an invariant and a guard we ensure that the node remains in the RECV_UPD state for the duration of $RECV\_TIME$ time units, and then proceeds to the IDLE state.

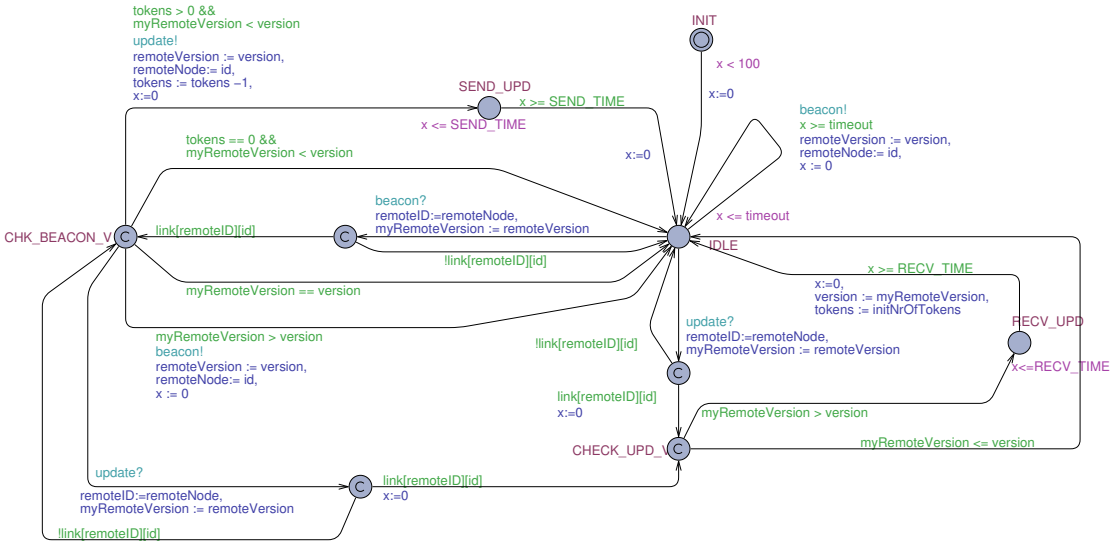


Figure 4.6: GCP UPPAAL time model

Verification results

Best-case code propagation speed

With connectivity and time as a part of the GCP UPPAAL model we can start to reason about the time characteristics of the GCP code propagation speed in the synthetic topologies in Figure 4.4 To retrieve the best-case code update propagation speed, we query UPPAAL to generate the *fastest trace* where all nodes in the system are in an updated state. We add the global clock g to the system as time keeper. When the *fastest trace* is found, clock g will yield the time duration for the best-case code update propagation speed.

$$E <> N0.version == 2 \ and \ N1.version == 2 \ and \ N2.version == 2 \ and \ N3.version == 2 \ and \ N4.version == 2 \ and \ N5.version == 5 \tag{4.11}$$

Worst-case code propagation speed

Determining the worst case code propagation time in UPPAAL and in general is not straight forward task. We cannot ask UPPAAL for the for exact time for the worst case code propagation speed, instead we manually narrow down this number by doing a binary search asking "is it possible to have a non-updated system after a period *timeBound* time units". Where we initially set *timeBound* to relatively high number and half this number if the query fails. The binary search is illustrated in Figure 4.7

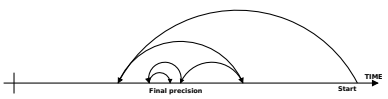


Figure 4.7: Binary search for worst case code propagation speed

We formulate the initial query to obtain wost cast propagation speed in Query 4.12.

$$E <> \ not\ (N0.version == 2\ and\ N1.version == 2\ and\ N2.version == 2\ and\ N3.version == 2 and\ N4.version == 2)\ and\ g >\text{timeBound} \tag{4.12}$$

If a query pass on a specific *timebound* it means that there are still non-updated nodes within the *timeBound*, and we increase the *time* bound an query the UPPAAL verifier again.

The time analysis result for best- and worst case code propagation speed is summarized in Table 4.2

Topology	Best-case CPS	Worst-case CPS
$A_{Straigh\_Line}$	9.164 sec.	$\approx$ 14.07 sec.
B_{Center_Node}	2.416 sec.	$\approx$ 10.26 sec.
C_{Corner_Node}	4.662 sec.	$\approx$ 15.63 sec

Table 4.2: Best- and Worst case time analysis

As expected we see that best case code propagation is achieved in the center node scenario, where all node can be updated using only one token. The slowest best-case propagation is not surprisingly when all nodes are aligned, and each node must pass on the update. The fastest worst-case propagation is the center node scenario, while the slowest being the corner node scenario.

4.3 Energy analysis

model assumptions

We calculate the energy consumed while sending and receiving updates, this includes energy consumption by the transmitter/receiver as well as the MCU. We assume that the MCU is active while sending and receiving, and that the MCU is in idle state between the periodical beacons.

We use the following typical power consumption characteristics for the nRF24L01 and the AT90CAN128 in Table 4.3 and 4.3

Operational mode	Supply current
RX (Low current mode) @ 3.3V	11.5 mA
TX @ 3.3V, -18 dBm 2000kpbs	7 mA
IDLE	22 μA

Table 4.3: nRF24L01 supply current

Operational mode	Supply current
Active mode @ 25 °C, 4.5V_{cc}, 16MHz	27 mA
IDLE mode @ 25 °C, 4.5V_{cc}, 16MHz	16.5 mA

Table 4.4: AT90CAN128 supply current

We first use Equation 4.13 to calculate the energy consumption J. J is the quantity of energy in joules used for a given operation transmit,/receive of an update, and moving

update code from buffer. V is the supply voltage, I is the supply current. s is the time in seconds.

$$J = V * I * s \tag{4.13}$$

The energy consumption of the nRF24L01 transmitting and receiving an update is described in Equation 4.14 and 4.15 using the typical power characteristics in Table 4.3 and the transmit time from Equation 4.4.

$$J_{nRF24L01\_TX} = 3.3V \times 0.007A \times 0.254s = 5.866 \times 10^{-3} \approx \underline{5.87mJ} \tag{4.14}$$

$$J_{nRF24L01\_RX} = 3.3V \times 0.0115A \times 0.254s = 9.637 \times 10^{-3} \approx \underline{9.64mJ} \tag{4.15}$$

The energy consumption of the AT90CAN128 while reading the buffer using the typical power characteristics in Table 4.4 and the buffer read time as calculated in Equation 4.6

$$J_{ReadBuffer} = 4.5V \times 0.027A \times 0.031744 = \underline{3.86mJ} \tag{4.16}$$

The energy consumption of the AT90CAN128 while writing an update to buffer using the typical power characteristics in Table 4.4 and the buffer write time as calculated in Equation 4.5.

$$J_{WriteBuffer} = 4.5V \times 0.027A \times (248pages \times 0.0045s) = 0.13559400 = \underline{135.59mJ} \tag{4.17}$$

Sender/Receiver	Operations	Energy
Energy_SEND	$J_{nRF24L01\_TX} + J_{WriteBuffer}$	141.5 mJ
Energy_RECV	$J_{nRF24L01\_RX} + 2 \times J_{WriteBuffer} + J_{ReadBuffer}$	284.8 mJ

Battery supply

We assume that a node is powered by three serially connected AA batteries. A single AA battery characteristic is 1.5 volts, and 1100 maH. We assume that the voltage of the battery is 1.5 volts during the entire discharge of the battery. We calculate the energy in three AA batteries in Equation 4.18

$$BatteryEnergy = \frac{1100\ maH\ * 3600\ sec.\ * 1.5\ volts}{1000} \times 3\ batteries = \underline{17820}Joules \tag{4.18}$$

Assuming that a node is continuously receiving updates performing no other actions, a is theoretically possible to receive $\frac{17820J \times 1000}{248.8\ mJ} \approx \underline{62596}$ updates. This is highly optimistic and will exceed the maximum of flash write/erase cycles. It should also be noted that the battery voltage eventually will drop below the operational voltage threshold for the AT90CAN128 during the discharge of the battery.

Modeling

To model energy consumption we introduce the meta variables *energy* and *totalEnergy* . In *energy* we accumulated the energy consumption of the individual node, in *totalEnergy* we accumulated the energy consumption of they entire system. When synchronizing upon the *update*! broadcast channel we add the energy consumption for a transmitted update to both the *energy* and *totalEnergy* variable. When receiving an update the energy consumption for a received update is added to *energy* and *totalEnergy*.

To capture the time spend by a node in the IDLE state, we introduce a new process Ticker, and add a looping transition in the IDLE state that synchronize on the *tick* broadcast channel. The role of the Ticker process is to "active" the node process in IDLE state. Ticker will broadcast a *tick*! each time unit, and allow we can hereby accumulate the energy consumed in IDLE state When the node process synchronize upon *tick*? we add the power consumed in one time unit (milisecond) to *energy* and *totalEnergy*. The Ticker process is illustrated in Figure 4.8

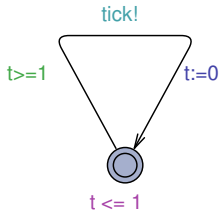


Figure 4.8: Ticker process

The energy extended GCP UPPAAL model is illustrated in Figure 4.9

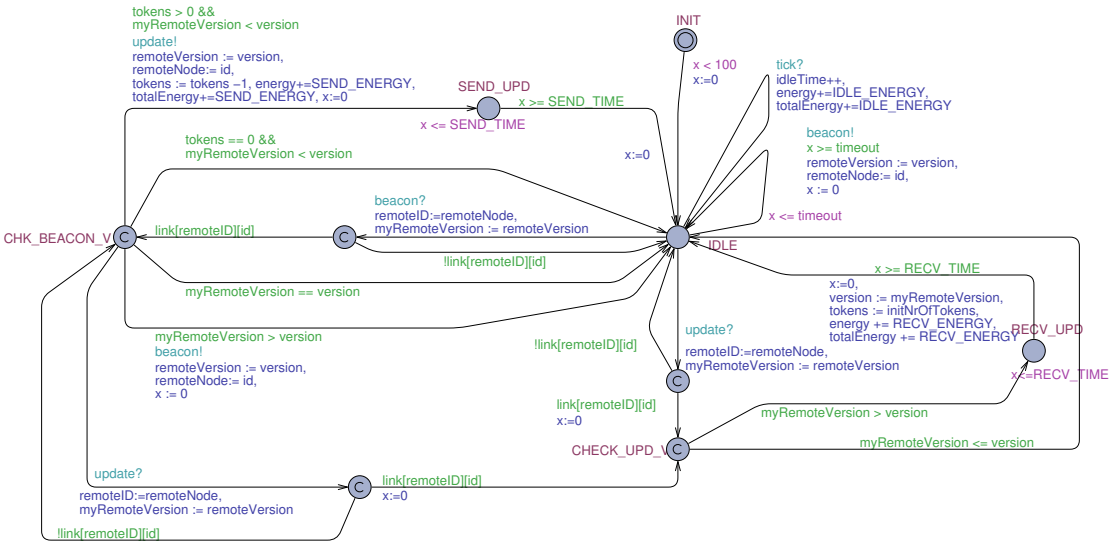


Figure 4.9: GCP UPPAAL energy model

Verification results

Having the energy consumption reflected in the model we want to explore the worst case energy consumption within a given time bound.

We ask the UPPAAL verifier if there exist at trace where we have used more than a specified energy threshold, we denote *EnergyLevel* within a certain time bound that we denote *TimeBound*. We use clock g as a timer keeper for the global time, and *totalEnergy* as the amount of energy consumed by entire system.

$$E <> (g < TimeBound) \ and \ (totalEnergy > EnergyLevel) \tag{4.19}$$

If the Query in 4.19 is passed, we know that the system is able spend more energy within the time bound, otherwise the system will always have energy within the time frame. We want to find the highest value of *EnergyLevel* for which it holds that Query 4.19 is satisfied. We perform a binary search to find the worst case energy consumption bounded by the best- and worst-case code update propagation speed.

The task of searching the state space for the worst case energy consumption became extensively time consuming and we did not manage to produce comparable worst case energy results for the chosen synthetic topologies, bounded by the best- and worst case propagation speed. We acknowledge that UPPAAL CORA performs Cost Optimal Reachability Analysis and is the proper tool to use in search of worst case energy consumption but we will have to leave this task for future work.

4.4 Summary

Through the verification chapter we presented simple mode to illustrate the mechanics of the GCP model. We extend the initial UPPAAL GCP model in three stages, first incorporating the assumption about the radio, and then time and energy use. We made assumptions about the underlying hardware of the model, and calculated the time and energy used during an update for both the sending and receiving node. We verified that the GCP protocol is free of deadlocks, and that the intended token uses is reflected in the model. On the time extended GCP model we performed time analysis and find best- and worst case code propagation speed for three synthetic topologies. On the energy extended GCP model we performed energy analysis but were not able to generate results.

Simulation of GCP

This chapter introduces a discrete event simulator named Network Simulator 2 (NS2)[6] [22]. In this tool we implement the GCP protocol and verify it behaves as we expect. Then the simulation scenarios are described so various tests can be conducted. The results will be compared with the original results presented in [9]. Finally, explanations are given for the deviation and replications of the original results are created.

5.1 Introduction to NS2

The NS2 tool is a packet-level network simulator developed at DARPA VINT [1] as a project collaboration between researchers at UC Berkeley, LBL, USC/ISI and Xerox PARC. It is a free open source tool which helps to simulate existing network protocols: TCP, UDP, routing, and multicast over both wired and wireless networks. The tool is written using to languages:

- C++ defines the internal mechanism, and is responsible for running the entire simulation. Using this language we define the functionality of the protocol.
- Object-oriented Tool Command Line(OTCL) is used to set up the parameters for the simulation. Such parameters could be defining the topology, configuring network objects, and scheduling discrete events.

This separation makes the tool flexible, because in order to change the system parameters, it is enough to change the simulation file. Hence it removes the necessity to recompile the entire code. Since the body of NS2 is fairly large, the compilation time is not to be negligible.

Figure 5.1 shows how the NS2 tool works. Firstly, the OTCL interpreter processes a script. Secondly, it creates objects that are linked to a shadowing objects in a compiled code, adds events to the EventQueue, and runs the simulation. Thirdly, when the simulation finishes, the results can be accessed using an OTCL interpreter and visualized graphically using the tool *Network AniMator* (NAM).

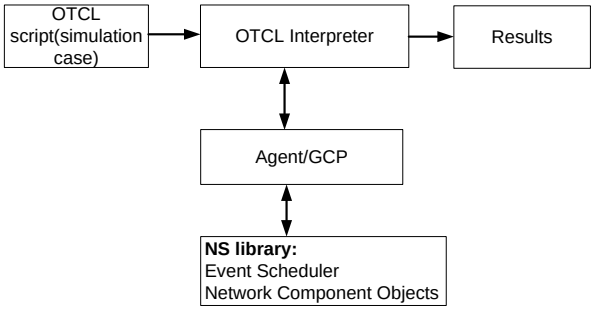


Figure 5.1: NS2 working principles

Event scheduler

The NS2 performs centric discrete event simulations. Discrete event simulation organizes events in chronological order in a fitting data structure referred to as the EventQueue.

[1] Defense Advanced Research Projects Agency Virtual InterNet Testbed

Each event is described by the event time (when the event is handled), an unique id, a pointer to the next event, and the associated event handler object. Each event is processed one by one, and the associated "handler" object deals with this particular type of event, updating the state space of the simulation scenario accordingly to the queue and potentially spawns one or more events. These events are subsequently inserted into the EventQueue ordered by the event time before the next event is processed. When an event has been processed, the global clock "jumps" to the time stamp of the next event in the EventQueue, and the next event is processed.

An example is illustrated in Figure 5.2 with two nodes A and B. We assume the nodes are within communication range, and with no packet loss will not occur.

1. The earliest event from the EventQueue is node A sending a beacon packet to its neighbourhood. The event is handled by a network object, in this case sensor node A.

2. The handler for node A, will in turn call other networks objects affected by this particular event, in this case Node B.

3. When Node B has been made aware that it has received a beacon, it will generate a new event. The event generated will broadcast an update packet, assuming that the beacon version sent from Node A, is older that the version of Node B.

4. This new event is inserted into the EventQueue, and the next element of the EventQueue is processed.

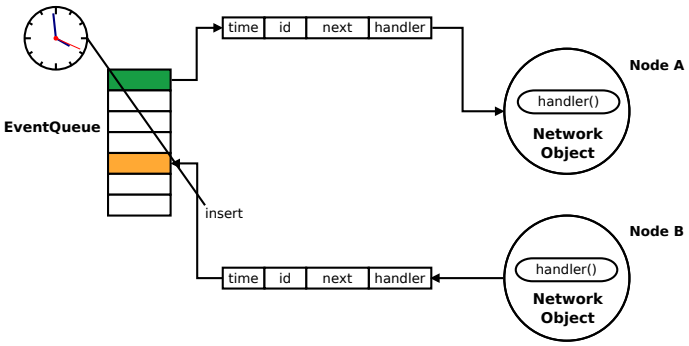


Figure 5.2: NS2 scheduling a beacon transmission from node A, and a replying update message from node B.

Network component classes

Basically, NS2 has two class hierarchies for both languages: a compiled hierarchy and interpreted hierarchy. These classes can be either standalone or linked together using a TCLCL interface. In Figure 5.3 we see a simplified C++ class hierarchy in NS2. All information and graphically presented documentation for NS2 can be found at [4]. To understand the class diagram of the NS2 tool in Figure 5.3 we first describe the labels used:

- *TCLObject.* All classes deriving from this class forms a compiled hierarchy.
- *Handler.* Each event contains a handler which specifies the action and the dispatching event.

- *NSObject* is a parent class for a network objects that are responsible for sending, receiving, creating, and destroying packet related objects.
- *Classifier* is responsible for forwarding the packet to several NSObjects. If, for instance, a mobile node broadcasts a packet, then a classifier object forwards this packet according to the headers to all nodes within range.
- *Connector* immediately forwards the received packet from one NSObject to another one.
- *Application* is a parent class of applications such as FTP, Telnet, Traffic Generator and others.
- *Agent* is an endpoint where packets are constructed or consumed. It is used in implementation of protocols, i.e, UDP, TCP, GCP, and others.
- *Queue* This class derives from the Connector class. The object of this class processes outgoing packets. It gives a representation of locations where packets may be kept or dropped. The process responsible for packets scheduling makes a decision which packet should be serviced or dropped. There is also a buffer management process that regulates the occupancy of a particular queue type. For instance, drop-tail(known as FIFO) or fair-queueing(FQ).
- *T*he Packet class defines the type for all packets in the simulation and provides member functions to handle any object by its type.
- *Node* itself is a stand-alone class. It can either act as a router to forward a packet, or as host to hand a packet to the agent attached to the specified port.

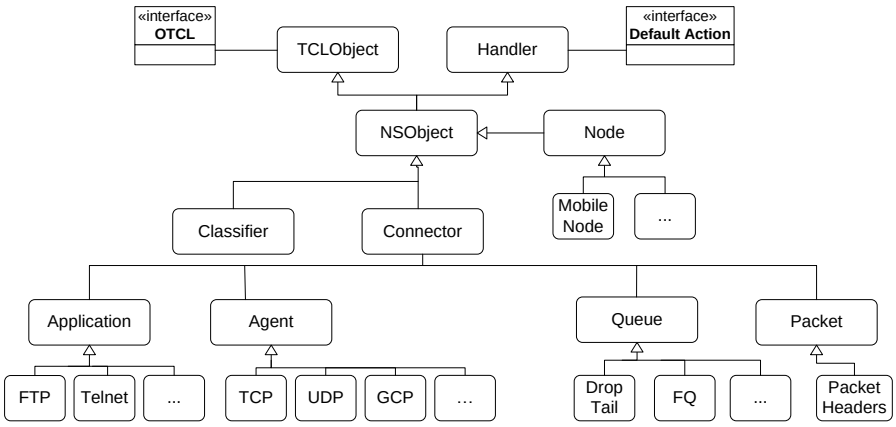


Figure 5.3: A simplified C++ class diagram of NS2

Mobile networking in NS2

The *Wireless model* is an extension to the NS2 tool and was included by the CMU Monarch Project [17]. The extension provides new additional elements at the physical, link, and routing layers of the simulation environment. With this extension we can create an accurate simulation of either a wireless- or multihop ad hoc networks.

The mobile node may have one or more network interfaces attached to a channel. These channel carries packets between the mobile nodes. When a packet is put onto the

channel, it distributes a copy of the packet to all the network interfaces of that channel. The interfaces then use the *radio propagation model* [18] to make a decision of accepting or dropping the packet. A basic schematic of a mobile node is shown in Figure 5.4. In the following we explain outgoing and incoming packets:

Outgoing Packets. When a packet arrives at the mobile node's entry, it is passed and processed by a demultiplexer. If the destination IP address matches the mobile node's IP address then the packet is passed to the port demultiplexer, which hands the packet to the respective destination agents. However, most packets will match the default target and be handed down to the routing protocol. It adds an IP address of the next mobile node, which has to process the packet. The ARP table is used to resolve the logical IP address with second layer physical address. For simplicity, IP addresses are reused at the MAC layer. The next step is to insert the packet into the interface queue(IFq). After this, the MAC object takes the packet from the top of IFq and sends it to the networking interface(NetIF). Lastly, NetIF stamps the packet with attributes such as the power and position of the transmitting interface and puts the copy of the packet to the channel. The channel makes a copy to all other interfaces connected to that channel.

Incoming Packets. A copy of each packet sent onto a channel is delivered to all network interfaces at the time at which the first bit would be delivered at the real system. Each NetIF puts a mark on the packet with the receiving network interface attributes and invokes Radio Propagation Model(RPM) object. This object uses the transmitted and received stamps of network interface to calculate the power with which the packet is received. The NetIF uses the result to determine if it actually successfully received the packet and hands the packet to MAC object. If the packet is error and collision free, it is passed to the mobile node's entry point. At this point a decision is made whether the packet has to be routed to the next mobile node, or it has reached its destination.

5.2 Implementation of GCP

The core elements like protocols, packet level content and agent behaviour are implemented in C++. To test the GCP in NS2 we must implement GCP packet content and a GCP Agent in C++ which defines the behaviour of the sensor node. We are using the current release of NS 2.34. To begin we create a new directory 'gcp' in the NS base directory in which we create the following two files that will make up GCP implementation:

- gcp.cc which describes the behaviour of the entire protocol.
- gcp.h is a header file for gcp.cc file.

A class diagram of GCP protocol

In Figure 5.5 the class diagram of GCP is shown. It presents the objects that are used by a GCP protocol. If the fields (methods or parameters) of the classes are empty, it means that we did not modify this class.

The following explains the the most interesting methods in the **GCPAgent** class that are used directly for implementation of the GCP protocol. This class defines the behaviour of GCP protocol. A description of the methods is given in the following:

- **recv**. It is used for getting, processing, and sending a packet out.

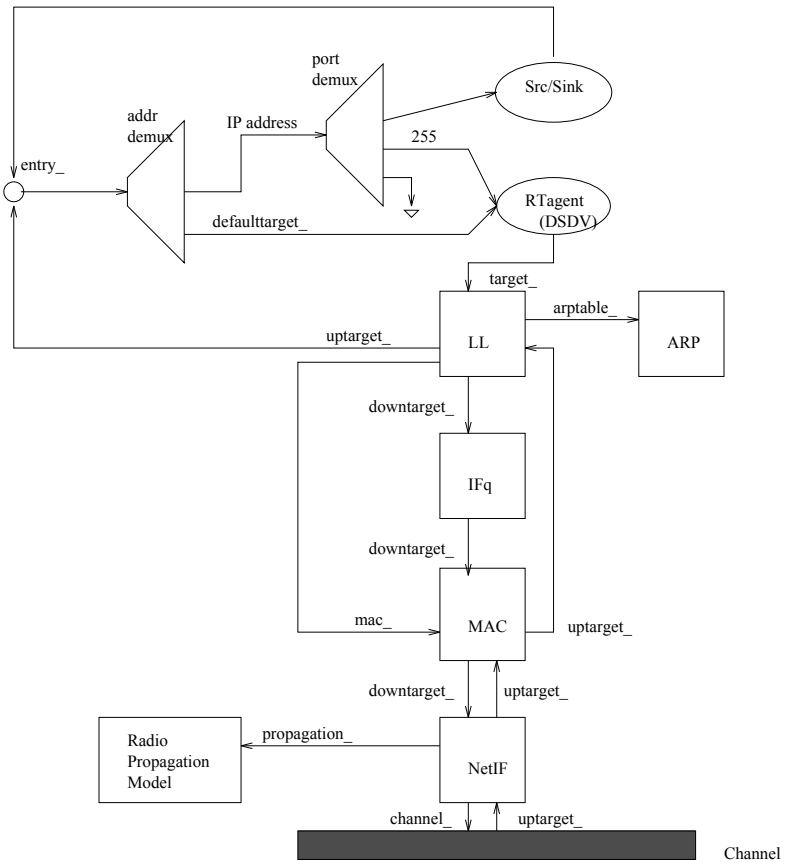


Figure 5.4: Schematic layout of a mobile node [6]

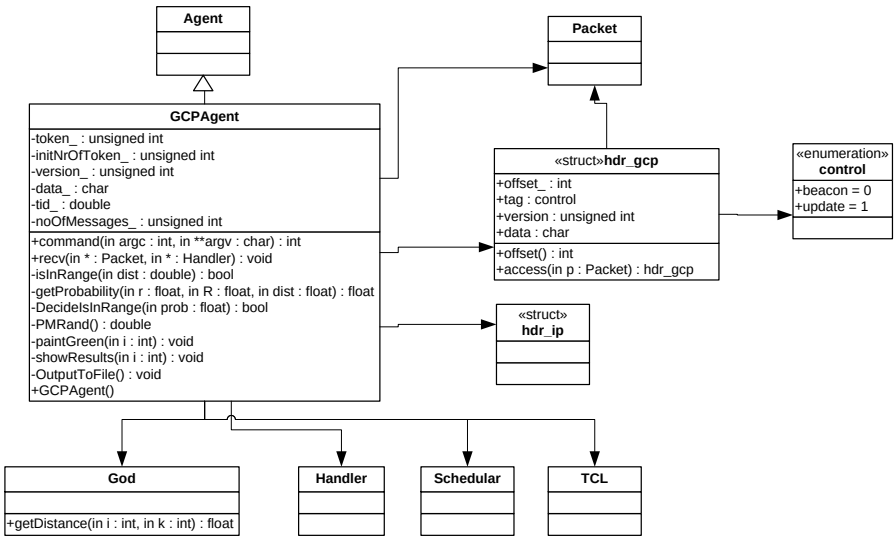


Figure 5.5: Class diagram of GCP protocol

- **isInRange**. Make a decision whether the packet will be forwarded for processing or discarded

- **getProbability**. Count the probability of getting the packet, when it is between 3m and 5m

- **DecideIsInRange**. Check if the received packet is in a specified range.
- **PMRand**. Return a random number from 0 to 1.
- **paintGreen**. The method invokes TCL object and binds it with C++ in order to change node colour at the time when node gets an update.

One of the most used methods is our receive method **recv**. This method describes the main loop of the protocol.

The recv method

When a node receives a packet it unfolds it and takes the necessary data. In our case, we need to access the headers of the ip and gcp protocols. After this, we count the distance between the sending and receiving nodes.

Every time a packet is received, the header is accessed, the distance is calculated and passed on to the isitReachable method. If the node is within range we accept it, else its dropped. If a packet is accepted, we check whether its a beacon or an update message. These two checks are described with code in the listing 5.1 and 5.2:

- *A beacon received.* The node compares the received version and its own. If the node has an older version, it constructs a new packet of a beacon type and include the version number where after the packet is transmitted.

```
recv(Packet *pkt, Handler*)
...
    if(hdr->version > version_)
      {
          Packet::free(pkt);          //free memory
          Packet* pktret = allocpkt(); // alloc space for next packet

          hdr_gcp* hdrret = hdr_gcp::access(pktret);
          hdr_ip* ipret = hdr_ip::access(pktret);

          ipret->daddr() = IP_BROADCAST;  //add broadcast add
          ipret->dport() = ipret->sport();  //add source port

          hdrret->tag = beacon;
          hdrret->version = version_;

          send(pktret, 0);
      }
...
```

Listing 5.1: Code example of receiving a newer version.

The code for receiving a lower version differs with few lines: the node adds an update tag instead of the beacon to the message together with the version number. After transmission, the token count is decremented.

- *An update received.* The node compares the received version and its own. If the version is older than its own, it simply discards it. Otherwise, the node saves the time stamp when it received the update, saves the update version, re-initializes its number of tokens, invokes a paint function which adds the event to the EventQueue. Thus a node paints itself with a green colour(we use this for visualization), and outputs the time and id to a file.

```
1       if(hdr->tag == update)
        {
            if(hdr->version > version_)
            {
5               tid_ = CURRENT_TIME;

                version_ = hdr->version;
                //reset tokens when we get update
9               token_ = initNrOfToken_;

                paintGreen(here_.addr_);
                OutputToFile();
13          }
            Packet::free(pkt);
        }
```

Listing 5.2: Code example of receiving a new update.

A description of the simulation file

The code listed in listing 5.3 demonstrates the part of the protocol, where the nodes send a beacon message every 100ms. Firstly, we chose a node randomly which will get the initial update. This node starts to send beacons after 1.000 seconds. After the transmission the node saves the time so it knows when to transmit the next beacon. The other nodes start transmitting their beacons between 1.0 and 1.1 seconds and also store their sending time.

```
1  set randStart [new RandomVariable/Uniform]
   $randStart set min_ 1.0
   $randStart set max_ 1.1
   set last_value 0
5
   set randNum [expr { int(($val(nn)   1) * rand()) }];#choose random node
   $ns_ at 0.0 "$p($randNum) set version_ 5"       ;#assign newer version
   $ns_ at 0.0 "$node_($randNum) color green"      ;#it becomes green
9  for {set ii 0} {$ii < $val(nn)  } {incr ii} {

     set startTime [$randStart value]              ;#starting time between 1.0s
                                                   ;#and 1.1s for all nodes
13   for {set i 0} {$i < [expr $val(stop)*10] } {incr i} {
       if {$ii ==  $randNum} {
         if {$i == 0} {
           set startTime 1.000                    ;# starting time for first node
17         $ns_ at $startTime "$p($randNum) broadcast-beacon"
           set last_value $startTime              ;#save the last time value
         } else {
           set startTime [ expr $last_value + 0.1]
21         $ns_ at $startTime "$p($randNum) broadcast-beacon"
           set last_value $startTime
         }
       } else {
25       if {$i == 0} {
           $ns_ at $startTime "$p($ii) broadcast-beacon"
           set last_value $startTime
         } else {
```

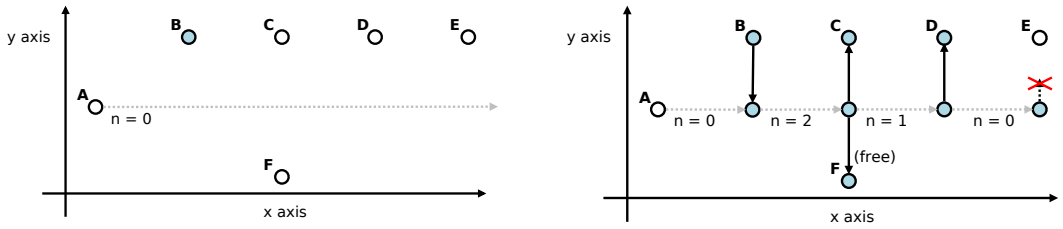
```
29              set startTime [ expr $last_value + 0.1]
                $ns_ at $startTime "$p($ii) broadcast-beacon"
                set last_value $startTime
              }
33          }
        }
    }
```

Listing 5.3: Code example of the periodic transmission of beacons.

5.3 Basic GCP tests

To test that the GCP implementation works as expected, two simple test scenarios has been devised.

The first test will demonstrate that the GCP implementation follows the procedure rules as described in Section 3.1. To avoid unpredicted behaviour caused by the non-uniform transmission probability described in 3.1, we set the transmission probability $P = 1$ for the purpose of this demonstration. In the demonstration illustrated in Figure 5.6, six nodes are used A,B,C,D,E and F. Five nodes B,C,D,E and F have a fixed position. Node A will move from left to right along the x-axis, passing nodes B,C,D,E and F. There is no communication between any of the nodes B,C,D,E and F. With one exception, node A will not enter an area simultaneous covered by the transmission range of two or more nodes. The exception is when node A simultaneously passes node C and F. In the initial state node B will have a new version of a code update. Nodes with the new code update are indicated with the blue colour. The maximum number of update tokens is 2. For the experiment we initially set $n = 0$. The initial number of update tokens for node B, is assumed to be greater than zero, $tokens_B > 0$. The initial number of tokens for the nodes C,D,E and F can be arbitrary in the interval $0 \leq tokens_{C,D,E,F} \leq 2$ as it will not influence the experiment, hence omitted in the illustration Figure 5.6.



(a) Test initial state

(b) Expected test behavior

Figure 5.6: GCP implementation test scenario, where node ***B*** contains a newer version.

Node A will move past node B, hereby receiving an update, and setting the number of tokens $n = 2$. Node A moves past node C and F, updating both these nodes, spending only one token $n = 1$, this shows that one of the nodes C or F is receiving a "free" update. As node A moves further right node D is updated, again using one token, $n = 0$. As node A reaches node E, node A is unable to perform further updates as $n = 0$, and node E will not be updated.

The second test will show that the transmission probability described in Section 3.1 is reflected in the simulation. Five nodes B,C,D,E and F are placed with increasing distance to node A, illustrated in Figure 5.7. Node A will transmit 10000 packets.

We will count the number of packets received at node B,C,D,E and F respectively and examine how the number of received packets corresponds to the transmission probability described in Section 3.1, test results are shown below in Table 5.1.

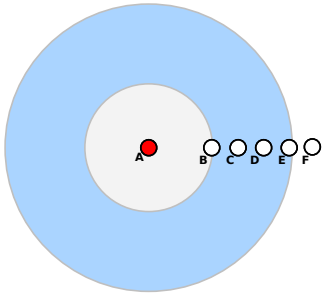


Figure 5.7: GCP range test setup

Node	Distance d	Received packets	Sim. $P(d)$	Calc. $P(d)$	Deviation
B	3.0m	100000	1	1	0
C	4.0m	85836	0.85836	0.856847	0.0015134
D	4.5m	71581	0.71581	0.715625	0.0001850
E	5.0m	30151	0.30151	0.3	0.0015100
F	5.5m	0	0	0	0

Table 5.1: GCP range test result

Test results show that the simulated transmission probability $Sim.P(d)$ and the calculated transmission probability $Calc.P(d)$ are closely matched, and $P(d)$ only differs with a maximum of 0.001534 with 10000 packets send.

With the implementation behaving as expected we continue with a description of the different test scenarios.

5.4 Simulation scenarios

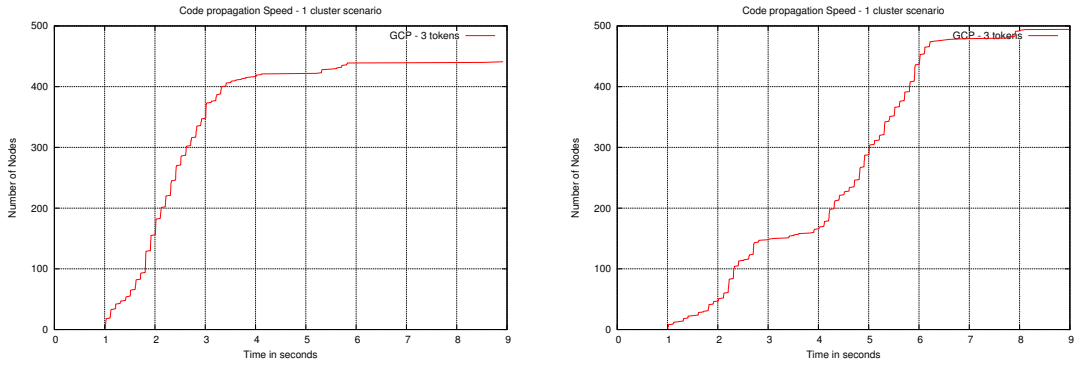
In [9] the test scenarios are divided into two groups, synthetic and realistic. The sections describing the movement of synthetic- and realistic nodes in gives rise to confusion. Parameters for random movement is given in the realistic simulation section along with a dataset of realistic movement of 100 individuals, captured at MIT campus in the academic year 2004-2005. This makes no sense because the movement of the people involved with the realistic test is recorded in the dataset. Also, no explicit assumptions regarding movement is given in the synthetic simulation. Therefore we assume, that the random movement parameters is misplaced within [9] as the realistic simulation scenario draws upon the movement patterns captured in the fore mentioned dataset. This means that we use the parameters for random movement given in the realistic section in [9] to define the movement of nodes for the synthetic simulation scenarios.

The movement parameters of the nodes given in [9] are somewhat ambiguous. We present both our interpretations and compare the result with the results in [9] to select the most fitting.

One interpretation is, that every duration d, where $100ms < d < 500ms$ the node chooses a random way point and moves towards it until a pause p comes and a new duration is commenced. The pause p is between $0ms < p < 100ms$. If a new way

random way point is selected within each of these new durations, the overall result is a node moving around in a relatively small confined area. This setup will often result in a network that will remain fragmented throughout the entire simulation, as the movement of the nodes are insufficient to successfully spread the update. A fast analysis supports the simulation, because during 50 seconds of simulation and a maximum movement speed of $2.0m/sec$ the maximum travel distance of a node is $2m/sec \times 50sec = 100m$ assumed the node always move in a straight line.

Having test scenarios in [9] that include deployment areas of sizes up to $2000m \times 2000m$ the "wandering" distance of the node is only a small fraction of this area, and with frequent $(100ms - 500ms)$ change of direction chosen by random, this movement approach seams inadequate to distribute code updates to the sensor network in its entirety. An example of an early test simulation of 500 nodes in a $250m \times 250m$ area illustrating this problem (shown in Figure 5.8a). The graph shows that after 6 seconds of simulation approximately 440 nodes have received the update and the number does not grow in the following seconds. It should be noted that the updating process starts 1 second after the initialisation, hence the number of nodes will first start to receive updates from the first second of simulation.



(a) First interpretation.

(b) Second interpretation.

Figure 5.8: Test results of two different interpretations of the random movement. The x-axis shows the seconds of simulation and the y-axis shows the amount of nodes, which have received the update.

Another interpretation is to assign a new random way point only when the node arrives at its current target way point. The duration is the same as described above, but instead of having a movement pattern as a "cloud" around the initial start point, the movement pattern changes to straight lines allowing the nodes to reach further around. A test simulation of 500 nodes in a $250m \times 250m$ using this movement model is shown in Figure 5.8b. The graph shows a continual growth in updated nodes and after 9 seconds of simulation, approximately 495 nodes have received the new update. It should be noted, that the same topology is used in both tests. If we solely look at the time frame the second interpretation is best since the overall number of updated nodes is higher, but also the characteristics of the graph is better. Comparing the two graphs, the first (a) shows an immediately updating process, but after a few seconds stops because the rest of the nodes are too far away from the "cloud" to receive an update. The other graph (b) in Figure 5.8 shows a better curve. Within the first 3 seconds every node close

to the source node gets updated. From there, the nodes move around and suddenly after the 4th second, one node reaches a new cluster thus the steep curve. From the 6th second, every node except nodes close to the borders has received the update. Finally, the progress slows down because the remaining nodes are scattered around the edge of the deployment area.

To conclude we choose the second interpretation because of the overall better achievements. Some of the additional information given regarding the different tests are contrary if not ambiguous. In some of the cases the cluster sizes are not fully listed. In others, the amount of clusters and nodes are not specified. In order to clarify our interpretation of the different scenarios, we list them in the following two subsections.

Random waypoint movement

Each node is initially placed at a random location within the deployment area. At the start of a simulation a node picks a random destination within the deployment area and moves towards this destination with a speed randomly chosen between a predefined upper and lower bound, *minSpeed* , and *maxSpeed*. When a node reaches its destination it pauses for a period of time between a predefined upper and lower bound *minPause*, and *maxPause*. The node alternately pause and then move to a new destination throughout the the entire simulation [13].

In the initial generation of movement patterns for the nodes we used the 'setdest' tool. Setdest is an independent tool for random movement generation bundled with the NS2 distribution. The time to create random movement with 'setdest' increased exponentially with the number of nodes, and became very time consuming for clusters with 2000 nodes or more. We found 'setdest' to be slow, but also impractical to our purpose as the output from 'setdest' required additional post-processing to reflect the cluster configuration of the individual simulation scenarios. We therefore wrote a small python application *PyMove*, that will generate random movement much faster(seconds compared to hours) while also reflecting the cluster configuration of a simulation scenario. PyMove generates a movement file that can be imported directly in NS2. An example of defining simulation scenario 3 through PyMove is listed in Listing 5.4.

```
1 from move import *

  scene3 = Scene(50) //  New scenery with 50 sec. simulation time.
  scene3.setSpeed(1.2 , 8.0) //  min and max speed of the nodes (meter/s)
5 scene3.setPause(0, 100) // min and max pause of the nodes (ms)
  // new Cluster x, y, width, height, numberOfNodes
  scene3.addCluster( Cluster( 0, 0, 800, 800,  1000) )
  scene3.addCluster( Cluster( 700, 700, 800, 800,  1000) )
9 scene3.generateMovement()
```

Listing 5.4: Defining simulation scene 3 through PyMove.

The cluster configuration generated by PyMove is visualized in Figure 5.9

It should be noted that the authors of [9] describe quote: "Every movement is bounded by the defined area. The border rules are defined as each node bounce back according to the bisector of incidence angle"[2]. But there will never be any kind of bouncing, because the random way point model will always chose a coordinate within the deployment area[3]. This means that a node can move to the border, but once it arrives it will choose

[2] [9] page 14 - middle of the page.
[3] [13] page 11 - middle of the page.

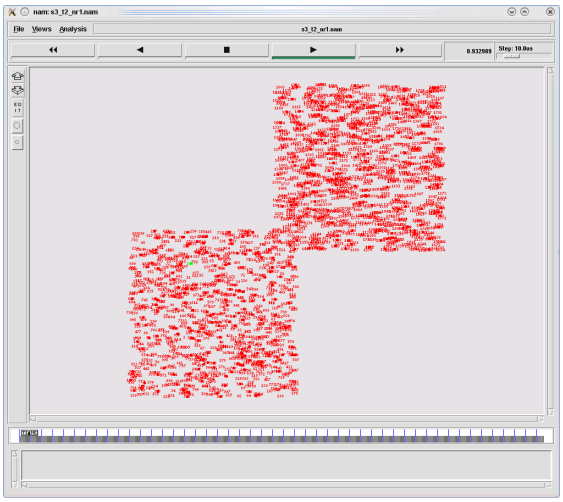
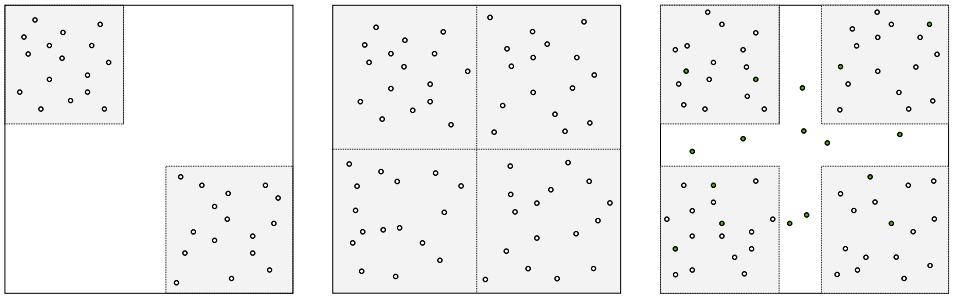


Figure 5.9: Scenerio 3 visualized trough NAM.

a new random coordinate within the deployment area instead of bouncing as described in [9].

Synthetic simulation scenarios

Each of the different synthetic scenarios will be run with the following random movement settings:



(a) Scenario 4, two clusters (b) Scenario 5, four clusters (c) Scenario 6, four socializing clusters

Figure 5.10: Examples of different cluster layouts.

- Nodes move according to a Random Way Point strategy for synthetic workload.
- Pause time: 0 - 100ms.
- Movement speed: 0.8 m/sec - 2 m/sec.
- Time is discretised by milliseconds.
- Beacon interval: 100ms
- Transmission range: r = 3 and R = 5, with a minimum transmission probability $P_{min} = 0.3$ inside range R.
- Simulation time: approx 50,000 ms.

Synthetic scenarios:

- *1 cluster scenario.* One large cluster composed of 2000 sensor nodes, in a 250m × 250m area.

- *1 sparse cluster scenario.* One large cluster composed of 2000 sensor nodes, in a 1100m × 1100m area.

- *2 cluster scenario.* Two clusters of 1000 nodes, in two 800m × 800m areas, with a single 100m × 100m intersection.

- *2 socializing cluster scenario.* Two clusters of 950 nodes, in two 800m × 800m areas. The two clusters are put in a 2000m × 2000m area. The two clusters are not within communication range. 100 additional nodes are inserted, and can move in the whole 2000m × 2000m area to ensure connectivity.

- *4 cluster scenario.* Four separate 500 node clusters, in four 550m × 550m areas. Connectivity is ensured through a shared borders.

- *4 socializing cluster scenario.* Four separated 475 node clusters in four 550m × 550m areas. The four clusters cannot communicate with each other. 100 additional nodes are inserted, these nodes can move in the whole 1300m × 1300m area to ensure connectivity.

- *9 cluster scenario.* Nine separated 250 node clusters, in nine 400m × 400m areas. Connectivity ensured by a common borders.

- *9 socializing cluster scenario.* Nine separated 240 node clusters, in nine 400m × 400m areas. The nine clusters are not within communication range of each other. 90 additional nodes are inserted which move in the whole 1500m × 1500 area.

Realistic simulation scenario

For the purpose of testing GCP in a scenario with realistic movement behaviour the authors of [9] use data sets that captures the location, communication, proximity and activity information of 100 subjects at the Massachusetts Institute of Technology (MIT) over the course of the 2004-2005 academic year [9].

The simulation parameters presented below is identical to the parameters presented in [9] with the except that we have moved information regarding the random movement to the synthetic test in Subsection 5.4.

- Time is discretised by milliseconds.

- Beacon interval: 100ms

- Transmission range: r = 3 and R = 5, with a minimum transmission probability $P_{min} = 0.3$ inside range R.

- Simulation time: approx 50,000 ms.

5.5 GCP test results

In this section we present the results of the GCP simulation. In each of the eight synthetic test scenarios (described in Chapter 5.4) ten random topologies have been generated according to the parameters for the given test case. It should be noted that in each simulation the node containing the update was randomly deployed.

Each of the ten generated topologies are used to run a test with 2,3 and 5 tokens respectively. For all eight scenarios this gives the total amount of 240 data sets which have been used to generate two types of graphs (code propagation speed and load balancing). All graphs are included on the appertaining CD. We emphasize the source node is deployed randomly. This mean that even though the same topology is used to conduct a test with 2,3 and 5 tokens, the source node differs each time. Because of this, the code propagation speed between the different runs within the same topology cannot be compared.

Scenario 1

In this scenario the size of the deployment area was 250 $\times$ 250 meters with 2000 nodes (dense). Looking at the graph for this test case with 2,3 and 5 tokens (displayed in Figure 5.11 and 5.12 (a) - for bigger graphs, see Appendix C) we see two general features:

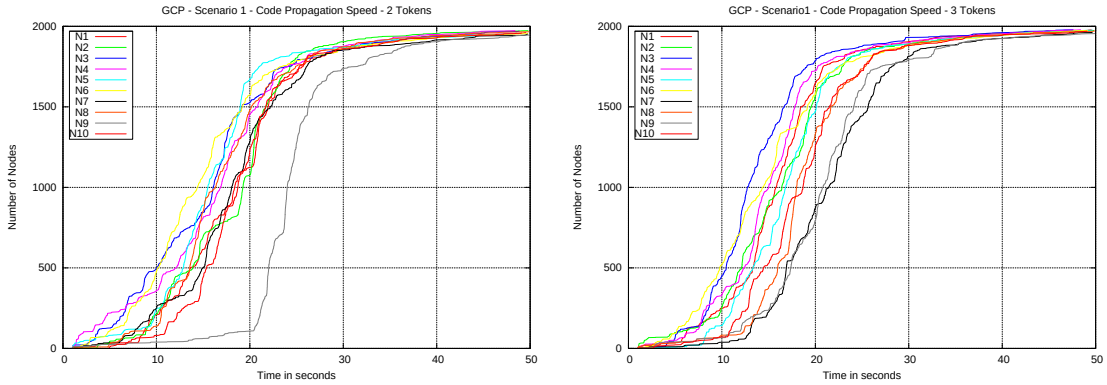
1. After the 50 seconds of simulation neither of the runs, disregarding initial token count, succeeds in updating all 2000 nodes.

2. After approximately 30 seconds of simulation the difference in code propagation among the tests seems to be diminishing.

In the ten simulations with 2 tokens we managed to update between 1946 and 1974 nodes which gives an average of 1961,5. With 3 tokens we managed to update between 1955 and 1979 nodes with an average of 1972,1. In the simulation with 5 tokens we updated between 1974 and 1987 nodes updated with an average of 1980,7. An illustration of the averages can be seen in Figure 5.12b. Comparing the average amount of total updates in Figure 5.12 (b) we see the difference of the initial token count. In the beginning of the simulation all nodes act alike since they have tokens left. The difference starts to become more clear after 20 seconds of simulation. From this point the difference between having 2, 3 and 5 tokens is reflected in the update speed. Not surprisingly, the scenario with 5 tokens surpass the one with 3, which in turn surpass the one with 2. All graphs converge at the point where the problem changes from having tokens left to reaching the last nodes around the edges and corners of the deployment area. In our case the remaining nodes were never within range to receive the update.

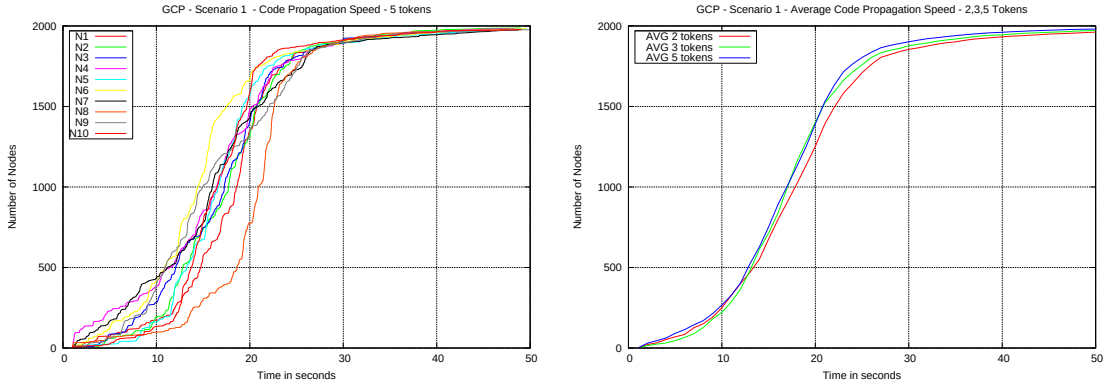
For the same scenario the authors of [9] claim to update all 2000 nodes in 1,7 seconds[4]. Our results do not support this claim. Within 2 seconds we achieve to update less than 100 nodes on average. It should be noted that the shape of our graph resembles that of [9], so a possible error could be a wrong display of time in their figure. We also note, that some of their graphs looks strange. An example is the graph from scenario 6, where the network consist of 4 separated clusters. In this scenario transmitters must travel 200 meters to relay an update from one cluster to another. This should result in a staircase shaped graph, but their graph is smooth. Because the source node was deployed randomly, the possibility arises where the source node was placed in a corner

[4]Code Propagation Speed - 1 dense cluster scenario. 1 Figure in the appendix of [9]



(a) 2 tokens. (b) 3 tokens.

Figure 5.11: The x-axis displays the time in seconds, and on the y-axis display the amounts of updated nodes. (a) shows the result of ten runs of test scenario 1 with 2 tokens. (b) shows the result of ten runs of the same scenario but with 3 tokens instead.



(a) 5 tokens. (b) Average.

Figure 5.12: The x-axis displays the time in seconds, and the y-axis display the amounts of updated nodes. (a) shows the result of ten test scenario 1 with 5 tokens. (b) shows the average graph of 2,3 and 5 token.

of the deployment area. This could influence our results, but after an inspection of the visualisation of each test we have confirmed that no such placement has happened.

Load balancing scenario 1 In order to estimate the load balancing, the number of sent messages have been collected from each simulation. The results of scenario 1 are illustrated in Figure 5.13. The graph shows the average token use throughout the simulation. It should be noticed that our results resembles those from [9]. As expected the majority of the nodes used either 0 or 1 token. This is explained by the fact that a lot of the nodes reside close to the edge of the deployment area. When the update reaches those nodes the rest of the deployment area has already been updated, thus no tokens will be used.

It should be noted that the first updated nodes have a greater chance of using the most tokens, since every node they get into range of will need the update. This becomes more clear when the initial token count is increased. Another important factor is the

random movement speed of the individual nodes. Among the firstly update nodes, some of them will choose a fast movement speed and a way point close to the edge of the area. Because of this, they have a chance to move faster than the spreading cloud of updating nodes and thus encounter nodes who will request the update. If such an event happens the node will possibly use all available tokens. As the graph shows though, this is rarely the case. For instance, the number of nodes using 5 tokens is only 11,4 in average.

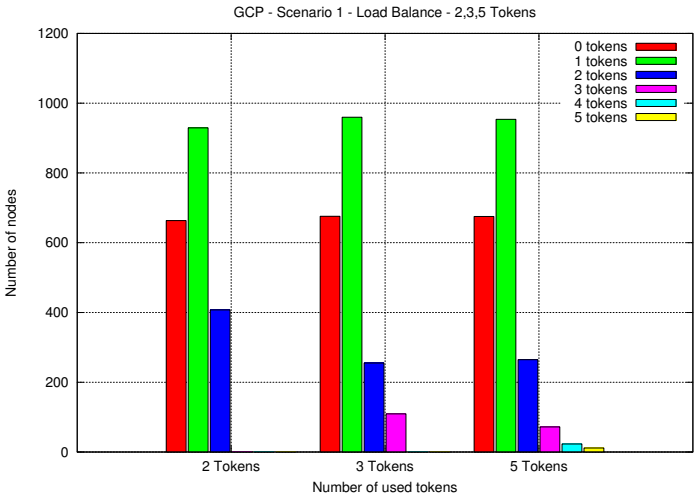


Figure 5.13: On the x-axis the number of tokens is displayed and on the y-axis the number of nodes are displayed. The graph shows the average amounts of used tokens in scenario 1. It should be noted that nodes which did not receive any updates would not have anything to transmit and thus would not use tokens. Therefore, the first pillar with 0 tokens contains the combined amount of nodes which did not transmit any update or did not receive any.

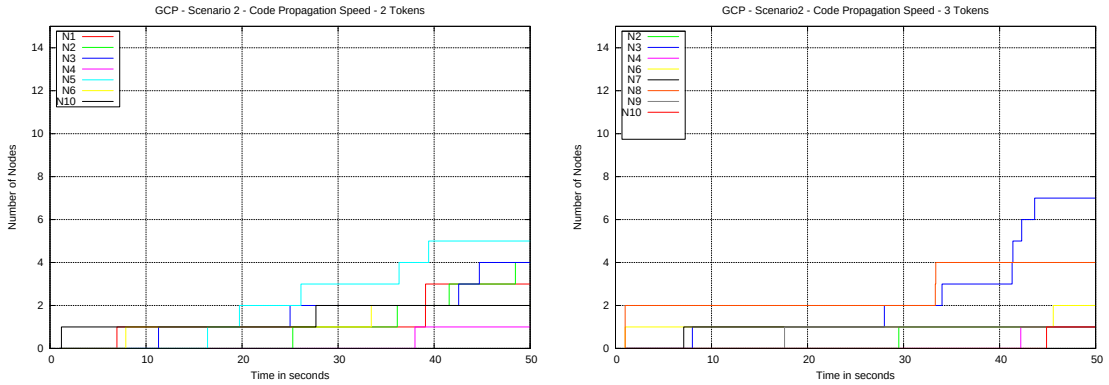
Scenario 2 - 8

The reflection upon the simulation results from scenario 2 to 8 has been gathered into the same subsection because the results are similar. Common for each of these scenarios are the size of the deployment area is huge compared with scenario 1, but the number of nodes stay the same. This gives a lower average density and thus a problem arises because the distance between nodes become too large. This means that the source node will have to spend a lot of simulation time moving into range on another node, thus only a few notes will get updated.

We choose to show the results for scenario 2 because it has the lowest average distance of the seven scenarios given uniform distribution and thus should provide the best results (in theory, though in practice they are almost identical).

The results from scenario 2 is depicted in Figure 5.14 and 5.15(large pictures can be found in Appendix C). Every run shows the same low performance and the characteristic is apparent in the graphs for the rest of the scenarios. In 50 seconds of simulation the number of nodes which get updated are between 0 and 11. It should be noted that some of the runs did not succeed in updating any nodes and thus has been omitted from the graph. This is extremely low, especially compared with the result of [9] which update all 2000 nodes in less than 20 seconds of simulation time [5].

[5]Code Propagation Speed - 1 sparse cluster scenario. 2 Figure in the appendix of [9]



(a) 2 tokens. (b) 3 tokens.

Figure 5.14: The x-axis displays the time in seconds, and on the y-axis display the amounts of updated nodes. (a) shows the amount of nodes in scenario 2 updated with an initial token count of 2. It should be noted that the figure does not contain ten graphs. This is because run N7,N8 and N9 did not succeeded in updating any nodes and thus was omitted. (b) shows the same scenario but with a token count of 3. Again 2 runs - N1 and N5 - did not succeed in updating any nodes and was omitted. Common for both Figures are their low performance.

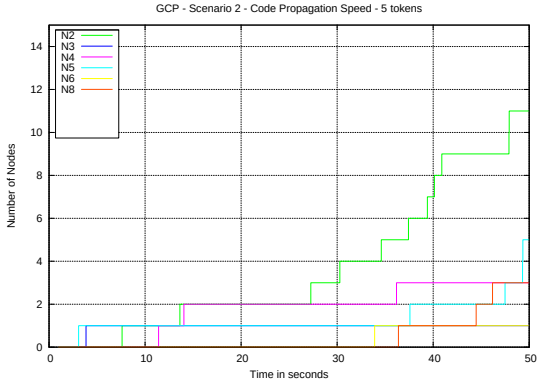


Figure 5.15: The x-axis displays the time in seconds, and on the y-axis display the amounts of updated nodes. This figure shows the number of nodes updated with a token count of 5. Like in the test with 2 and 5 tokens some of the ten runs did not succeed in updating any nodes. Therefore, N1, N7, N9 and N10 has not been shown in the graph.

Again, we find the result in [9] very doubtful. To update every node within the 50 seconds simulation time is impressive, but with these distances between each node and the short radio range and the slow movement it becomes unlikely in our opinion.

Load balancing scenario 2 Because of the low performance in the scenario 2 graphs for load balancing has not been made. The same applies for scenario 3 to 8. They all look the same. None of the 2000 nodes have received the update and thus used zero tokens except a few nodes who was very close to source node.

Realistic case with movement

As we have seen with the synthetic test regarding code propagation, our results does not match those from [9]. In the SQL data sheet for the realistic case we see that though the transmission range is longer, we only have 100 people (nodes), thus the average density will still be small as in scenario 2 - 8. Because of this, we choose to not use time on extracting the information of the realistic case from the data sheet, but instead focus on the reason for the our big deviation.

5.6 Explanation for deviation

From a mathematical point of view the results from [9] does not match up. With a perfect uniform distribution a grid of $\sqrt{2000} \approx 45 \times 45$ nodes will cover the 250×250 meters area. This means that the average distance between two nodes is $250 meters / 45 nodes \approx 5,7 meter$. An illustration of the uniform distributed deployment area can been seen in Figure 5.16.

The network is dynamic so the distance between two nodes $\boldsymbol{A}$ and $\boldsymbol{B}$ changes from the second the simulation starts. The distance between those two nodes can either grow or shrink depending on the direction of their movement. On average this can be seen as $\boldsymbol{A}$ moving towards $\boldsymbol{B}$, where $\boldsymbol{B}$ is static thus does not move.

With this assumption and the information regarding the radio range and movement speed of the nodes (from Section 3.1 and 5.4) we know:

- nodes move between 0,8 and 2,0 meters - on average 1,4 meters per second.
- updates are guaranteed on ranges less than 3 meters, from 3 - 5 meter the chance for reception is decreasing. Above 5 meters the chance of reception is 0.
- updates are sent every 0,1 second and the nodes initialises with a small random delay. This means they do not transmit their updates at the exact same time. Additionally, the update is instant thus can be relayed in the first scheduled update message following the reception.

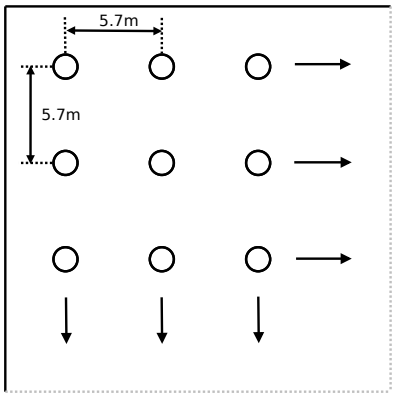


Figure 5.16: A figure showing the a uniform distribution of nodes in a deployment area with the same size of scenario 1.

Best case scenario 1

We now try to estimate the best case scenario for the results presented in [9]. In 1,7 seconds the nodes can move 2 meters at best since movement would take 1 second. This means that every node has to be in range of at least another node and the entire network is somehow connected after the first second when the update process starts. In the best case every radio transmission will be received up to 5 meters away. Assuming that the source node is dropped directly in the middle of the deployment area and instantly updates as soon as possible, nodes within 5 meters will receive the update. Now, after the first 1,0 second, initialisation has been made, the update get transmitted and the first wave of nodes receive it. They all have their first update message scheduled for 1,01 and thus transmit the update. Now the next wave receives the update. Again - the nodes in the second wave have scheduled their update for 1,02 and thus transmit it. With an average count of 45 nodes per line in the node grid and the source node placed in the middle, a total of 22 nodes has to be updated, thus 0,22 second is needed to update the nodes closest to the edges. From there, additional 0,22 is needed to reach the nodes in the corners, which in total gives an update time of 1,44 seconds. So if the nodes are distributed so the range between is maximum 5 meters, the source node is directly in the middle and the scheduling of the individual nodes are perfectly aligned all 2000 nodes can receive the update in only 1,44 seconds.

A contradiction occurs since the gap between the nodes has to be 5,7 meter for a uniform distribution. This could indicate that the authors of [9] either did not have a uniform distributed topology or their radio had a longer range than described in their work.

Best case scenario 2

In scenario 2, the deployment area is 1100 $\times$ 1100 meters with 2000 nodes. If they are evenly distributed in that area the average range between two nodes will be:

$$\frac{1100}{\sqrt{2000}} \approx 24,5 \frac{meters}{node}$$

.

The distance between two nodes is 24,5 meters given an evenly distributed network. Calculating the average distances for scenarios 3 to 8, we even get somewhat larger distances ranging from 24,6 to 25,3 meters. As said in the beginning of this subsection the results from scenario 2 to 8 are identical with small deviations.

To estimate the best case scenario, the assumptions about movement and radio range mentioned above is used. This means that every message send can be received at a distance of 5 meters. The source node is dropped perfectly in the middle of the deployment area so there is 550 meters to the edges of the area and 777,8 meters to the corners. After 1 second of initialisation, the source node needs to travel 19,5 meters to a neighbouring node to get within radio distance. With 2 meters per second the source node needs 10 seconds of simulation time to reach one neighbour and update it. From here, the two source nodes can now within another 10 seconds update one other node each, which gives a total of 3 updates in 20 seconds. This does not match the results shown in [9]. Again we must conclude that either some explicit assumptions has been left out, their time scale on their graphs are wrong, the topology is not uniform or their radio settings does not match the assumptions given in their work.

5.7 Replicate results of GCP

In this section we change the settings of our movement and radio range in order to see what it takes to replicate some of the results in [9]. We wish to replicate is the code propagation speed of scenario 1, scenario 2, scenario 5 and scenario 6. These scenarios has been picked out because:

- scenario 1 and 2 are described in detail in the previous section
- scenario 5 and 6 are more complex scenarios than the first pair but easier than scenario 7 and 8 (which requires extreme settings to match the original results).

To replicate the results we adjust the settings regarding radio range and movement speed. It should be noted that they somewhat compliment each other in the way, that increasing the radio range could be seen as improving the movement speed as more node will get visited and vice versa. It should also be noted that we only focus on code propagation speed in the replication of the result of a given scenario.

Scenario 1

In less than two seconds the authors of [9] claim to update all 2000 nodes within a 250 $\times$ 250 meter area (dense cluster). With the original settings of movement speed and radio range we managed to update in average 1961,5, 1971,2 and 1980,7 nodes(2,3 and 5 tokens respectively) in 50 seconds. This means that the goal is twofold:

- lower the update time to a maximum of 2 seconds.
- update close to 2000 nodes.

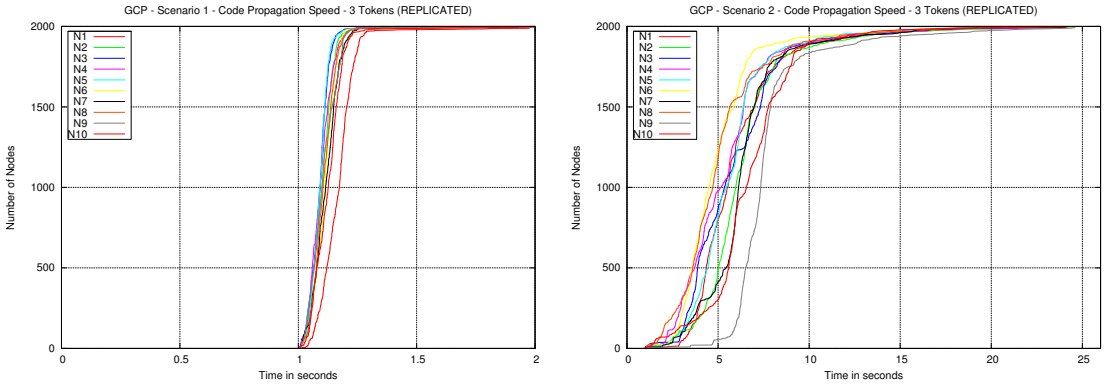
We incrementally increased the movement speed and radio radio range while comparing the new result with those in [9]. With a movement speed of 15 to 20 meters per second and a radio range of 6 - 10 meters we managed to update 1999 nodes in 1,38 seconds. These settings are far higher than those given in the GCP report. To replicate the results from scenario 1 we had to increase the movement speed with a factor ten, and double the radio range. A picture of the graphs for the 10 runs with 3 tokens can be seen in Figure 5.17a. A larger graph can be seen in Appendix D.

Scenario 2

Next we replicate the original results of scenario 2. This scenario is the sparse cluster where the authors of [9] claims to update all 2000 nodes in a maximum of 20 second. With the original settings we manage to update between 0 and 11 nodes in 50 seconds of simulation.

A possibility was the author somehow used the settings found in Subsection 5.7 without knowing it. Therefore, we start by testing those settings in the replication of scenario 2. These settings does not provide a result close to what the authors claim, so an increase of the settings are needed again. With a radio range of 17 - 23 meters per second and a movement speed of 15 - 20 meters we managed to update all 2000 nodes within 20 seconds, which matches the results shown in [9]. Our results with 3 tokens can be seen in Figure 5.17b. A larger graph can be seen in Appendix D.

As in the replication of scenario 1, the needed radio range and movement speed to replicate the results provided in [9] is far higher than stated. In this scenario the movement speed was a factor 10 higher and the radio was four and a half time longer.



(a) Scenario 1: Ten runs with 3 tokens.

(b) Scenario 2: Ten runs with 3 tokens.

Figure 5.17: The x-axis displays the time in seconds, and on the y-axis displays the amounts of updated nodes. (a) shows the amount of nodes in scenario 1 updated with a token count of 3, while (b) shows scenario 2. To create this replicated results the movement speed has been increased multiple times and so has the radio range, compared to the original assumptions given in [9].

Scenario 5

This scenario contains four clusters aligned in a square. In this scenario the original results completes the update in less than 30 seconds. As before, we started with the previous found settings but again had to increase the movement speed to increase the likelihood that two nodes would go get close to a shared border, and the update could be transmitted from one cluster to the next. With a movement speed of 20 - 25 meters per second and a radio range of 17 - 23 meter we manage to update all four clusters in 2 of the ten runs. This can be seen in Figure 5.18a - for larger figures see Appendix D. Each step on the graph is a new cluster which gets update. This means, that with these settings and 10 simulations we managed to update 4 clusters 2 times, 3 clusters 7 times and 2 clusters 1 time.

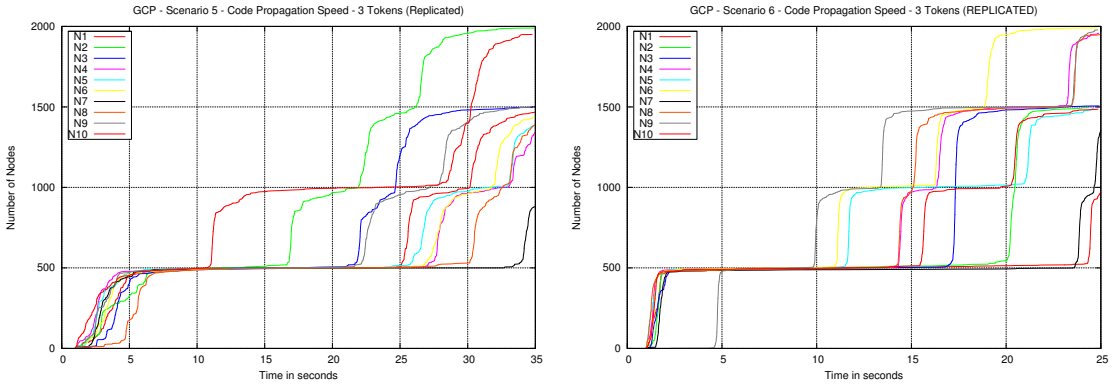
We also discovered that increasing the radio range yields a better result when an update shall transmitted across a common border compared to increasing the movement speed.

Scenario 6

The last scenario we are replicating contains four socializing clusters. This means that transmitters will have to carry the update around and eventually enter the clusters which will lead to an update of it. The authors finish this update in less than 25 seconds. With the settings found in replication of scenario 5 we are not able to update more than 2 clusters in 25 seconds of simulation, thus we increase the settings again. Our result can be seen in Figure 5.18b. A larger graph can be seen in Appendix D. With a movement speed of 25 - 30 meters per second and a radio range of 25 - 30 meters we managed to update all 4 clusters in four of the ten runs.

This indicates that with those settings an update of all 4 clusters is possible but still we need to further increase it if we want to update all clusters in each simulation.

Summarising the replication of the 4 different scenarios we see that the assumptions about the different settings provided in [9] does not produce the results presented in the



(a) Scenario 5: 10 runs with 3 tokens.

(b) Sceranio 6: 10 runs with 3 tokens.

Figure 5.18: The x-axis displayed the time in seconds, and on the y-axis display the amounts of updated nodes. (a) shows the amount of nodes in scenario 5 updated with an token count of 3, while (b) shows scenario 6, also 3 tokens. To create this replicated results the movement speed has been increased by multiple times together with the radio range, compared to the original assumptions given in [9].

same article. Different events could be the reason of this. Firstly, the description of the scenarios were insufficient. Especially a change of topology or size of the deployment area would make a huge different. Secondly, they could have made a mistake when defining the different settings of the movement speed and radio range. This is doubtful though, since no direct connection could be found in the increase of settings compared to the scenarios.

5.8 Summary

In this chapter we have described the NS 2 tool in which we run our simulations. We have then described our implementation of the GCP protocol and afterwards verify that it behaviour as we expect. To test this we have devised a few test cases. Then we introduced the synthetic and realistic test cases which, in total, gives 9 different test scenarios. We run the 8 synthetic test cases and discovers the original results does not match ours. Our results for scenario 2 to 8 resembles each other. In 50 seconds of simulation we only manage to update between 0 and 11 nodes. To investigate this we created best case scenarios and came to the conclusion, that the original results presented in [9] is very optimistic. Finally, we changed the original settings of the radio and movement speed to replicate the original results. We found out that the settings should be increased with multiple factors in order to produce a result that was somewhat similar with those presented in [9].

Extension of the GCP protocol

This chapter describes our extension of the GCP protocol. Firstly, we elaborate on the main problems with the original GCP protocol with focus on the lack of recovery management. Hereafter, ideas to remove some of the problems within the original protocol will be introduced and used to create the specification for our extension of the protocol. We call it eGCP. With these specifications we create an implementation and verified that it behaved as expected. Finally, we compare the our extension with the GCP protocol.

6.1 Problems with the GCP protocol

In Chapter 3 we described the GCP protocol and noticed several problems:

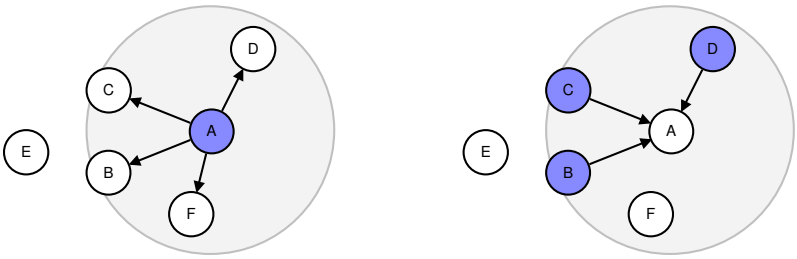
- Transmitting beacons 10 times per second is expensive energy wise. Especially with a movement speed of 0,8 to 2 meters per second.
- One of the assumption mentioned in Section 3.1 was that collision and packet losses was assumed not to occur.
- Another problem was the broadcast approach the GCP protocol used. Without the assumption of nodes not crashing, broadcasting a beacon or update message will give rise to possible implosion. However, it is a rare event.
- Finally, some scenarios trigger multiple use of tokens instead of the intended one. If two sources are available to a node both will transmit their update and use a token. This problem contradicts the authors idea with evenly balanced code update.

The GCP protocol described in [9] will give rise to implosion under certain conditions. An example could be a node enters the vicinity of a multiple number of nodes and broadcasts a beacon with an outdated software version. Now, the entire neighbourhood of that node will respond with update messages. The problem is illustrated in Figure 6.1a and 6.1b, while the configuration of the WSN is listed in Table 6.1

Node	Version	Tokens
$\boldsymbol{A}$	1	3
$\boldsymbol{B}$	2	1
$\boldsymbol{C}$	2	2
$\boldsymbol{D}$	2	3
$\boldsymbol{E}$	1	2
$\boldsymbol{F}$	1	3

Table 6.1: WSN configuration.

Node $\boldsymbol{A}$ starts by broadcasting its beacon containing id and software version to the neighbouring nodes. The nodes $\boldsymbol{B}$,$\boldsymbol{C}$,$\boldsymbol{D}$,$\boldsymbol{F}$ receives the beacon broadcasted by node $\boldsymbol{A}$. Since node $\boldsymbol{F}$ has the same software version as $\boldsymbol{A}$, $version = version\_r$, no action will be taken. On the other hand, nodes $\boldsymbol{B}$,$\boldsymbol{C}$ and $\boldsymbol{D}$ all have a newer software version than node $\boldsymbol{A}$, $version_{\boldsymbol{B},\boldsymbol{C},\boldsymbol{D}} > version\_r$. Given that update tokens are available on node $\boldsymbol{B}$,$\boldsymbol{C}$ and $\boldsymbol{D}$, $tokens_{\boldsymbol{B},\boldsymbol{C},\boldsymbol{D}} > 0$ these nodes will start to transmit their software version to node $\boldsymbol{A}$, hereby causing implosion while wasting tokens.



(a) Node A sends beacon (b) Node B,C,D sends update

Figure 6.1: Implosion problem in GCP

Specifying a recovery mechanism

In the GCP protocol a build-in recovery mechanism exists as long as the update can be contained in a single message. This means that if a node misses an update a new beacon will be transmitted shortly after. Our recovery mechanism supports updates spanning over a multiple number of packets. To specify a recovery mechanism a lot of different decisions have to be made. We start from the top with specifying how a transmission starts and then proceeds to handling the different errors and packet losses.

In our protocol, the only communication that requires a recovery mechanism is our update messages. When an update transmission is started, no handshake is used. The nodes which need the update should be ready to receive it at all times so a handshake should not be necessary. Also, by keeping the amount of protocol messages at a minimum we ensure that battery power is not used unnecessarily.

When the update has begun acknowledgement messages will not be transmitted back upon successful reception. Instead NACK-messages will be used in order to lower the amount of protocol messages needed. Another reason for using NACK-messages is that it minimizes the chance that the source node will implode, compared to ACK-approach if the average number of neighbours is high.

The next part of the recovery/reliability mechanism is to ensure that in case of packet drops or node crashes, the rest of the network can continue. We will start by defining the rules of our recovery/reliability mechanism. It should be noted our recovery mechanism somehow resembles the *Go-Back-N* instance of the Automatic Repeat-reQuest (APQ) protocol [21]. The difference is our implementation uses NACKs instead of ACKs, and thus the time-out responsibility is shifted from the original sender in Go-Back-N to the receiver. Also, it should be noted that we cannot use the "window" since it represents a queue of consecutive sequence un-ACKed packets.

Rules of recovery

- A beacon transmission will trigger a best update candidate election if a message with a higher version has come. number. The election picks the source node with the highest version and token number (will be called the *best candidate* from now). The best candidate transmits its update.
- The candidate will continue to transmit the updates in numbered sequence starting with the first packet continuing until the last packet has been transmitted.
- Whenever the update process starts, the receiving node starts a timer when the first packet is received. Each time the node receives the next packet in line or

one with a lower sequence number, the timer is reset. This will happen when the receiving node looses and transmits a NACK-message. If the timer reach zero a NACK-message will immediately be broadcasted back to the source node. The NACK-message contains the highest-in-sequence packet number received by the receiving node. In order to do so, an node id is required of the source. We therefor add a node id in the packet (detailed in subsection 6.1).

- If the receiving node detects a packet has been dropped, due to a jump in the sequence numbers, it will reply with a NACK-message containing the last packet number it received. Then the timer is restarted on the node who detected the packet loss. If this timer reach zero before the node has received the lost packet the node retransmit the previous NACK-message. This can continue until a threshold of nack counts is reached, where the node will abort the update.

To grasp the rules mentioned above we explain them in a few different scenarios using sequence diagrams.

Explaining our NACK-approach if node ***A*** (candidate node) is transmitting an update to node ***B***, ***A*** will continue to transmit the packets until it either finish or receives a NACK-message on a specific packet from ***B***. When a NACK-message is received the previous packet will be retransmitted and the candidate node continues from that packet number. This procedure can be seen in the sequence diagram in Figure 6.2.

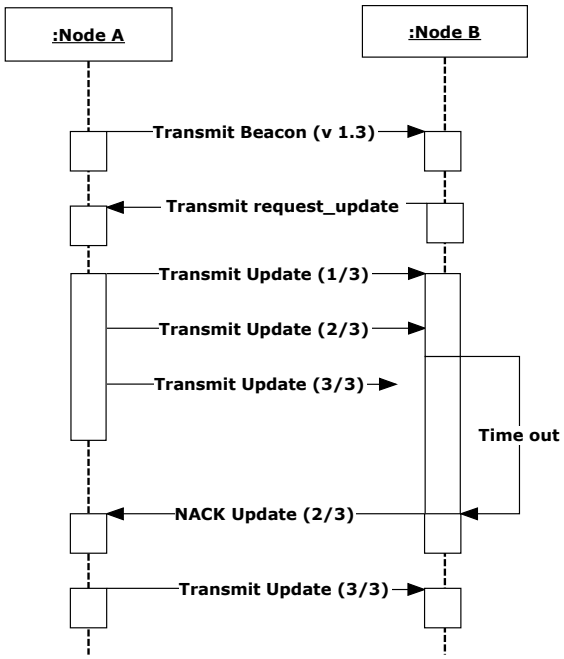


Figure 6.2: A sequence diagram of the NACK-approach. The message containing the 3rd update package is lost. This leads to a time-out on the receiving node which generates a NACK-message containing the last received message which was the part second update package. The source node ***A*** receives the NACK-message, retransmits the next packet indicated by nack and continues until all packages have been sent.

In the next sequence diagram in Figure 6.3 we see a case where an update packet is lost (it should be noted that the beacon transmission has been omitted). Node ***B*** detects the loss, transmit a NACK-message and starts the timer. Before the source node ***A*** receives the NACK-message it has already transmitted the next packet. This means

that ***B*** will receive the next packet which will be dropped thus prevent a restart of the time-out clock. This leads to another time-out on ***B*** which retransmit the previously send NACK-message and reset the timer. Before the third time-out, the second packet is received and the rest of the update goes as planned.

It should be noted the a buffer on the nodes could improve the performance. It is a waste to drop received packets just because they are not in sequence, but adding a buffer would put addition requirements on the target node. Additionally, the protocol would need a control mechanism for the buffer which induce complexity. For future work a possibility could be to implement the "sliding window" hinted in [20]. It should also be noted that our managing of the NACK-messages could be improved. By listening for packets from the source the node might receive a packet second in line and thus transmit a NACK immediate instead of waiting for the time-out. This will provide a higher throughput, since throughput is not (typically) a prioritised resource this change was moved to future work.

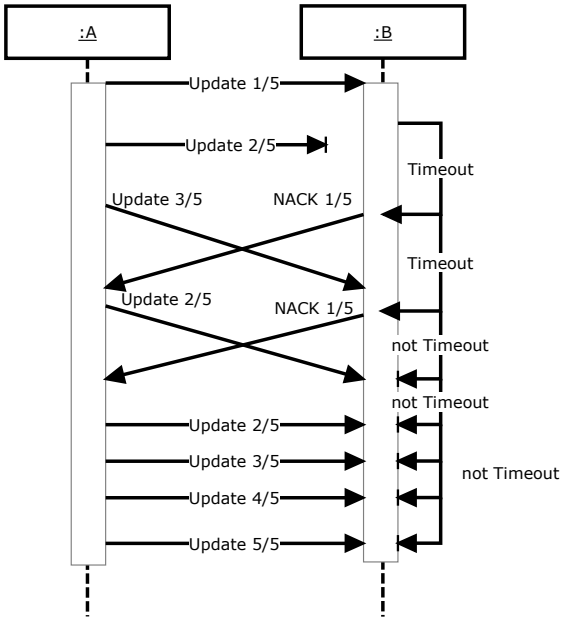


Figure 6.3: A sequence diagram of the NACK-approach where the initial election has been omitted. The message containing the second update part is lost. This leads to a time-out which generates a NACK-message containing the last received message which was the first part of the update message. Before the source node ***A*** receives the NACK-message it transmit the third update message. The receiving node ***B*** now receives the third message from ***A*** indicating that the NACK message was lost. Therefore, the time-out clock is not reset and shortly after, ***B*** retransmit the NACK-message and then receives the second message it was waiting for. The time-out timer is started again but resets when the second update message is received again. The rest of the update continues as normal.

We have identified a problem with our NACK-approach and the dynamic WSN. The mobile nodes will roam and if a node comes into range of a node while broadcasting an update, it will respond with a NACK, saying it missed the first update packages. This will result in a restart of the entire update since the source node will retransmit the first package and continue in sequential order. With a high node density, the probability for such an event is high. Just after deploying a node with a new update. The example is

illustrated in Figure 6.4.

How severe the problem is depends on the average density of the scenario. Tests will have to show if the precedent rules should be changed. A possible way to solve the problem is to introduce some kind of packet number awareness. An example could be if the ratio between the packet number received divided with the total amount of packets exceeds a predefined threshold the node should not transmit a NACK but instead drop the update packets.

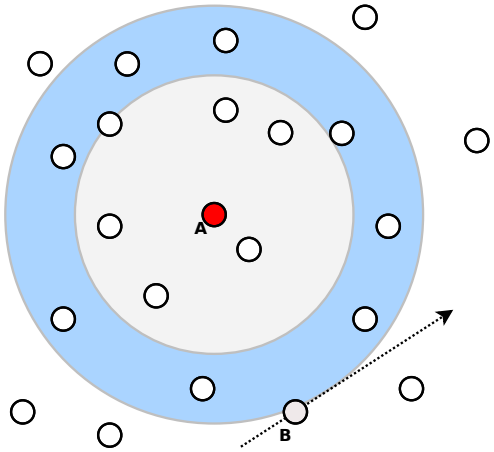


Figure 6.4: The node ***A*** is displayed broadcasting an update message together with its transmission range to it neighbouring nodes. Because of the moving nodes, node ***B*** moves through the area of ***A*** and thus might receive packets in the middle of the update. ***B*** will think the first messages was lost and thus broadcast a NACK-message back to ***A*** which will retransmit the entire update. Unfortunately, before the update is complete, ***B*** moves out of range of ***A*** thus the reset is a waste of time and energy.

Time-out Period

To select the proper value for the time-out variable is difficult. Its important to keep it as low as possible so NACK-messages can be transmitted as fast as possible. This way, the source node will react faster to the packet loss, thus speeds up the overall update. On the other hand, congestion can happen and sending more packets would be a bad reaction compared to slowing down the transmission rate (induce longer time-outs). A possible solution could be adding a randomly small delay to the time-out variable after each usage.

Saving tokens

The original idea using the tokens was, that every time a node transmitted an update, the token count decreases at the given node. This gives a better load balancing. Unfortunately, with the original design, possible implosion could happen under certain circumstances. As mentioned in Section 6.1, if node ***A*** crashes and later wake up, its neighbourhood might have received an update in the meantime. When ***A*** then transmit its beacon, ALL neighbours will start sending an update message back to it. This will result in great collision but more importantly waste $N-1$ tokens, where N is the number of nodes in the neighbourhood of ***A***.

We propose to add additional information in the beacon and create a new message type. A byte is added to the beacon which contain the number of tokens left on the node.

This way, each node has a way to announce through its beacon how many tokens it has left and thus indirect tell, how many times the node has transmitted update messages of this version.

We create a new message type called *request_update*. With this new message type we change the way our nodes ask for updates. Previously requests for an update was initiated by a beacon-handshake. Instead, nodes will now transmit a beacon with its version number, node ID and available token count. When a node ***A*** receives a beacon with a newer version than its own a election period of 100 milliseconds is started. Since the interval of beacons are 100 milliseconds, ***A*** will receive a beacon from each node in its neighbourhood within this election. The highest token count is stored together with its appertaining node ID. With this information, ***A*** will know which node within its neighbourhood that has transmitted the least amount of updates and thus elects that node by sending a *request_update* message (containing the ID of the node with the most tokens left). This action will start the update process. If more nodes are tied with the highest token count, the first one received within the election is chosen.

6.2 Specification of extension

In the previous section we presented some extensions to the GCP protocol to solve some of the issues regarding to the implosion and recovery mechanism. In this section, we review our previous description of the GCP protocol from Chapter 3.1 and add our own extension to it (abbreviated eGCP from now).

To describe our extension, the approach of [12] is used again:

Service

The service remain the same as in the original GCP protocol. Additionally, reliability is insured for larger updates which span over multiple packets.

Environment assumptions

The assumptions are the same as in the original GCP protocol, with exception of collisions and data losses which are are possible now.

Message vocabulary

Two new message types, *request_update* and *NACK* are added to the existing vocabulary of the GCP protocol. We have also changed the layout of the beacon packet shown in Figure 6.5 introducing nodes id and tokens fields. Moreover, we altered the update packet layout by adding extra fields such as nodes ID, packet number, total packet number. This is illustrated in Figure 6.6.

Message encoding

In order to implement our extension to the GCP protocol the original message encodings are changed. Each node has an individual id, which is attached to the beacon message together with its version number and token count. System now is susceptible to failures. The update message is changed, so it now contains the id together with the four values - the version number, the packet number, the total amounts of packets, and the data in the update. The reason for adding the id to the update message is, that roaming

nodes can receive an update in the middle of the process and thus transmit a NACK to the right node. The request_update message contains the id of the node with the most tokens available. Lastly, the message NACK includes the id of the receiver together with the last received packet number.

The information is outlined in the following listing:

```
beacon(ID, version, tokens)
update(ID, version, pk_nr, pk_total, data)
request_update(ID_r)
NACK(ID, last_received)
```

The Figures 6.5, 6.6, 6.7, 6.8 illustrate the packet layout of the eGCP protocol.

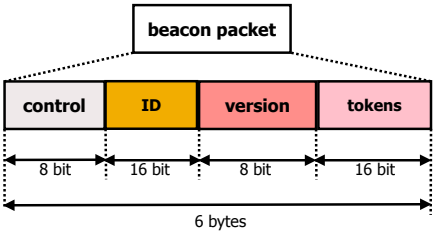


Figure 6.5: eGCP beacon packet

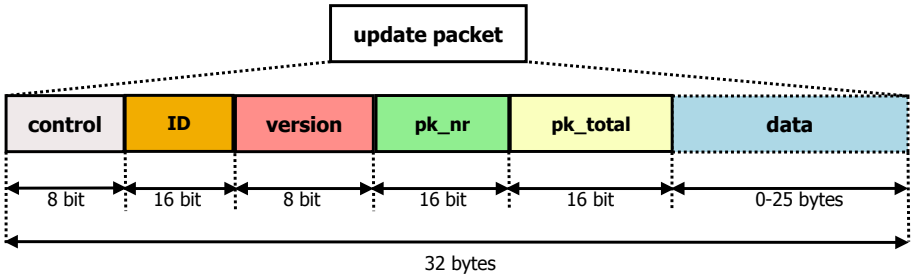


Figure 6.6: eGCP update packet

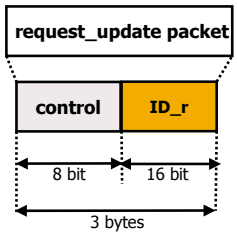


Figure 6.7: eGCP request update packet

Procedure rules

With the eGCP changes formally described the procedure rules can be addressed. As with the GCP protocol we use flow chart diagrams to describe the behaviour of the protocol. First, we describe the different labels used in the flow charts. Thereafter we address some special notation in the charts and finally present the diagrams with a thorough explanation.

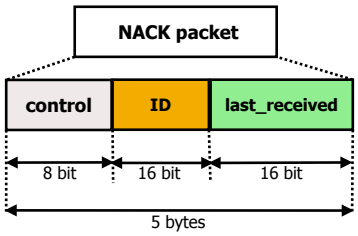


Figure 6.8: eGCP nack packet

It should be noted that the labels in the diagram still have the extension *_r*, which represents a variable from a remote node. Additionally, the labels ID, tokens and version can have the extension *_upd*. These variables are containers for information regarding the node with the newest version and highest amount of tokens - in other words the best candidate for transmitting the update.

- *version* represents the local version number of the software.
- *ID* is the unique identification number of the node.
- *pk_nr* is the current packet number.
- *pk_total* denoted the total amount of packets in a current update.
- *tokens* represents the current amount of tokens available on a sensor node.
- *initNrOfTokens* is the predefined number of tokens that initially are available on a sensor node.
- *timeout* is a function with take a parameter of time before an action is triggered or must be taken.
- *nack_count* is the amount of times a NACK message can be retransmitted.
- *threshold* is the maximum value for nack_count.
- *last_received* denotes the packed number a node last received successfully.
- *software* is the constructed image which is created after every packets in an update has been received.

After introduction of labels we continue with the description of the diagrams. It should be noted as in Section 3.1, states are marked with **bold**, message types in `verbatim` and variables with *italic*. Additionally, to keep the number of transition arrows at a minimum we introduce duplicate states which are marked with a dotted line. For instance - every branch of the protocol returns to the main state **Idle** at some point so to minimise the use of long arrows, the state has been moved closer and shown with a dotted edge. Lastly, in order to create the flow charts as clear as possible we have divided them into separate pictures. To see where the previous chart ended and the new one begins we have used a double border to highlight this.

Starting with Figure 6.9 at the start, an initialization of the internal variables are made and the protocol continues to the main state **Idle**. From here four different events can happen. Either the node can receive a beacon, receive a NACK, receive an update, receive an update_request or transmit a beacon (if the token count is greater than zero). The following paragraphs address them individually in detail.

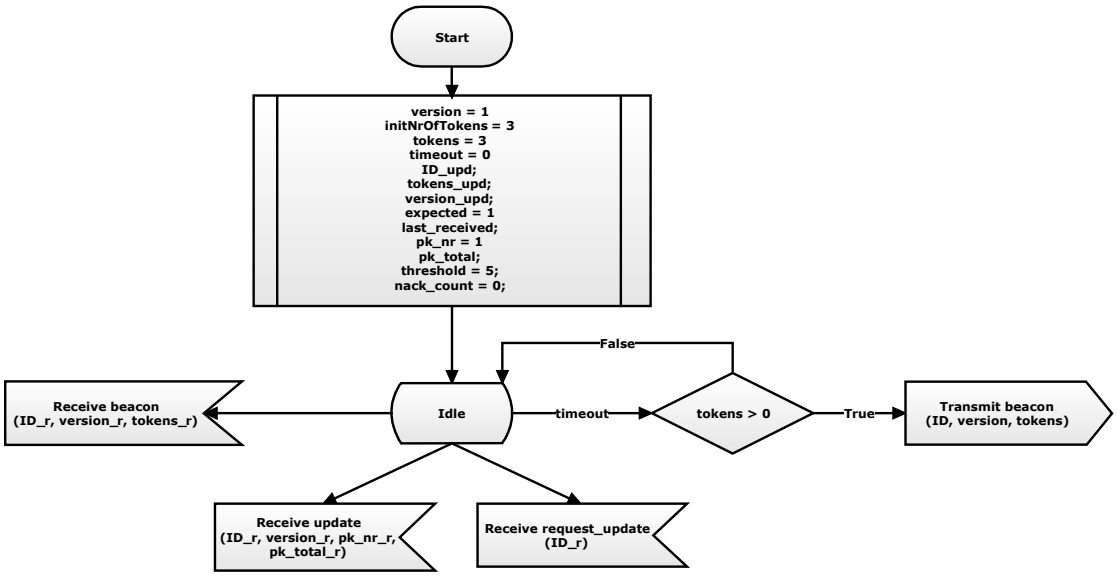


Figure 6.9: The main part of the eGCP protocol.

Transmit beacon From the **Idle** state a predefined time-out occur and the node checks the token count. If the node has at least one token a `beacon` is transmitted and the protocol returns to the **Idle** state.

Transmit update Figure 6.10 shows a detailed illustration of the "Transmit update" process. Initially, the packet number is reset to one. This is done to ensure, that the updates are transmitted from the beginning. Now, the first update is transmitted including the version number, packet number, total amounts of packets, and data. The node waits for a short period for eventual NACK messages. If none was received it compares the packet number with the total amounts of packets in the update. If the packet number is less than the total, the packet number is incremented and the next packet is transmitted. If the packet number is equal with the total amount of packets in the update, the node will conclude the update is finish, decrement its amount of tokens and return to the **Idle** state. If an `NACK` message is received, the node compares the id to ensure that the message was intended for it. If the id matches, the node changes the packet number to the needed packet and continues transmitting updates until it reaches the total amount of packets in the update. If the id did not match the node just continues with transmitting the next update.

Receive beacon In the **Idle** state, the node can receive `beacons`. When a `beacon` is received, the node compares the versions. If the `beacon` contains a newer version, the node will store the information in the local variables *ID_upd*, *tokens_upd* and *version_upd* as the best candidate. Now an election begins where the nodes listen for incoming `beacons`. If it receives any, the node will compare the version number with the number from its best candidate. If they are equal the node will look at their tokens. If the count is higher than the best candidate, the recent received beacon is stored. If the version received is not equal but greater and token count is larger than 1, it is immediately stored as the best candidate and the node returns to the **Electing candidate** state. From this state a node may receive a beacon or and update message. After a predefined

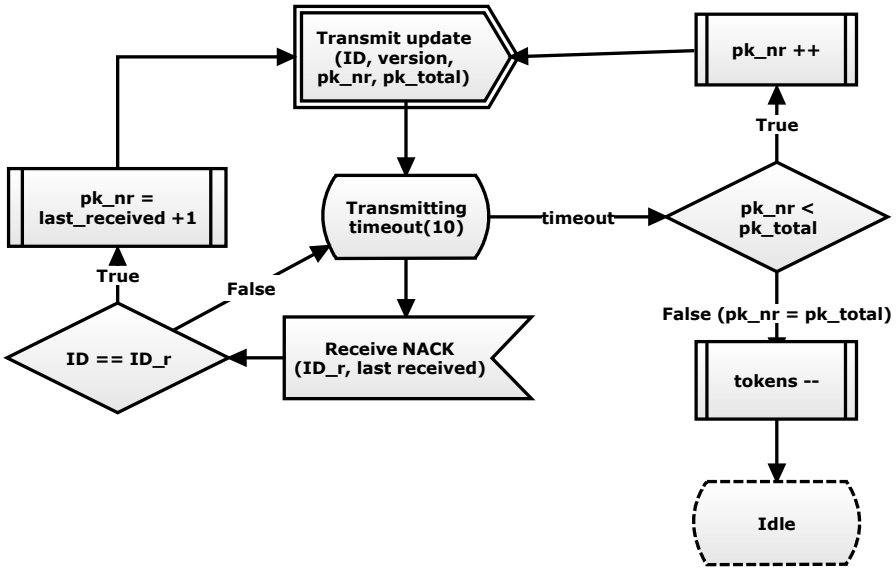


Figure 6.10: The transmission part of the eGCP protocol in detail.

period, a `request_update` is transmitted to the best candidate and the node returns to the **Idle** state.

Receive update An illustration of the "Receive update" process can be seen in Figure 6.12. From the start, the node receives an `update` message and verifies that the packet number matches the expected value and the version is newer. If this is the case the packet is stored, *expected* is incremented, the *nack_count* is reset and the node return to the **Waiting for packet** state. However, if the version is newer, but the packet number is lower than the expected, the *nack_count* is reset and the node returns to the **Waiting for packet** state. If the version is lower or equal to the nodes own, it will just return to the **Waiting for packet** state. In the **Waiting for packet** state a time-out occurs if the expected packet is not received within a predefined period. The node then checks its *nack_count* and compare it with a threshold value. If the value is lower, a `NACK` will be transmitted, the *nack_count* incremented and the node returns to the **Waiting for packet** state. If the *nack_count* value reaches the threshold value and a time-out happens, the update is discarded and the node returns to the **Idle** state. Finally, the *expected* value will match the *pk_total_r* which indicate all packets has been received. The node then create the software image from the packets, update itself and return to the **Idle** state.

Receive request_update From the **Idle** state a node can receive a `receive_update` message. When such a message is received and ID and token check is preformed. If it message was intended for the node and it has *tokens* left it will initialise its update and transmit the update. However, if the `receive_update` message was not intended for the node it returns to the **Idle** state.

6.3 Implementation of eGCP

As with the implementation of the GCP protocol we create a directory, which contains our eGCP source file and the appertaining header file. We have decided not to draw

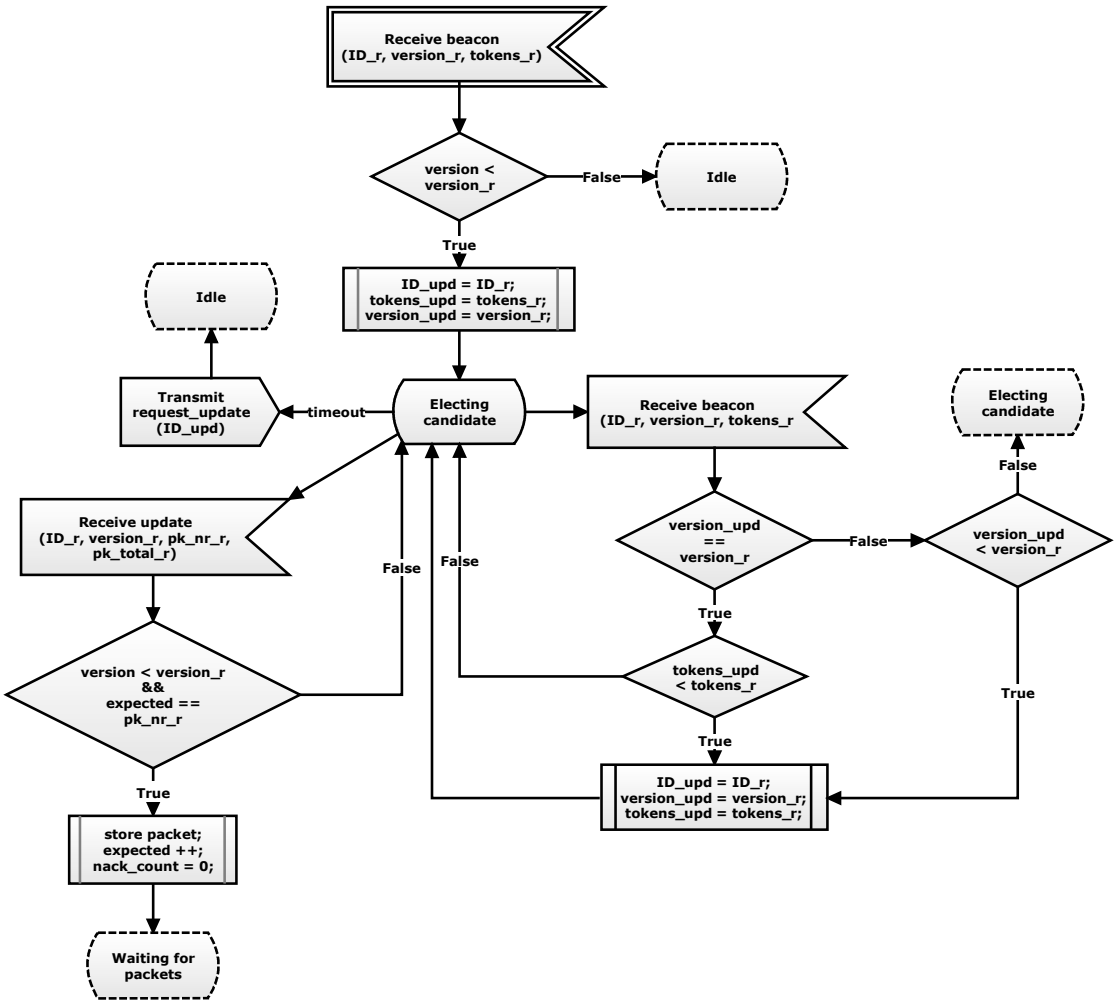


Figure 6.11: The reception of a beacon in the eGCP protocol.

a class diagram of eGCP because it is similar to the diagram of GCP protocol. The following explains what we have added to the eGCP protocol.

eGCPAgent. We have introduced 5 different timers that inherits from the abstract class TimerHandler. These timers are responsible for controlling execution of the functions such as *send_update*, *send_req*, *waiting_nack_timeout()*, *waiting_req_timeout()*, and *send_nack()*. The name of the function explains its purpose. The following describes timers. However the timers need to be detailed:

- *eGCPTimerWaitCNDT* controls the time given to choose the best candidate. When the time expires the node sends a request message to the candidate node.
- *eGCPTimerWaitNack* represents the amount of time time for a nack message from the destination node.
- *eGCPTimerWaitREQ* defines the time a node waits for request messages from other nodes. In this way the node that has an update will avoid implosion from nodes that have chosen it as an update source at the same.
- *eGCPTimerTriggerNack* controls the time given to send out a nack message if the node did not receive the expected message in the period of time.

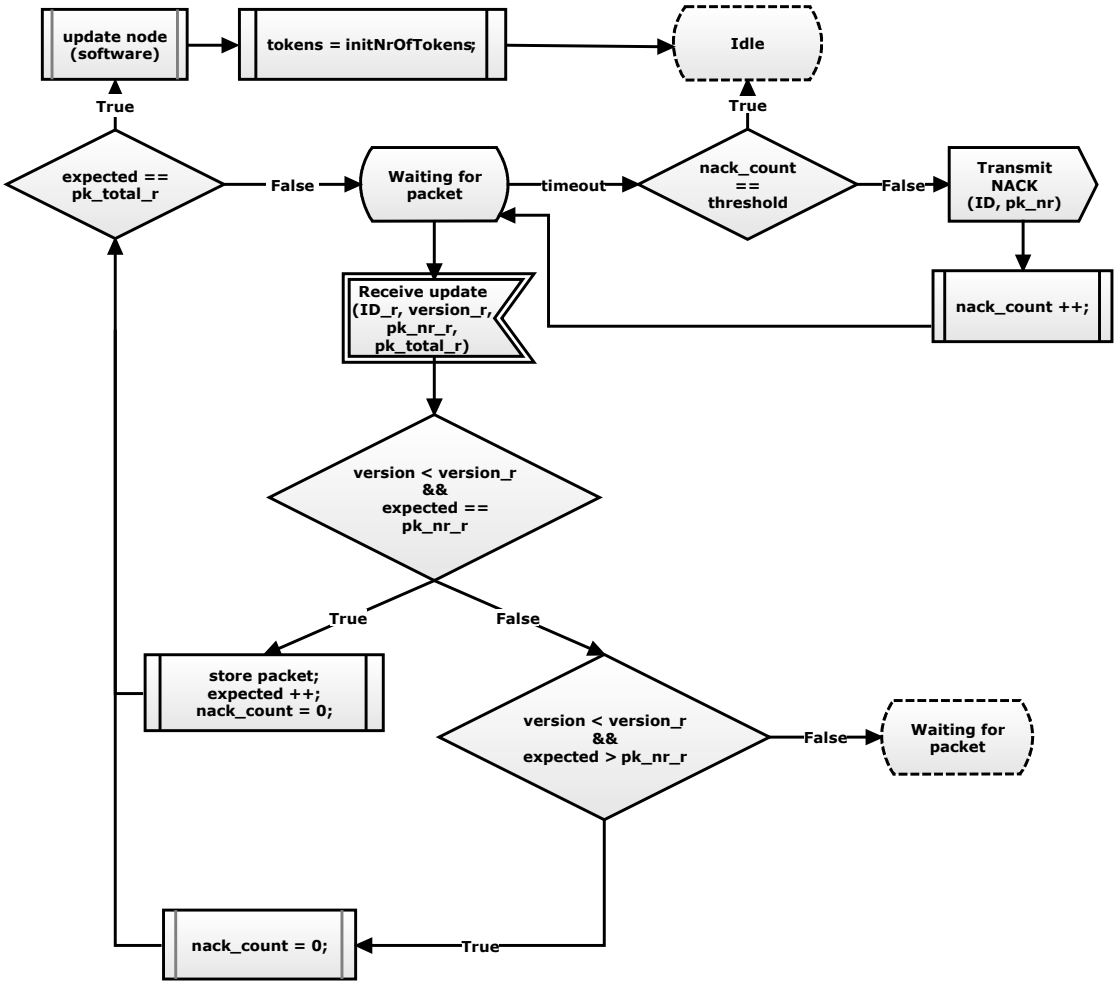


Figure 6.12: The reception of an update message in the eGCP protocol described in detail.

- *eGCPTimerSendBCN* defines the time given to send the beacon.

The send beacon method

In the OTCL simulation file we schedule the first beacon event. When the time comes, this event is dispatched and executed. This triggers the beacon send timer, which invokes the sending procedure every 100ms if the node is not busy (for instance, sending an update). If the node is busy and has tokens left, it is rescheduled to send a beacon after the next 100ms. The following Listing 6.1 shows the implementation of the sending procedure.

```
        // we have tokens and we are not busy
        if(tokens_ > 0 && allow_send_beacon() == true)
        {
4           Packet *pkt = allocpkt();          // create packet
            hdr_ip *iph = hdr_ip::access(pkt);
            hdr_egcp *egcph = hdr_egcp::access(pkt);

8           iph->daddr() = IP_BROADCAST;       // add info to header
```

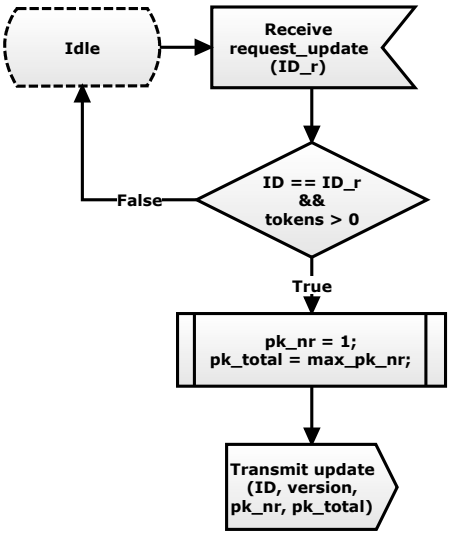


Figure 6.13: The reception of a request_update message in the eGCP protocol described in detail.

```
        iph->dport() = iph->sport();


12      egcph->tag = beacon;              // additional info
        egcph->version = version_;
        egcph->tokens = tokens_;

16      send(pkt, (Handler*)0);           // send packet
        tm_send_bcn.resched(0.1);        // reschedule new beacon
    }
    else if(tokens_ > 0)
20          tm_send_bcn.resched(0.1);  // new sending time
                                        // if we are busy
```

Listing 6.1: Send beacon procedure

The recv method

When a packet is handed to eGCP agent it is processed inside the recv method. The following describes what kind of actions have to be taken if the node receives a particular type of a message

Beacon received. When the node receives a beacon, it checks if it is not busy. If the timer is idle, the node saves the candidate and starts the timer for next possible candidates. When the next beacon arrives the node checks if it is a better candidate. When the time exceeds, the request message is sent to the best candidate. The following piece of code in Listing 6.2 demonstrates the procedure of choosing the best candidate.

```
    if(accept_beacon() == false)                    // busy
        {Packet::free(pkt); return;}
3
    if(tm_wait_cndt.status() == TIMER_IDLE)          // first candidate
    {
        save_candidate(hdrip, hdr, pkt);
7       tm_wait_cndt.resched(0.11);
```

```
        return;
    }

    if(tm_wait_cndt.status() == TIMER_PENDING)  // other candidates
    {
        if(version_upd == hdr->version)      // version is the same
            if(tokens_upd < hdr->tokens)  // another has more tokens
                save_candidate(hdrip, hdr, pkt);
            else {Packet::free(pkt); return;}
        else if(version_upd < hdr->version)  // another has newer version
                save_candidate(hdrip, hdr, pkt);
        else {Packet::free(pkt); return;}
    }
    return;
```

Listing 6.2: Save candidate procedure

Request received. When the node is chosen as a candidate it will receive a request message form another node. However, it is possible that when the node starts informing other nodes that it has newer version, it will be the chosen by many nodes at the same time, beacause the nodes will send request messages to that node. In order to avoid broadcasting more update messages the node waits 40ms. Then it will start an update process. The Listing 6.3 demonstrates how the request message is processed.

```
    if(accept_req() == false)          //busy
      {Packet::free(pkt); return;}
    // message for me
    if(hdrip->daddr() == here_.addr_ && tokens_ > 0)
    {
      Packet::free(pkt);
      pkt_nr_ = 1;

    // wait to receive more messages
    // and then launch the update process
      tm_wait_req.resched(0.04);
    }
    else Packet::free(pkt);
    return;
```

Listing 6.3: A request received procedure

Nack received. When a nack message is received, the node checks if the address matches its own. The next step checks whether the message arrived on time. If yes, the nodes saves a last received packet number, that came within the nack message. Then the node resends this packet.

```
    if(accept_nack() == false)                  // busy
       {Packet::free(pkt); return;}

    if(hdrip->daddr() == here_.addr_)        // packet addressed for me
    {   // nack received on time
       if(tm_wait_nack.status() == TIMER_PENDING)
       {
          pkt_nr_ = hdr->pkt_nr + 1;        // last received
          Packet::free(pkt);
             send_update();
       }
```

```
        else {Packet::free(pkt); return;}
    }
14  else {Packet::free(pkt); return;}
```

Listing 6.4: A nack received procedure

Update received. The following piece of code in Listing 6.5 demonstrates the processing steps when a node receives an expected packet. When the update packet with a right version is received , the node checks if it is necessary to stop the candidate election timer that was started in the choosing candidate state. Next, the packet number is compared with the expected. If yes, and it is the last packet, the node just resets variable values to the initial states, saves the whole update(changes the version). If it is not the last packet, the node saves the packet and waits for another one by rescheduling the sending nack timer.

```
    if(hdr->pkt_nr == 1)
2     cancel_candidate_timer();

    if(hdr->pkt_nr == expected)          // we got expected pkt
    {
6       pkt_nr_ = hdr->pkt_nr;
        pkt_total_= hdr->pkt_total;
        if(pkt_nr_ == pkt_total_)      // last pkt is received
        {
10          expected = 1;              // reset
            timeout_counter_ = 0;      // reset

            version_ = hdr->version;  // save an update
14          Packet::free(pkt);
            tm_trigger_nack.force_cancel();// all pkts are received.
        }                                  // Cancel timer
        else
18      {
            rm_nodeID = hdrip->saddr();     // save the source node
         rm_nodePort = hdrip->sport();  // id and port in order
                                          // to know where to send nack
22          expected++;
            timeout_counter_ = 0;
            tm_trigger_nack.resched(0.08); // wait for another packet
        }
26  Packet::free(pkt);
    }
```

Listing 6.5: An update received procedure

The simulation file

In the process of implementing the eGCP protocol we found that our previous approach of defining beacon timer events, and timers in general trough the OTCL script is unpractical. Although we could capture the behavior of the GCP agent, problems arise when multiple timers are introduced in the eGCP protocol. In GCP we use a single timer and updates are instant using only one packet. In eGCP there are states where we wait for a certain amount of time e.g. while electing the best candidate. A predefined beacon timer event trough OTCL would conflict with the behavior of the eGCP protocol, and possible send a beacon in a state where a node is dedicated to listening. We therefor

moved the handling of timers away from OTCL into the C++ implementation. The eGCP protocol is hereby captured within the C++ implementation of the eGCP Agent.

Agent initialization. The following piece of code shows the eGCP agent initialization. We set a number of tokens, packets, and attach the agent to the node.

```
for {set i 0} {$i < $val(nn)} {incr i} {
set p($i) [new Agent/eGCP]          # create agent
$p($i) set tokens_ $initToken       # init tokens
$p($i) set pkt_total_ 5             # set pkt no
$p($i) output name $cp_file         # set output file
$ns_ attach agent $node_($i) $p($i) # attach agent
}
```

Listing 6.6: Agent initialisation

Broadcast event scheduling. In the eGCP, as we see in Listing 6.7 we schedule only the first broadcast event in order to trigger the node to start sending the beacons. The following piece of code shows the scheduling process of the broadcast event.

```
for {set ii 0} {$ii < $val(nn)  } {incr ii} {                # all nodes
  set startTime [$randStart value]                           # time 1.0 1.1
  if {$ii == $randNum} {                                     # random node
    set startTime 1.000                                      # starts 1.0
    $ns_ at $startTime "$p($randNum) broadcast-beacon"       # add event
  } else { $ns_ at $startTime "$p($ii) broadcast-beacon"}
}
```

Listing 6.7: Start broadcasting process

6.4 Test of the extension

In order to test the eGCP protocol we devise different test scenarios. We conduct two different tests in order to validate the implementation of eGCP.

eGCP test 1

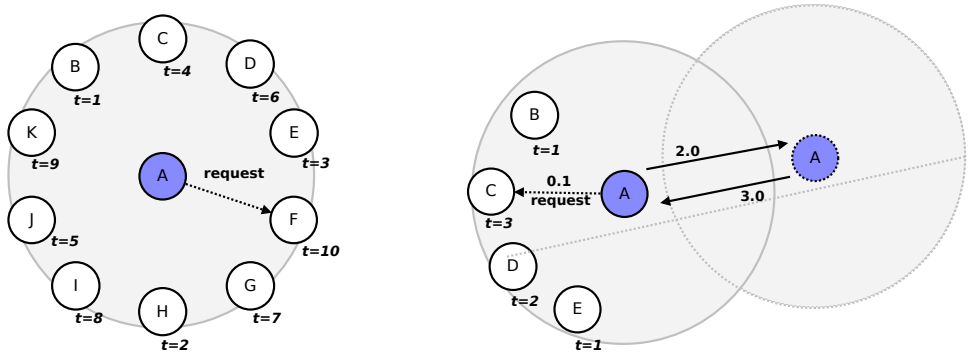
The first test is designed to verify the selection of the correct candidate, and to check if only 1 token is used in the update process. To test this, we place 10 nodes within the transmission range. Each of them has a unique token counts ranging from 1 to 10. The ten nodes are within reliable communication range. Then we place a node with a lower version in the same deployment area, which transmit a beacon. In the election period each of the candidates will send a beacon. Node ***A*** should choose the best candidate and request the update. The setup can be seen in Figure 6.14a.

GCP We expect node ***A*** to transmit the beacon. This will trigger an update from every node inside the deployment area and in total use 10 tokens.

eGCP We expect node ***A*** to initiate an election and choose the node with the highest amount of tokens. Each node deployed has a unique token number ranging from 1 to 10. This means that ***A*** should request the update from the node who has the 10 tokens available. After the simulation we expect to have node ***A*** updated with only 1 token used.

Results

The results matched the expectations. The GCP protocol used 10 tokens because each node reacted the the beacon and commenced an update. The eGCP protocol elected the node with the highest token count and after the update every node had their initial token values expect the candidate.



(a) The setup of the first eGCP test. In this test node ***A*** wakes up after a crash and discovers the entire neighbourhood has been updated in the mean time. Node ***A*** initiate an election and chose the best candidate.

(b) This is the setup of the second test. In this scenario, node ***A*** will start an election and chose a candidate from which the update process starts. In the middle of update, node ***A*** is instantly moved out of range of the candidate for later to be brought back. When node ***A*** gets back into range, the update process should go back and continue from the first missing packet. back.

Figure 6.14: An illustration of the the test scenarios for the eGCP protocol.

eGCP test 2

The second test is designed to check the reliability mechanism. In this test we have 5 static nodes, 4 of the nodes has a newer version than the fifth node ***A***. The update process starts but after receiving the first few packets of the update we instantly move node ***A*** away form the other nodes. This will trigger NACK-messages from ***A*** but no one can receive them. Then, before we reach the nack count threshold on the receiving node ***A*** and before the candidate finishes the update process, we instantly move node ***A*** back into range. Now, the NACK-message should be received thus ***A*** would receive the missing packets and complete the update. The setup can be seen in Figure 6.14b.

eGCP When the test is started we expect ***A*** to start an election and choose the best candidate. That candidate starts sending packages to ***A***. After we move ***A*** we expect to see it transmitting NACK-messages, while the candidate node continues to transmit the update. After a few NACK-messages ***A*** should return in range and transmit a NACK-message which the candidate node receives. The candidate node reacts on the NACK-message and sets the next packet number to the expected packet of the receiving node before continuing transmitting packets.

Results

The test results fits the expectations. With a transmission interval on 0,5 second per packet and 0,6 second timeout for NACK-message the candidate node transmitted 3 out of 8 packets before our receiving node gets removed. The receiving node transmit 2 NACK-messages before it was put into range of the candidate node which then received the 3 NACK-message. This reversed the update process which continued from packet 4 and the receiving node completes the update.

6.5 eGCP test results

The purpose with eGCP was to ensure update capabilities when the network was error-prone and improve the usefulness of tokens by choosing the best candidate. To test for these properties different scenarios are used with specific settings.

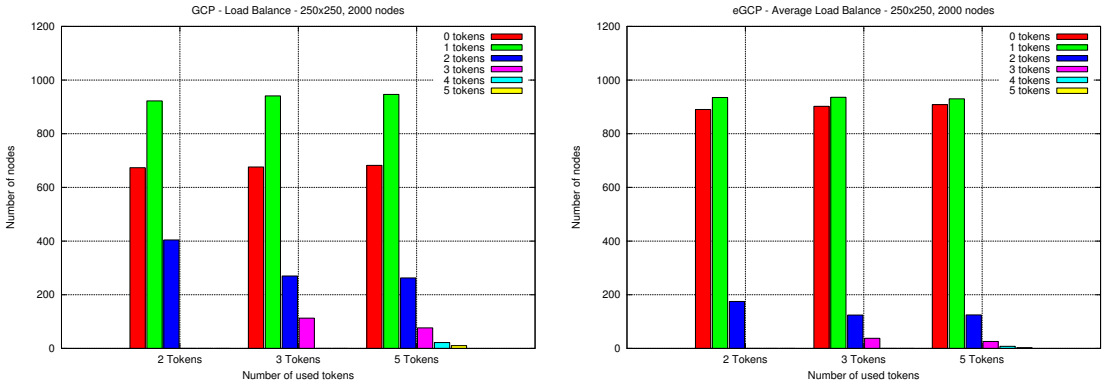
eGCP reliability mechanism

To test the reliability mechanism we increase the number of packets in the update from 1 to 10 and run the simulation in scenario 1. This scenario has been chosen because the density is sufficient to provide a reliable result compared with scenario 2 through 8. It should be noted that since the GCP protocol does not support updates spanning over multiple packets, thus no comparison between the protocols will be made.

Load balancing

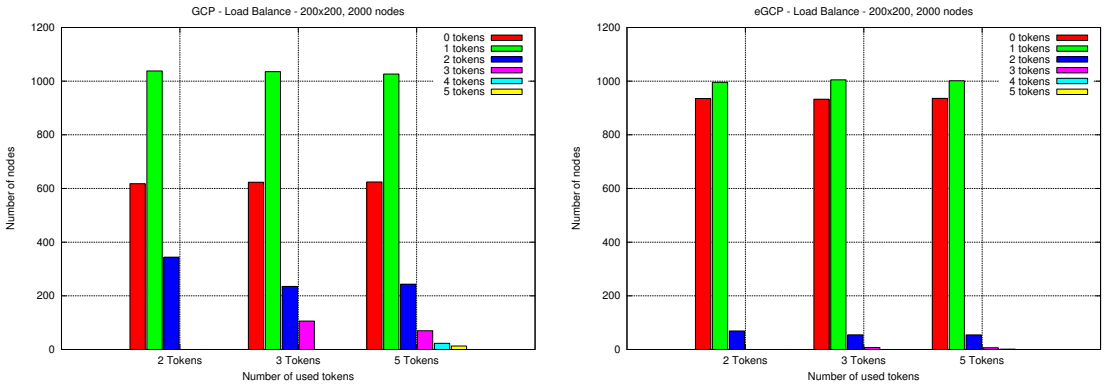
To see a difference in load balance between the GCP and eGCP protocol a fairly high density is needed. If the density is low the purpose of election the best candidate diminishes since only one node will be available for the majority of the time. Therefore, we chose scenario 1 which has the highest density. To get comparable results, the settings for the test are identical. By this we mean that instead of randomly deploying the source node, it is now placed so the only difference between the test runs are the protocol. Ten test runs with each protocol was created and the average load balance for both protocols can be seen in Figure 6.15(For larger pictures see Appendix E. It should be noted that both protocols updated equally many nodes, thought the eGCP has a slower code propagation speed. This means that the simulation time was extended until both protocols had updated more that 95% of the nodes. An example is our 300×300 scenario which took 100 seconds of simulation to update the entire network with the eGCP protocol. The GCP protocol did it in 35 seconds.

The results matches our expectations. Because of a rather high average density, nodes will have multiple nodes to choose between when electing a candidate. The total number of tokens used to update all nodes is smallest for the eGCP protocol. Also, the number of nodes who uses more than 1 token is halved compared to the GCP protocol. The results look very promising for the eGCP protocol. Unfortunately, with results from only one scenario a general opinion cannot be made thus we create two new scenarios alike that of scenario 1. The first is even more dense with a size of 200 × 200 meters and 2000 nodes. The second is more sparse with a deployment area of 300 × 300 meters and 2000 nodes. In the following, we will refer to them as scenario 1d (**d**ense) and 1s (**s**parse). With identical settings as in the previous test case we conduct the test 10 times in both scenario 1d and 1s. The average results are depicted in Figure 6.16 and 6.17.



(a) The GCP protocol. (b) The eGCP protocol.

Figure 6.15: An illustration of the average load balancing for the two protocols in scenario 1 (250×250).

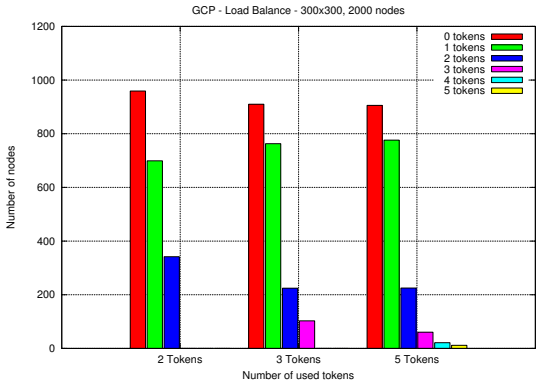


(a) The GCP protocol. (b) The eGCP protocol.

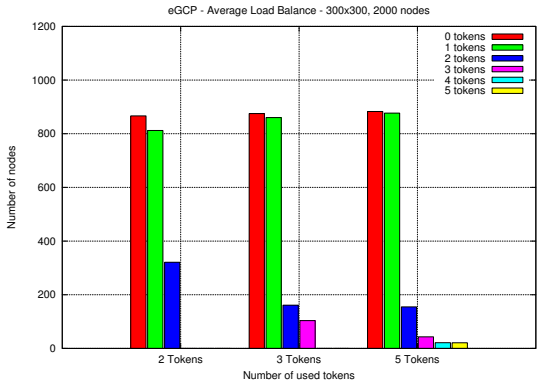
Figure 6.16: An illustration of the average load balancing for the two protocols in scenario 1d (200×200).

In scenario 1d(Figure 6.16), we see the best overall result of the eGCP protocol. Compared to the GCP protocol, our extension rarely uses more than 1 token in average to perform the same amount of updates. This results matches our expectations because this scenario is very dense, thus our election should have multiple nodes to chose between. This support our claim that in dense networks our extension provides a better load balancing compared to the GCP protocol.

Looking at scenario 1s(Figure 6.17) we see the eGCP protocol still provides better average load balance because the number of nodes which uses more than 1 token is less than in the GCP protocol. However, our results are beginning to converge towards those of GCP. This converging also matching our expectations. With a more sparse cluster, the load balancing between the GCP and eGCP protocol will become more equal. At some point, the average density becomes so low, that they will be equal in terms of load balancing. It should be noted though, that the code propagation speed will be higher. At which average density they have an equally load balancing we have not determined. For practice use a hybrid solution could be interesting. Before initiating an update the average density could be calculated and the best approach could be used. If the average



(a) The GCP protocol.



(b) The eGCP protocol.

Figure 6.17: An illustration of the average load balancing for the two protocols in scenario 1s (300×300).

density was lower than a threshold, election would be omitted and updates would occur as in the GCP protocol. Opposite, if the average density is higher than the threshold, the eGCP could be used if the slower code propagation is acceptable.

The results from scenarios Figure 6.15, Figure 6.16 and Figure 6.17 has been summarised in Table 6.2.

GCP 2 tokens				**eGCP** 2 tokens			
	200×200	250×250	300×300		200×200	250×250	300×300
0	618.1	673.4	959.3	0	935.3	890.3	866.6
1	1037.9	922.3	699	1	995.7	934.9	812.2
2	344	404.3	341.7	2	69	174.8	321.2
GCP 3 tokens				**eGCP** 3 tokens			
	200×200	250×250	300×300		200×200	250×250	300×300
0	623.5	676.2	910.1	0	932.7	902.2	875.2
1	1035.5	941.1	762.9	1	1005	935.9	860.2
2	235.1	269.8	224.4	2	54.7	124.3	161.1
3	105.9	112.9	102.6	3	7.6	37.6	103.5
GCP 5 tokens				**eGCP** 5 tokens			
	200×200	250×250	300×300		200×200	250×250	300×300
0	624.2	682.1	905.8	0	936	908.7	883
1	1026.4	946.6	776.3	1	1001.6	930.1	876.9
2	243.5	262.6	224.9	2	54.4	124.9	154.8
3	70	76.5	60.3	3	6.7	26	43.1
4	22.9	21.9	21.2	4	1.2	7.7	21.2
5	13	10.3	11.5	5	0.1	2.6	21

Table 6.2: The average numbers for the GCP and eGCP protocol doing 10 runs with 2,3 and 5 tokens in scenario 1, 1d and 1s.

6.6 Summary

In this chapter we have discussed the problems in the GCP protocol, especially the assumption with reliable communication within wireless networks. This assumption is not realistic, so we propose an extension to the GCP protocol which removes this limitation by specifying a reliability mechanism. We also improve the utilisation of the token which improves the load balancing in dense networks. This is done by introducing elections and choosing the best candidate for transmitting the update. We give a systematic specification of our extension (called eGCP) with the same methodology that we used to define the GCP protocol. Then we implement the extension and verifies it behaves as expected. To do this we devise two small tests. Finally, we compare our eGCP protocol with the GCP. As expected our protocol performs better as the network becomes more dense. This can be seen in the way the load balancing is distributed. The eGCP protocol uses less tokens to update all nodes compared with the GCP protocol. However, the trade-off is a slower code propagation due to all the elections. It should also be noted that if the density of the network is low, the difference between the GCP and eGCP protocol vanishes, since the election process needs more than 1 node in the neighbourhood to become effective.

Conclusion

Being able to update the software of WSNs is becoming a necessity as WSN applications are often deployed in hard to reach locations, or harsh environments. The energy resource of a sensor node is a scarce resource and the efforts in minimizing the power consumption is critical to the lifetime of a WSN application. Updating a WSN is an energy consuming task and devising energy efficient update mechanisms is not a trivial task. One way of minimizing power consumption is by ensuring even load balance among the sensor nodes distributing the update, this way the overall lifetime of the network is prolonged.

In this project we gave a systematic presentation of the GCP protocol using the protocol specification methodology of [12]. We also implemented the GCP protocol in the simulation tool NS2 to reproduce and verify the very promising test results in [9] in terms of code update propagation speed and load balancing. Furthermore, we modelled the GCP protocol in UPPAAL for formal verification of the protocol, and performed time- and energy analysis on selected synthetic topologies. As a supplementary tool to the NS2 simulator we implemented the python application PyMove that generates random node movement for user defined cluster scenarios.

While analysing and implementing the GCP protocol according to [9] we encountered inadequate and ambiguous protocol descriptions. We also discovered that the protocol did not support realistic wireless communication since it assumes communication always occur without collision and any kind or data loss. Finally, we encountered different scenarios where multiple tokens were used instead of the needed one. This contradicts the purpose of the tokens.

We proposed and implemented an extension to the GCP protocol called eGCP. Our protocol includes a reliability mechanism, which ensures that nodes receive updates despite faulty communication. The mechanism is designed with resource constraints in mind and thus uses NACK instead of ACK-messages. This will lower the amount of protocol messages and therefore save battery power. It also includes an election mechanism which further improves the load balancing by choosing the best candidate based upon version number and remaining tokens. This provide an even better utilization of the tokens, while minimise the possibility of implosion. A simple form of resource awareness is added to the node. Before broadcasting a beacon, nodes will verify they have tokens left to perform an eventual update. If they do not have any tokens left, they will not transmit a beacon, until they eventually receive a new update and reset their token count.

The results: GCP

Our test results using the NS2 tool under the same assumptions given in [9], did not correspond to the results presented. Our test results showed that the code propagation speed in scenario 1 was off by factor of more than 40. In the test results for scenario 2 trough 8 we updated a maximum of 11 nodes in the individual scenarios, rendering these result unsuitable for further comparison to the results in [9]. In the search of a plausible explanation of the variation in test data, we tried to replicate the results in [9] trough changing simulation parameters. We succeeded in replicated some of the result in [9] by modifying the range and speed parameters. To achieve similar results as in [9] scenario 1, we had to increase the movement speed of the nodes to a maximum of 20 meters per second (or 72 km/hour) and increased the radio range to a maximum of 10 meters. For replication of scenario 2, 5 and 6 even higher settings were needed to

duplicate the original results. These settings do not resemble those presented [9], which indicate that the results in [9] are incorrect under the given simulation parameters.

We have identified best case scenarios which also indicates that unless the radio and movement settings are changing or the topology is not uniformly distributed their results are very optimistic.

Another strong indication that the results provided in [9] are not accurate, is based on the shape of their graphs. For instance, scenario 6 is *4 socializing cluster scenario* in which a staircase shaped graph would be expected. After one cluster has been updated, a transmitter will need to travel 200 meters to reach the next cluster hereby leaving the number of updated nodes stagnant for this period of time. The original test results in [9] show is a smooth graph.

Although it may be theoretically possible to update the network given a highly non-uniform node deployment, it is very unlikely. Therefore, we conclude that the result in [9] does not reflect the average behaviour of the GCP protocol under the assumptions given in [9].

The results: eGCP

We designed the eGCP protocol to improve the load balancing and minimising the chance for implosion by choosing the best candidate for transmitting the update. In scenario 2 through 8 the average density is low compared to the radio range thus the chance for having more that 1 node to select as candidate is very small. This degrades the performance of the eGCP protocol compared to GCP because though the load balancing will be very similar in these cases, but the code propagation speed will be slower since an election has to be made before each update.

The density is much higher in scenario 1, and in this test case, the load balancing is clearly improved. The amount of nodes who uses the maximum of tokens is very small compared with the GCP protocol. In every test the result is the same. Especially, in the tests with 5 initially tokens we see that the eGCP protocol does not have a single node who uses all tokens and almost none which uses 4. Overall the results show the load balancing is more evenly distributed. Also the total number of tokens used to perform the update is lower compared to the GCP protocol.

In our own test scenario we created a very dense cluster. This further improved the results of the eGCP protocol compared with the GCP protocol. From our results in Table 6.2 we see that our eGCP protocol outperforms the GCP protocol in dense scenarios. In scenario 1 (2 tokens) we see that the GCP protocol has 404,3 nodes which uses 2 tokens compared with the eGCP which only has 174,8. The GCP protocol has over twice the amount of nodes which uses all tokens. The eGCP protocol performs even better when the density increases. In scenario 1d (dense - 200x200) the GCP protocol has 344 nodes which uses both their tokens. eGCP has only 69 nodes which uses both tokens. The same characteristic applies for 3 and 5 tokens. In scenario 1d with 3 tokens, the eGCP protocol has 7,6 nodes which uses all 3 tokens. The GCP protocol has 105,9 nodes. The difference is a factor of more than 12.

To conclude the eGCP protocol provides better load balancing than the GCP protocol, if the density is high enough. If the density is low the load balancing would be very similar between the two protocols, but the code propagation speed of eGCP would be slowest.

We have concluded that our extension to the GCP protocol will improve the load balancing when distributing an update in dense WSN. We also concluded that knowledge

of the density of the network provides valuable information when choosing the code update dissemination protocol.

7.1 Future work

While designing and implementing the eGCP protocol various ideas emerged which was left for future work. In this section we describe different approaches which could improve our eGCP protocol.

Buffer management

As of now, packets out of order are discarded and a NACK message is replied back. An improvement could be to include a mechanism to control the buffer, and thus make sure that the missing packets would be retransmitted. The "sliding window" mechanism hinted in [20] could be a possibility. The window has a frame from where it can store successfully received packets up to the size of a given offset. Applying the sliding window would require some changes to the usage of NACK message. Instead of transmitting a NACK message as soon as a packet loss is detected the mechanism would have to wait until the packet becomes the first in the frame. This solution would also be useful if a node was receiving an update but moved out of range of the source node.

Interrupted updates

As a consequence of the moving nodes, situations will occur when a node will move into the vicinity of an ongoing update. The node will transmit a NACK message to indicate they missed all previous packages and then continue on its path and perhaps move out of range. If the source node received this NACK message the update process will be unnecessarily restarted. This will result in slower propagation speed and increased battery usage. The chance for this scenario to occur is low though, since it will only happen in the vicinity of the transmission range thus chance of reception is only around 30 %. This means that the "trespassing" node might not get the update messages, and if it does, the source node might not receive the NACK message. The chance for this scenario to occur increases with the size of the update, since the total update time will increases as well. A way to contain this problem could be to define a threshold on the packet number. The trespassing node that received the update will evaluate if the number of transmitted packages compared to the total number is too high to interrupt the ongoing update. For instance, if a node evaluates that the update process is already more than 20% completed it will not NACK and hereby reset the update process.

Timing

In the eGCP protocol timers has been used to control when the best candidate should be elected, wait for requests messages and when a NACK message should be transmitted. We have made no tests or analysis to discover which numbers give the optimal performance. Future work could include such investigation which could lead to achieving higher propagation speed.

Token drops

A problem occurs if a node crashes in the middle of receiving an update. The source node cannot know this since no NACK-messages will be send and thus after transmitting

the update decrement its token count. The same problem arise if a node is receiving an update but moves out of range of the source node. Again, the source node will act like the update was successfully received and decrement its token count. This could be a problem when the source node (only node containing the new version) is deployed. If the node uses all tokens without updating any other node, the update will never be distributed.

Avoiding update request implosion

Whenever a multiple number of nodes receives a beacon from one node with a newer version, the election timers will start on all the surrounding nodes almost simultaneous. When the candidate election timer expires the nodes will request an update from the same candidate, at the same time causing implosion on the candidate node. Implosion will can cause the update request to be lost, and an additional beacon period will pass before a new election start.. the same thing can happen continuously until the topology of the moving nodes resolves the problem. To resolve the problem we propose that each node append small random delay to the election period, this minimize the changes that two or more nodes request the update from the candidate at the same time hereby causing implosion.

Bibliography

[1] 8-bit Microcontroller with 128k Bytes In-System Programmable Flash. `www.atmel.com/dyn/resources/prod_documents/doc7679.pdf`.

[2] 8-bit Microcontroller with 16k Bytes In-System Programmable Flash. `http://www.atmel.com/dyn/resources/prod_documents/doc2545.pdf`.

[3] Avr910: In-system programming. `http://www.atmel.com/atmel/acrobat/doc0943.pdf`.

[4] Network simulator 2 documentation. `http://www.auto-nomos.de/ns2doku/`.

[5] nrf24l01 single chip 2.4ghz transceiver product specification. `http://www.nordicsemi.no/files/Product/data_sheet/nRF24L01P_Product_Specification_1_0.pdf`.

[6] The ns2 manual. `http://www.isi.edu/nsnam/ns/ns-documentation.html`.

[7] Tinyos. `http://www.tinyos.net/`.

[8] A. Arora, P. Dutta, S. Bapat, V. Kulathumani, H. Zhang, V. Naik, V. Mittal, H. Cao, M. Demirbas, M. Gouda, Y. Choi, T. Herman, S. Kulkarni, U. Arumugam, M. Nesterenko, A. Vora, and M. Miyashita. A line in the sand: a wireless sensor network for target detection, classification, and tracking. *Computer Networks*, 46(5):605 – 634, 2004. Military Communications Systems and Technologies.

[9] Yann Busnel, Marin Bertier, Eric Fleury, and Anne-Marie Kermarrec. Gcp: gossip-based code propagation for large-scale mobile wireless sensor networks. In *Autonomics '07: Proceedings of the 1st international conference on Autonomic computing and communication systems*, pages 1–5, ICST, Brussels, Belgium, Belgium, 2007. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[10] Chih-Chieh Han, Ram Kumar, Roy Shea, and Mani Srivastava. Sensor network software update management: a survey. *Int. J. Netw. Manag.*, 15(4):283–294, 2005.

[11] Wendi Rabiner Heinzelman, Joanna Kulik, and Hari Balakrishnan. Adaptive protocols for information dissemination in wireless sensor networks. pages 174–185, 1999.

[12] Gerard J. Holzmann. *Design and validation of computer protocols.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[13] David B. Johnson and David A. Maltz. Dynamic source routing in ad hoc wireless networks. In *Mobile Computing*, pages 153–181. Kluwer Academic Publishers, 1996.

[14] Pedro José Marrón, Matthias Gauger, Andreas Lachenmann, Daniel Minder, Olga Saukh, and Kurt Rothermel. Flexcup: A flexible and efficient code update mechanism for sensor networks. In *In Proceedings of the Third European Workshop on Wireless Sensor Networks (EWSN 2006*, pages 212–227, 2006.

[15] K. Martinez, J.K. Hart, and R. Ong. Environmental sensor networks. *Computer*, 37(8):50–56, Aug. 2004.

[16] Robertas Backys Peter Finderup, Thomas Birk Abildgaard. Energy efficiency in wireless sensor networks and exploration of potential hardware. Technical report, Aalborg University, 2009.

[17] The CMU Monarch Project. The cmu monarch project's wireless and mobility extensions to ns. `www.monarch.cs.cmu.edu`, 1998.

[18] T.S. Rappaport et al. *Wireless communications: principles and practice.* Prentice Hall PTR New Jersey, 2002.

[19] Niels Reijers and Koen Langendoen. Efficient code distribution in wireless sensor networks. In *WSNA '03: Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, pages 60–67, New York, NY, USA, 2003. ACM.

[20] Thanos Stathopoulos, John Heidemann, and Deborah Estrin. A remote code update mechanism for wireless sensor networks. Technical report, 2003.

[21] Andrew S. Tanenbaum. *Computer Networks.* Prentice-Hall, Inc., Upper Saddle River, New Jersey, USA, 2002.

[22] Ekram Hossain Teerawat Issariyakul. *Introduction to network simulator NS2.* Springer, SpringerScience+BusinessMedia,LLC,233SpringStreet,NewYork,NY 10013,USA, 2009.

[23] Uppsala University and Aalborg Unicersity. Uppaal.

Appendix A

Calculating average density

Test scenario	Size of area (m)	Result (m^2)	Result (km^2)
1	250 x 250	62.500	0,0625
2	1100 x 1100	1210.000	1,21
3	800 x 800 x 2 - 100 x 100	1270.000	1,27
4	2000 x 2000	4000.000	4,00
5	550 x 550 x 2	1210.000	1,21
6	1300 x 1300	1690.000	1,69
7	800 x 800	640.000	0,64
8	1500 x 1500	2250.000	2,25

Table A.1: A table with the different sizes of the deployment areas used in the test scenarios.

Test scenario	Area (km^2)	Number of nodes	Result (nodes/km^2)
1	0,0625	2000	32000
2	1,21	2000	1653
3	1,27	2000	1575
4	4,00	2000	500
5	1,21	2000	1653
6	1,69	2000	1183
7	0,64	2250	3516
8	2,25	2250	1000
Total for 2nd to 8th scenarios			11080

Table A.2: A table displaying the density of nodes pr. km^2 in each test case.

It should be noted that the numbers of nodes stated in Table A.2 are the combined number of nodes together with the number of *transmitters* from the different test cases. It should also be noted that we omit the first test case from the scenario because the deployment area is so confined and thus will give misleading numbers to the normal average.

The average density for the seven test cases are: $\frac{11080 nodes/km^2}{8} \approx 1385$ nodes pr. km^2.

UPPAAL models

B.1 UPPAAL model of the original GCP protocol

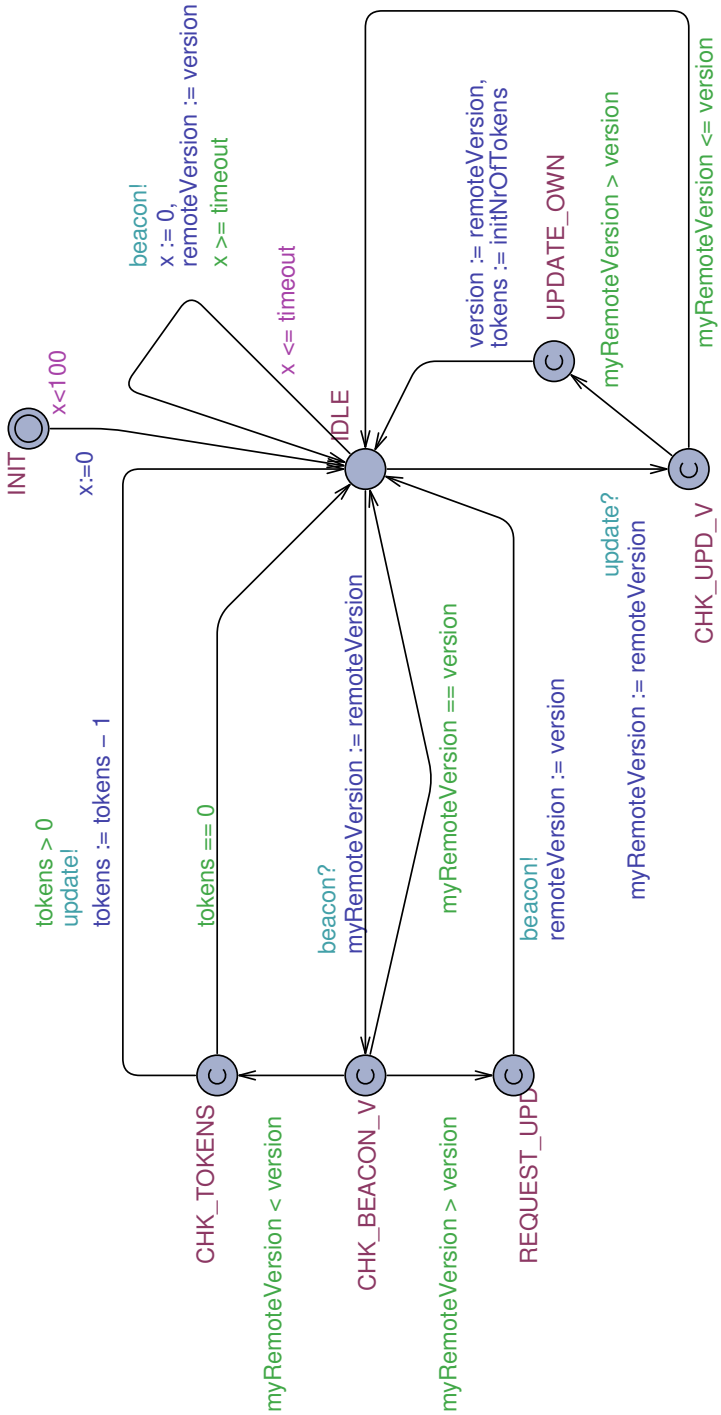


Figure B.1: Initial GCP UPPAAL model.

B.2 UPPAAL model of the original GCP protocol with a half duplex radio

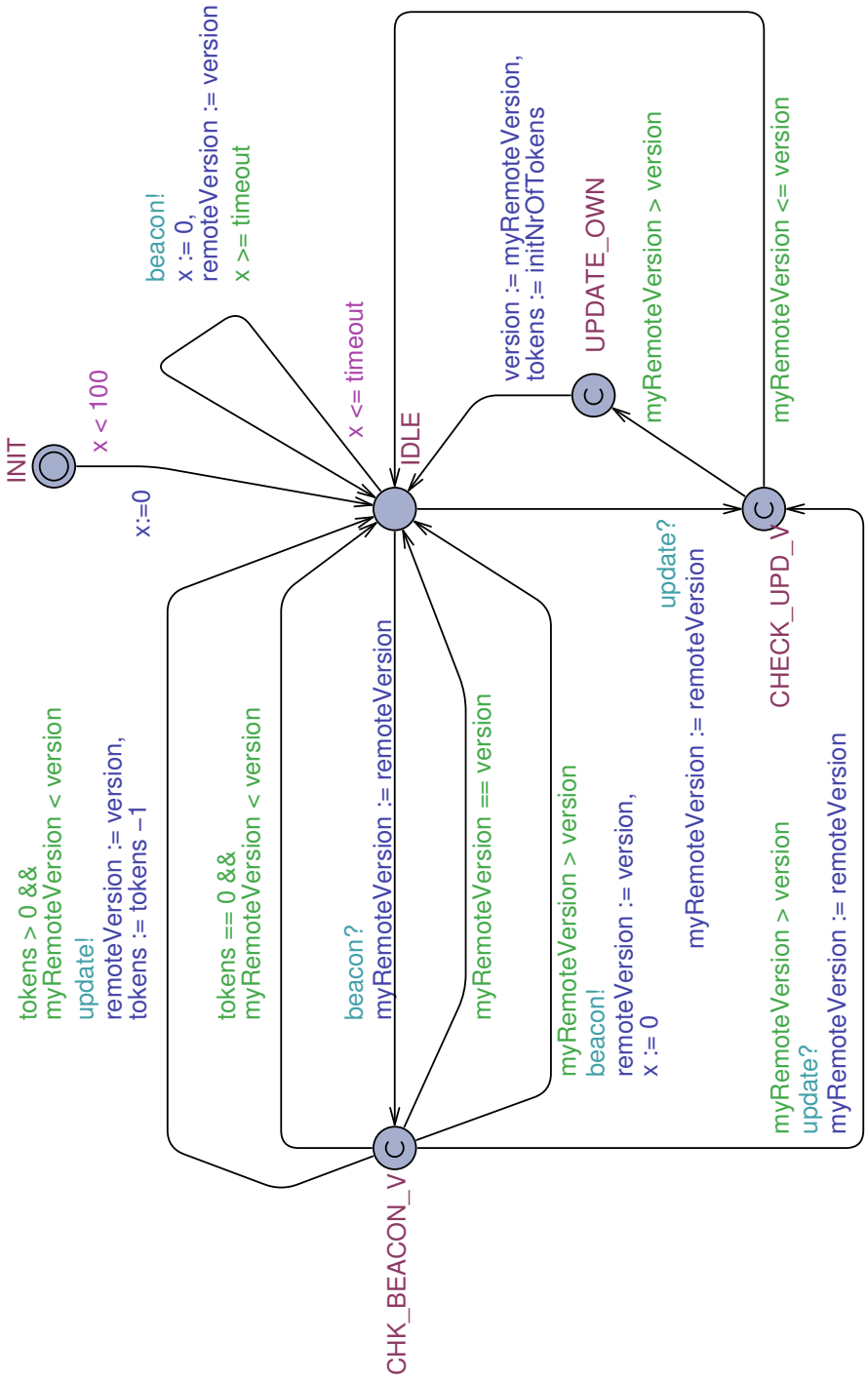


Figure B.2: GCP UPPAAL model - half duplex.

B.3 UPPAAL model of the original GCP protocol with time and node connectivity

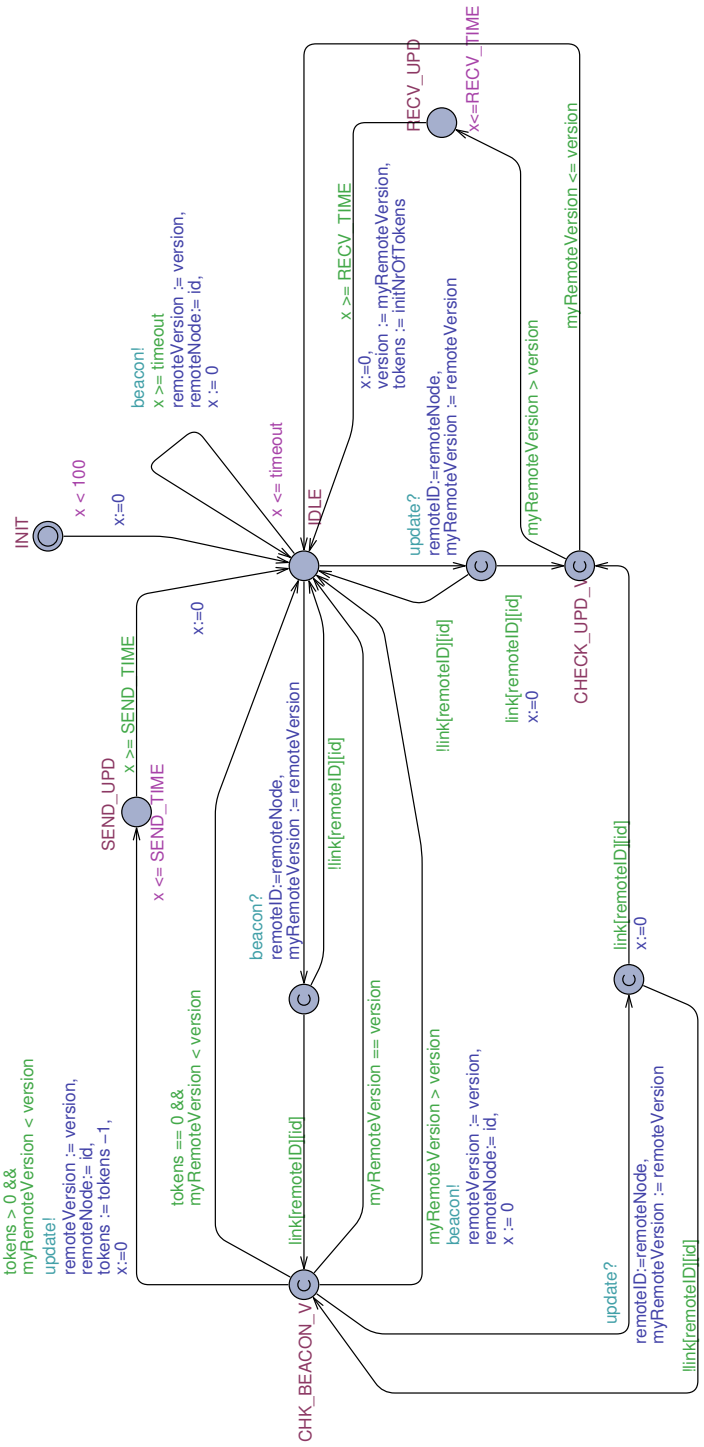


Figure B.3: GCP UPPAAL model - Time and Node connectivity.

B.4 UPPAAL model of the original GCP protocol with energy, time and node connectivity

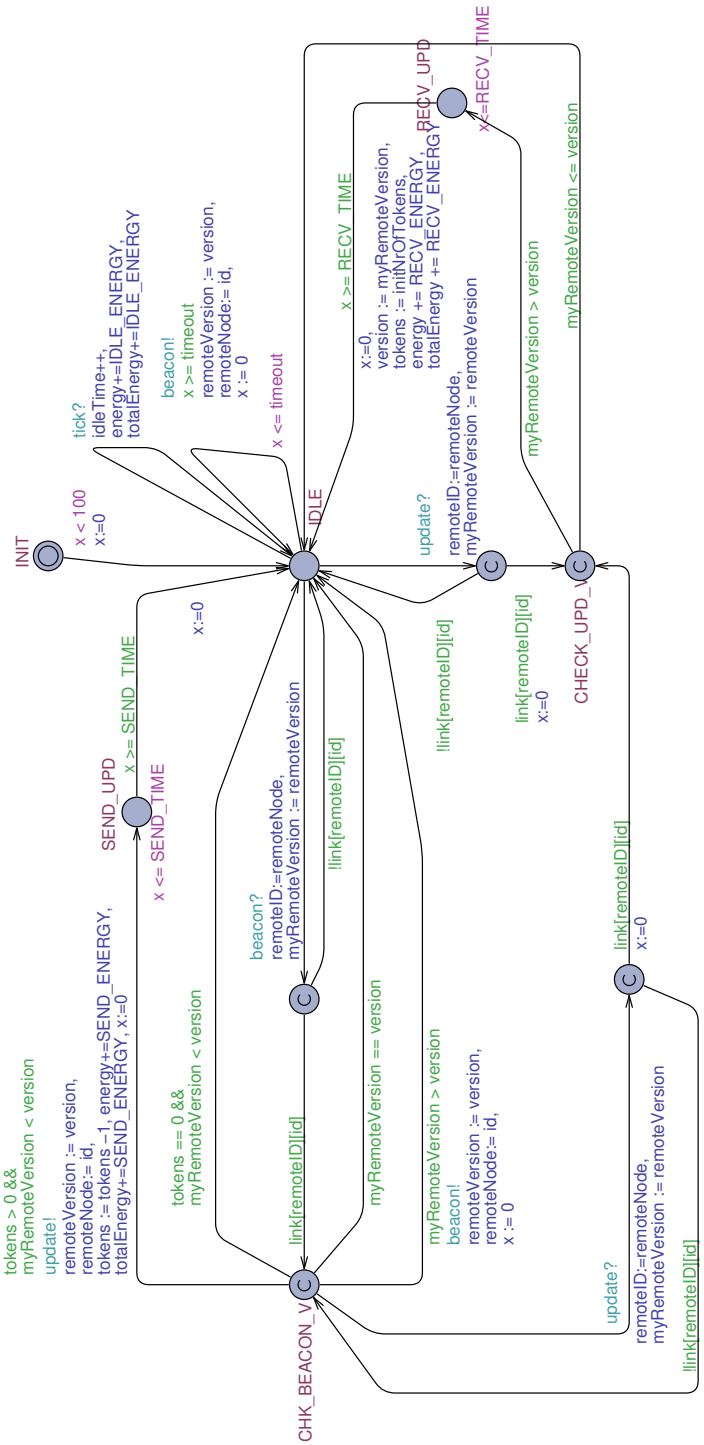


Figure B.4: GCP UPPAAL Energy, Time and Node connectivity.

B.5 Maple Time and Energy calculations

PageWriteTime := 0.0045; Pages := 248; PageSize := 256;

$$0.0045$$
$$248$$
$$256 \quad (1)$$

BufferReadTime := $\frac{8 \cdot 2}{16 \cdot 10^6} \cdot \frac{\text{Pages} \cdot \text{PageSize}}{2}$;

$$\frac{496}{15625} \quad (2)$$

BufferWriteTime := PageWriteTime·Pages;

$$1.1160 \quad (3)$$

TotalRecieveTime := (2·BufferWriteTime) + BufferReadTime;

$$2.263744000 \quad (4)$$

TxSpeed := 2000000; UpdateSize := Pages·PageSize·8;

$$2000000$$
$$507904 \quad (5)$$

TxTime := $\frac{\text{UpdateSize}}{\text{TxSpeed}}$; = $\frac{3968}{15625}$

Vradio := 3.3; Vmcu := 4.5; Irx := 0.0115; Itx := 0.007; ItxIdle := 0.000320; *Imcu* := 0.027;

$$3.3$$
$$4.5$$
$$0.0115$$
$$0.007$$
$$0.000320$$
$$0.027 \quad (6)$$

Jrx := Vradio·Irx·TxTime;

$$0.009637478400 \quad (7)$$

Jtx := Vradio·Itx·TxTime;

$$0.005866291200 \quad (8)$$

JtxIdle := Vradio·ItxIdle·(BufferWriteTime − TxTime);

$$0.0009103226880 \quad (9)$$

JrxIdle := Vradio·ItxIdle·(TotalRecieveTime - TxTime);

$$0.002122340352 \quad (10)$$

MCU assumed to be active for the duration of the recievers bufferWriteTime.

JmcuSend := Vmcu·Imcu·BufferWriteTime;

$$0.13559400 \quad (11)$$

JmcuRecv := Vmcu·Imcu·TotalRecieveTime;

$$0.2750448960 \quad (12)$$

$\textbf{Energy}_{\textbf{SEND}}$:= Jtx + JtxIdle + JmcuSend

$$0.1423706139 \quad (13)$$

$\textbf{Energy}_{\textbf{RECV}}$:= Jrx + JrxIdle + JmcuRecv;

Figure B.5: Maple worksheet

GCP results

C.1 GCP simulation results - Scenario 1

Overview of scenario 1

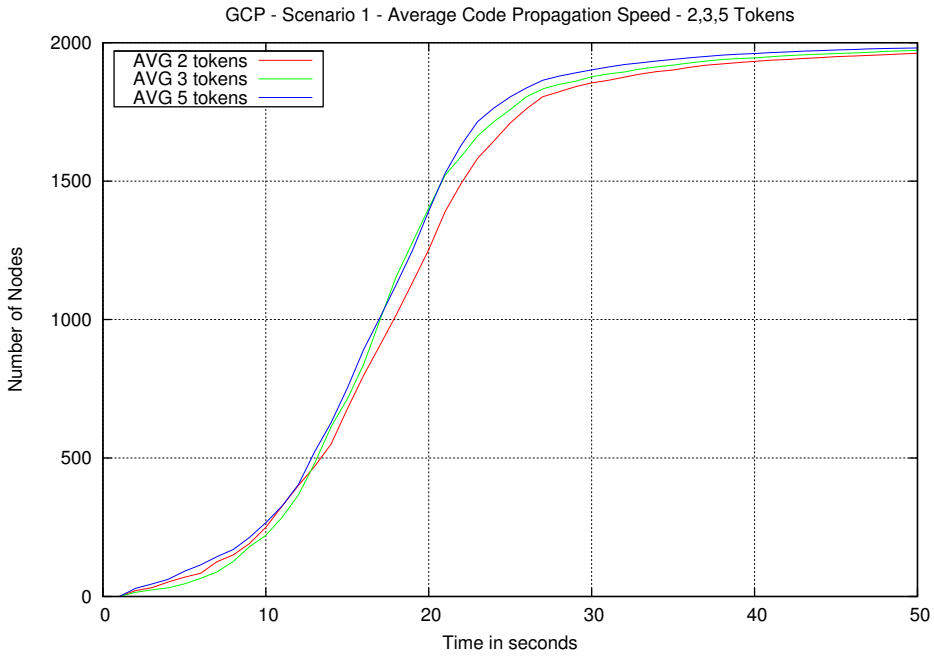


Figure C.1: Scenario 1: average code propagation speed.

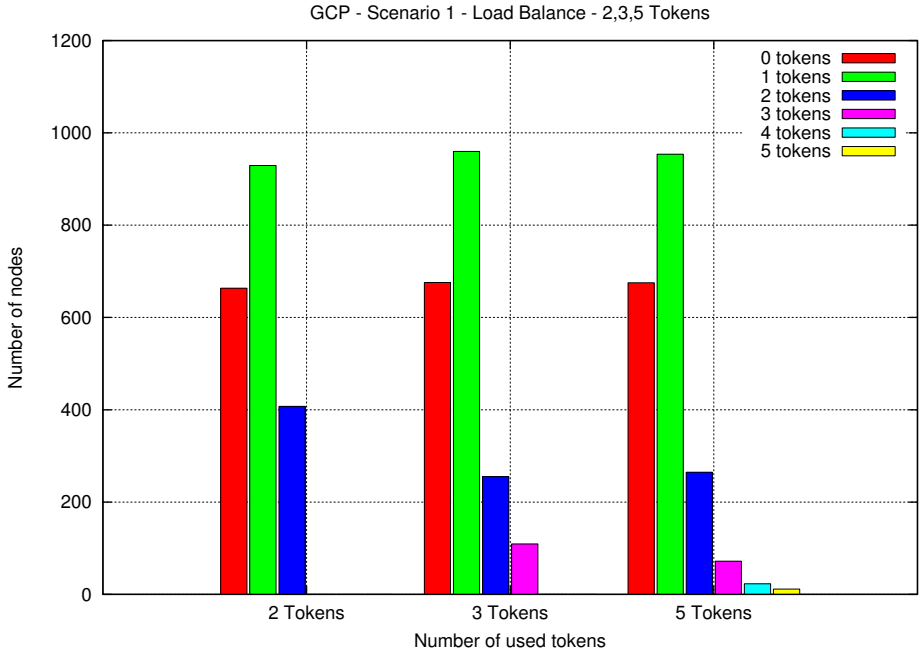


Figure C.2: Scenario 1: average load balance.

Scenario 1, 10 runs with 2,3 and 5 tokens

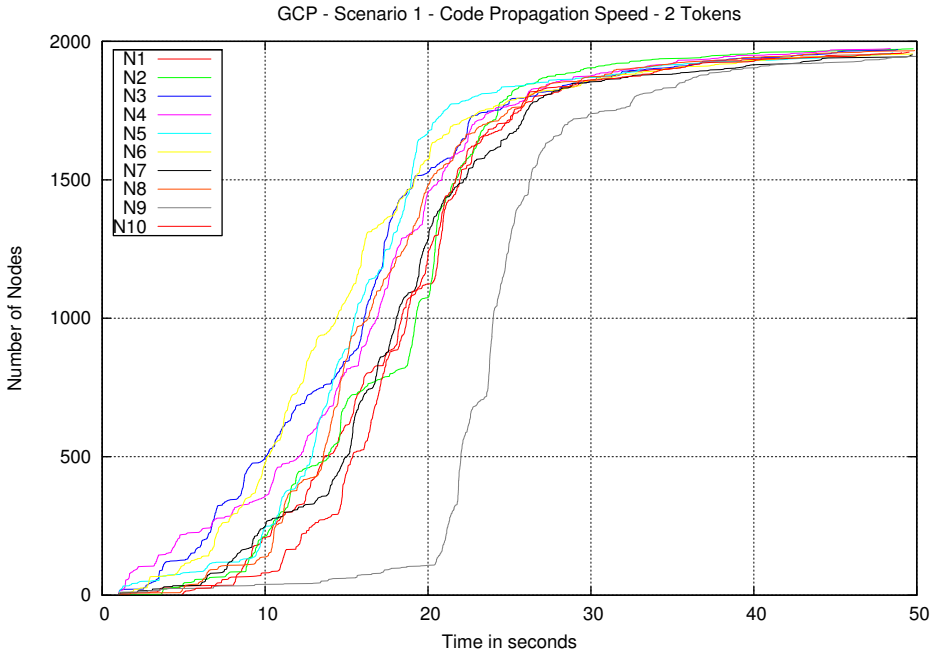


Figure C.3: Scenario 1: 10 runs, 2 tokens.

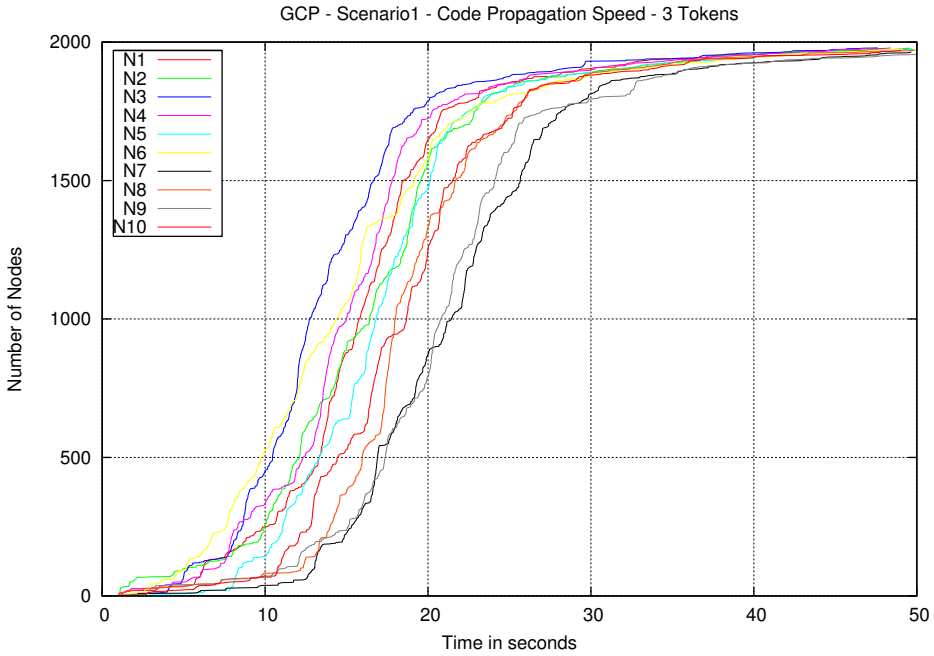


Figure C.4: Scenario 1: 10 runs, 3 tokens.

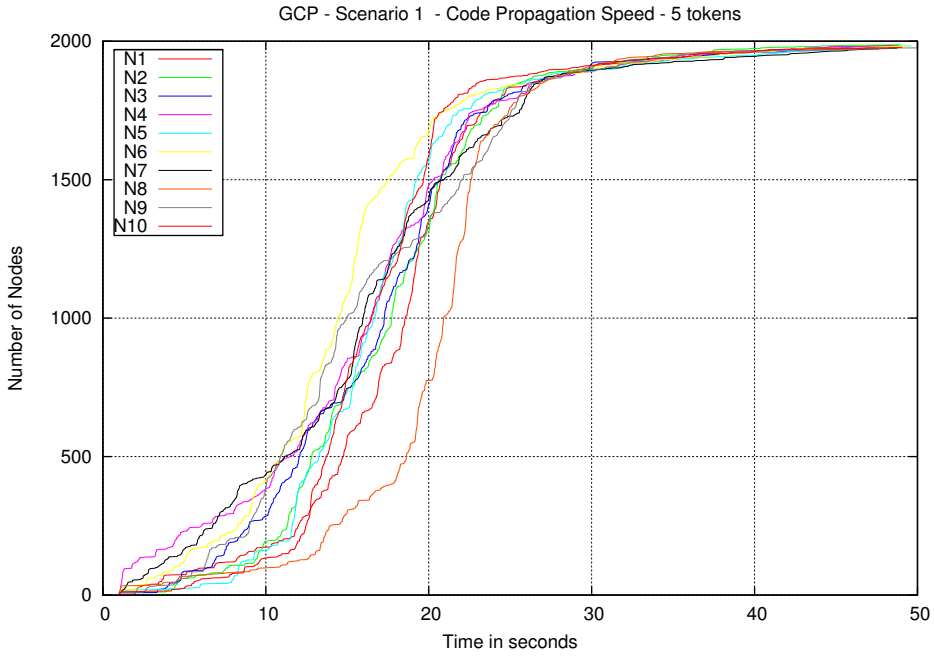


Figure C.5: Scenario 1: 10 runs, 5 tokens.

Scenario 2, 10 runs with 2,3 and 5 tokens

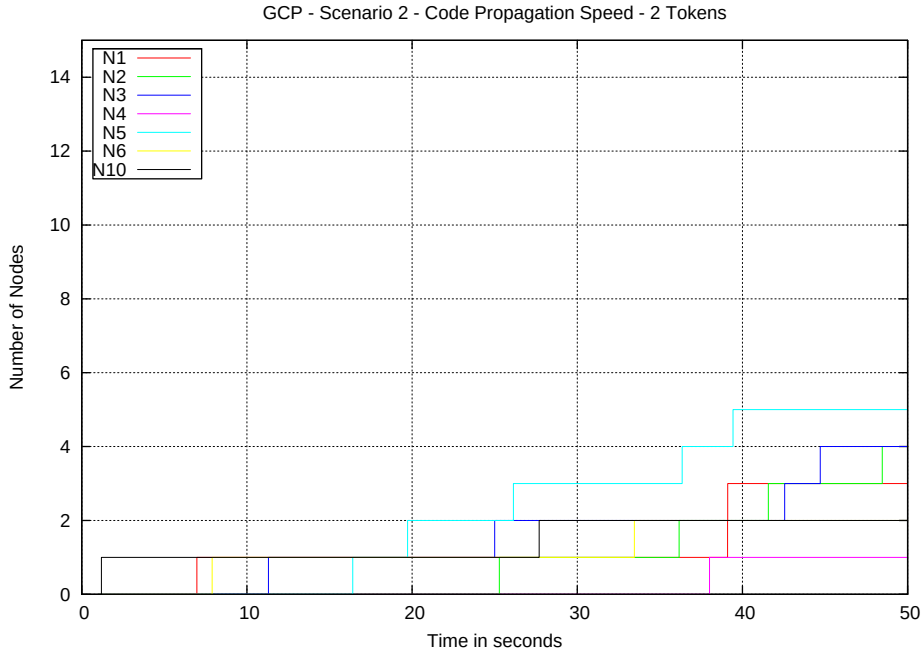


Figure C.6: Scenario 2: 10 runs, 2 tokens.

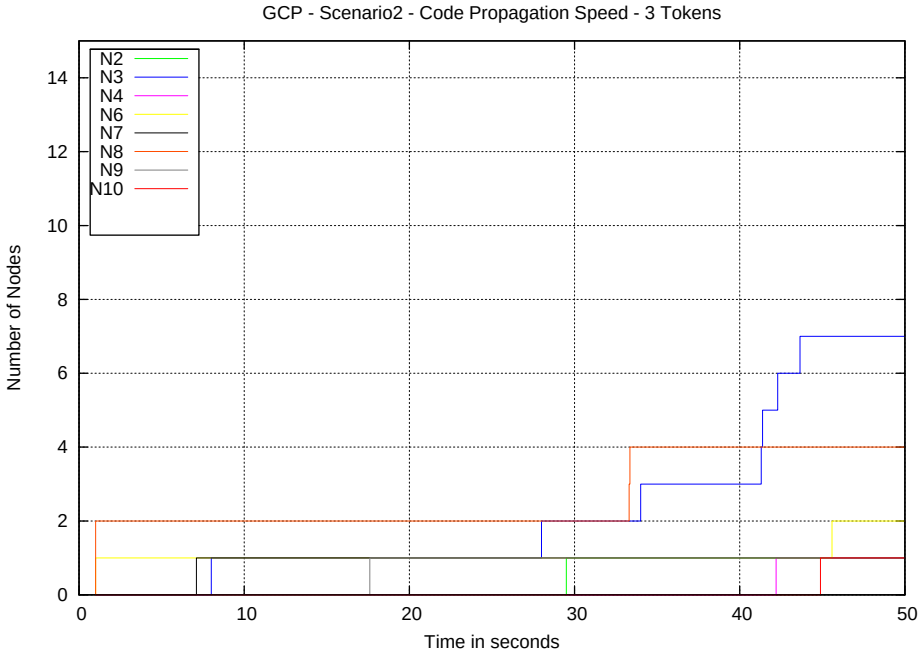


Figure C.7: Scenario 2: 10 runs, 3 tokens.

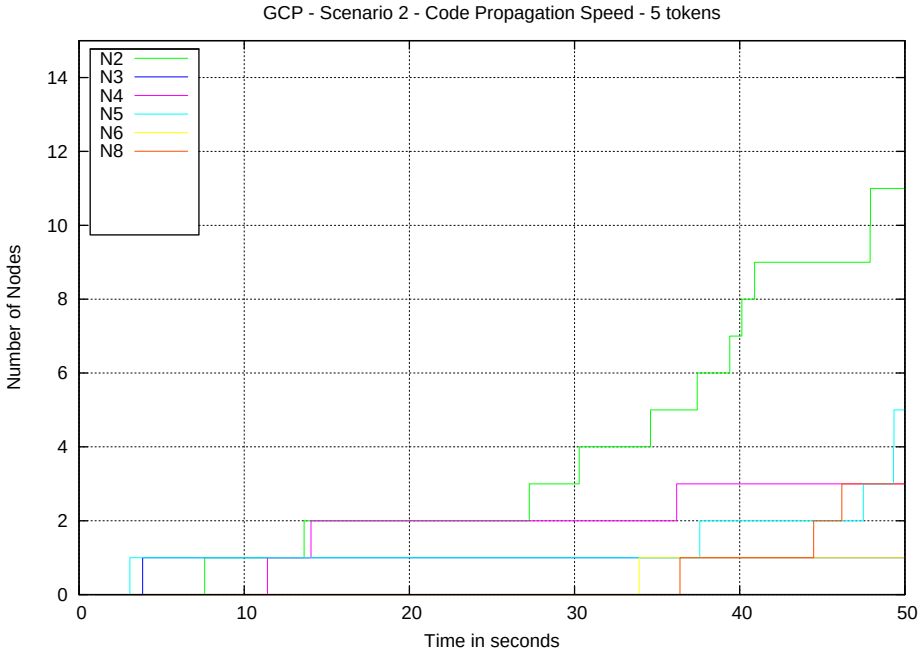


Figure C.8: Scenario 2: 10 runs, 5 tokens.

GCP Replication results

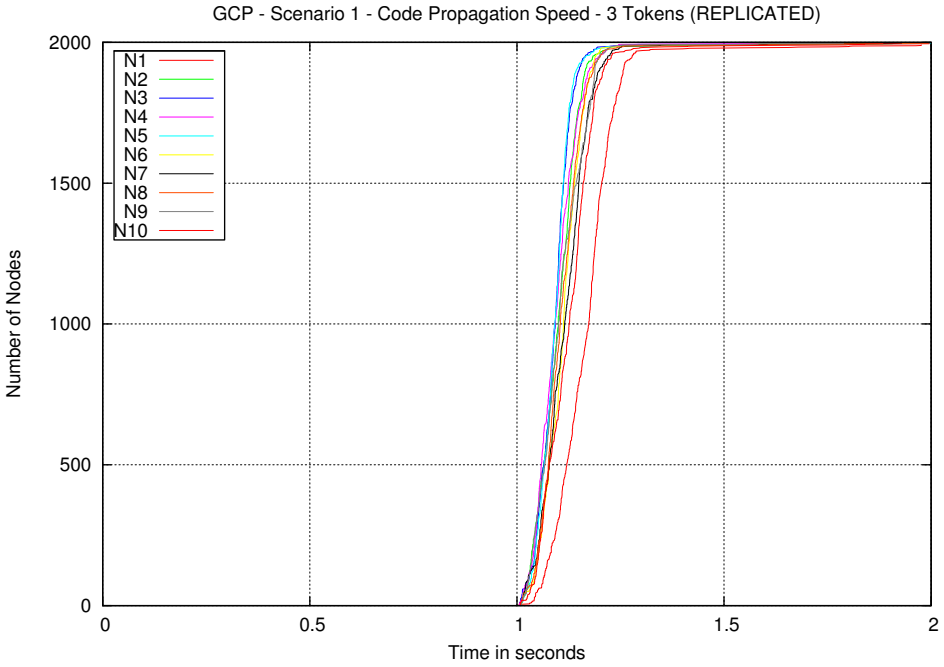


Figure D.1: Replication: scenario 1 - 10 runs - 3 tokens.

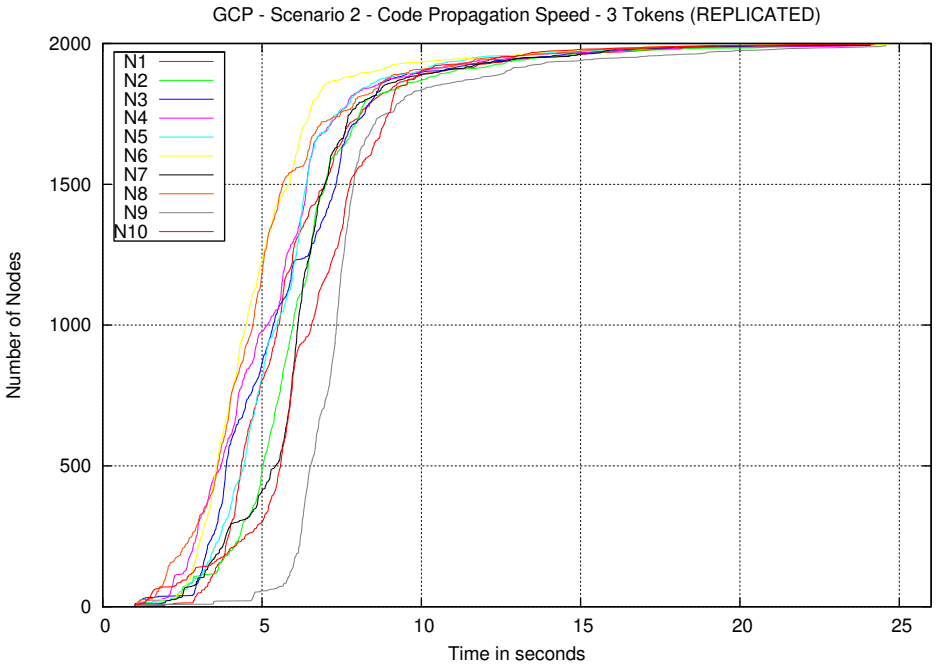


Figure D.2: Replication: scenario 2 - 10 runs - 3 tokens.

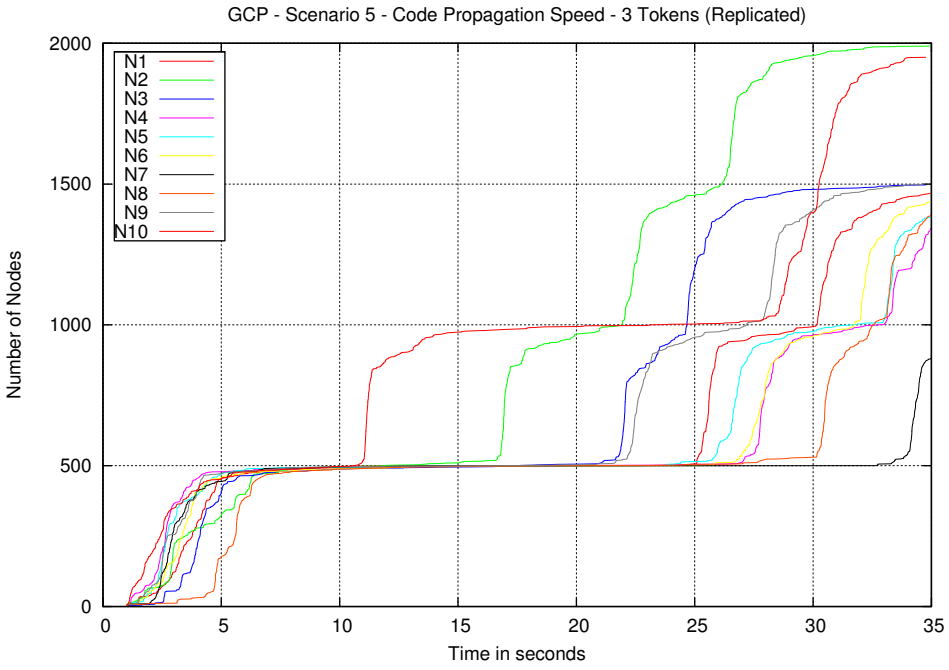


Figure D.3: Replication: scenario 5 - 10 runs - 3 tokens.

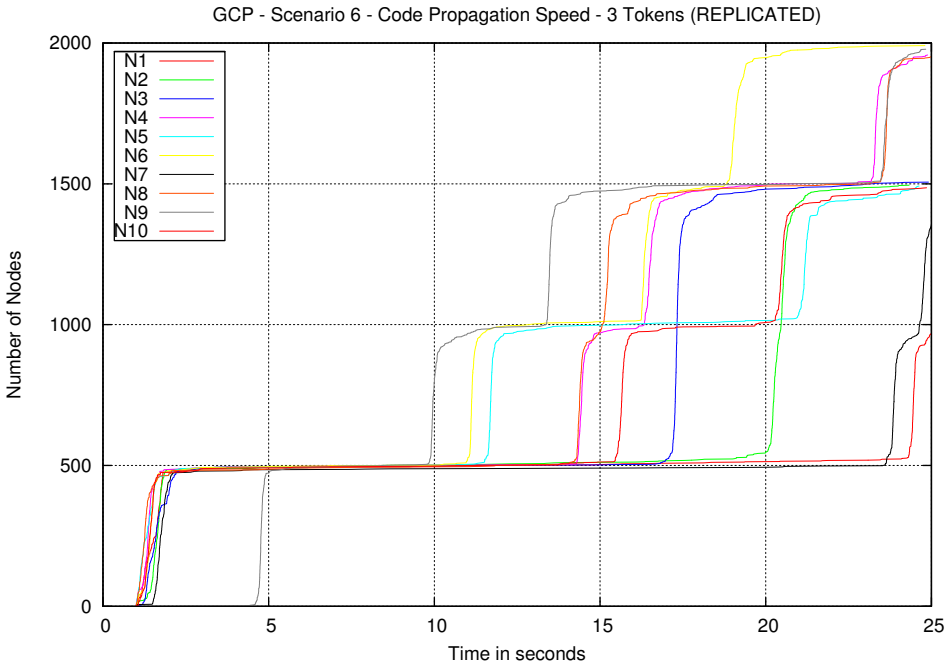


Figure D.4: Replication: scenario 6 - 10 runs - 3 tokens.

eGCP results

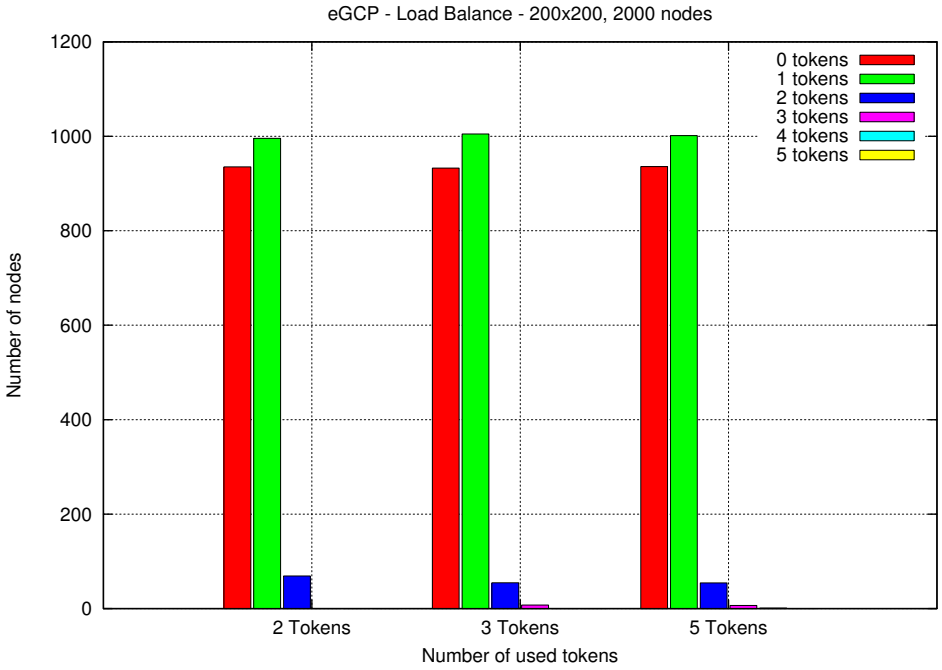


Figure E.1: eGCP: scenario 1d - 10 runs.

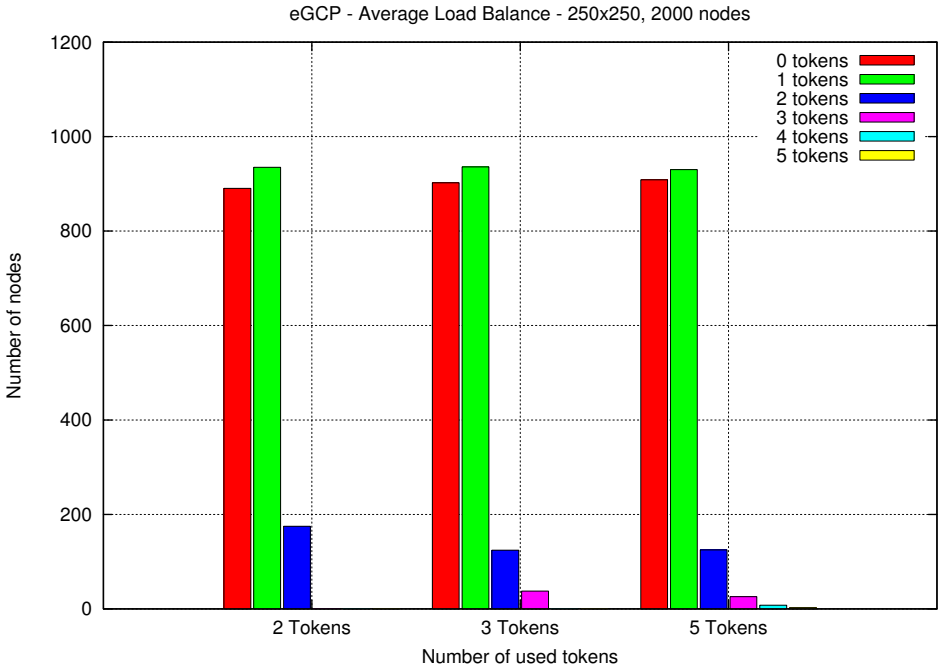


Figure E.2: eGCP: scenario 1 - 10 runs.

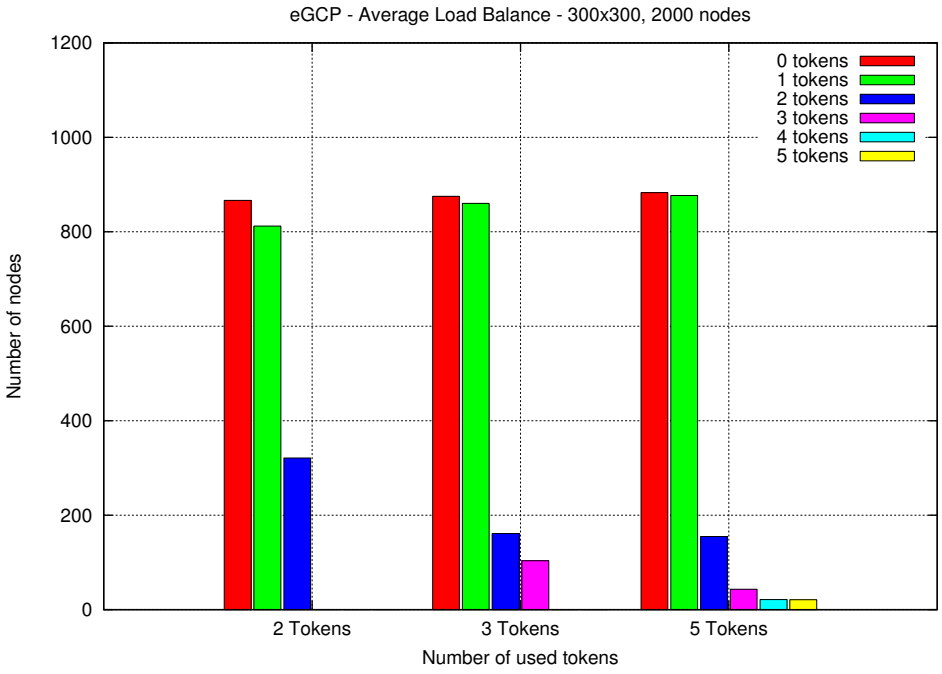


Figure E.3: eGCP: scenario 1s - 10 runs.

Report summary

In wireless sensor network energy consumption is the main problem. When a node is deployed in the field, its battery power denoted its life time thus resource awareness is required. A problem occurs once the nodes are deployed if the system needs an update. Collecting each sensor node by hand would be a tedious task, impractical if not impossible. Therefore, when such sensors need to be updated another approach is required. In this project we focus on *Energy Efficient Code Updates*. When the update is disseminated throughout the network some nodes will use more power relaying the update than others. This can cause key nodes to exhaust their power supplies faster than the rest of the network, and the network will become fragmented.

We want to improve the energy consumption while diffusion updates in a wireless sensor network. When distributing an update, the load balance of the nodes will vary. If the balance is too uneven the network becomes fragmented which in worst case can prevent the running application to fulfil its purpose. A way to solve this problem is to use the ideas of even load balance as described in [9]. The results presented by the authors in [9] look very promising. Going into greater detail of the GCP protocol reveals some flaws though. The specification of the GCP protocol in [9] is insufficient as important details are omitted. For instance, the movement pattern of the nodes is not clearly specified. Another example is the description of the periodically transmission of the beacon message. According to the authors this message should be transmitted after a given time out throughout the lifetime of the sensor node. But a radio can only either transmit or receive at one time. This means that if the radio was in a middle of receiving an update message and the time out happens, the radio apparently has to switch and send the beacon before it continues to receive the update, unless it postpone the message till after the reception has been completed. Whatever happens is unclear since the precedent rules have not been clearly specified. The authors assume that no collision / packet loss happens in their wireless setting, thus they do not need to specify recovery management, a vague assumption in our opinion since we operate with wireless communication. Also, the way tokens are used when sending update messages has not been specified properly. This means that under certain circumstances multiple tokens can be consumed whereas one was enough.

In this report we present a short background regarding the hardware used in wireless sensor networks. This is done to characterise the need for resource awareness. We give an overview of typical hardware platforms such as AT90CAN128 and CRUMB168-USB and present an update management model which can be divided into two parts: the resource constrained sensor network, and a base station.

We describe the GCP protocol in a detail manner. We specify this protocol according to the protocol specification methodology in [12]: we describe what service the protocol provides, in which environment it can be applied, depict the message vocabulary and define the encodings. Lastly we characterize procedure rules and visualize them using flow charts.

We then verify the GCP protocol using the UPPAAL modelling tool in order to ensure the correctness of the protocol. In addition, we examine time and energy properties of the GCP protocol so that we have a knowledge how much time and energy we need to update a node. Additionally, we extend the UPPAAL model with the notion of transmission time, connectivity ,and energy in order to see the best and the worst case code propagation speed and energy consumption with the given network topology.

We implement the GCP protocol in Network Simulator 2. The results are compared with those found in [9]. They did not match so we make estimations of the the best cases for scenario 1 and 2. Lastly, we replicate selected scenarios.

We present an extension to the GCP protocol called enhanced GCP (eGCP). A systematic specification of our protocol will be specified. In eGCP protocol we add a reliability mechanism, so the network can be subject to failures. We also change the way tokens are used in order to improve the load balancing. An implementation of the eGCP protocol will be made and tested to see if the behaviour matches our expectations. Finally, the GCP and eGCP protocol will be compared in the selected test scenarios.

To conclude we discovered that the use of tokens can improve the load balancing if the network is dense.