Universiteit Leiden

Computer Science

Smart Grid: Combining RF mesh grid and public carrier networks for last-mile communications

Name: B.J.C. van der Drift
Student-no: s0522929

Date: 16/06/2011

Supervisor: Todor Stefanov

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Smart Grid

Combining RF mesh grid and public carrier networks for last-mile communications

Universiteit Leiden

Westland energy solutions

B.J.C. van der Drift

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University

Westland Infra

Table of contents

1. Introduction and background knowledge

1.1 Introduction to smart grid

Power grids have been run the same way for decades all around the world. Since the Alternating Current camp won the *War of Currents* in the late 1880's and all countries around the world adopted the resulting electricity system, not much has changed about the way electricity is delivered to the customers. Although technology throughout the grid has been added and updated, many utilities still don't know about outages until they receive a phone call from one of their customers. This leads to a lower reliability of the power grid. An unreliable grid damages the economy, as outages are extremely costly for many businesses. A healthy electricity network is crucial for a healthy economy.

Other problems that have emerged are related to environmental issues such as global warming and carbon emissions. These problems are a threat to mankind and need to be dealt with. Governments and institutes around the world have set goals to counter these issues, but a lot of work is still to be done to reach those goals. Another related problem is the depletion of fossil fuels. Because of the enormous amount of fossil fuels we are using, we have been slowly depleting the earth's natural resources, and one day, this supply will come to an end. In order to cope with this problem, we need to start using fossil fuels in a more responsible and conservative way. We also need to be looking for alternative energy sources, such as solar and wind power, to supply us with renewable energy that has no impact on the environment and cannot be depleted. However, the use of these technologies poses some challenges, as these sources are uncontrollable, variable and unpredictable.

By improving and optimizing the power grid, we can help countering these problems, and create a *smart gird*. Smart grid is a term that covers many improvements and additions to the power grid, which helps improving it by making it more reliable, energy and cost efficient, environmentally friendly, monitorable and controllable. It encompasses technologies such as smart metering, distributed generation, outage detection and peak load reduction. Using these technologies, we can transform the traditional grid and create a smart grid that is ready to cope with the challenges that the future will bring.

One interesting and diverse problem, is how to communicate with devices at the end of the electricity chain: the actual customers. For smart metering, utilities' data networks need to reach all the way to the homes of their subscribers. Some experts even foresee utilities will be able to control customers' appliances such as air conditioners and heating systems, in order to spread the load on the grid, and thus reducing the peak load. In this paper, we focus on these *last-mile* communications to a great extent.

1.2 Smart grid approach

Smart grid is a concept that has no clear definition. Around the world, many governments, utilities, regulatory bodies and businesses are involved in smart grid development. Each of these stakeholders have their own approach to the smart grid concept in correspondence with their policies and business models. Opinions differ about a variety of matters, such as what the main focus of smart grid should be, which technologies are included in smart grid and which are not, and how the smart grid should be constructed. In previous work, we have gone into these difference of smart grid visions around the world in great detail. [1]

In this paper, our intent is to research smart grid communications from a general perspective, holding value for a global public. Especially the technical realization of smart grid communications is important for many smart grid stakeholders in many parts of the world. It is also a core aspect, independent of higher-level, more conceptual aspects of the grid.

Although written from an independent, scientific perspective, it should be taken into account that our research is performed in collaboration with Dutch utility Westland Infra. As a result, their vision and interests are somewhat reflected in our research, meaning that we might go into detail at points that are interesting to them, and discuss matters more superficially where this interest does not exist. Additionally, some assumptions, restrictions, advantages and disadvantages might not hold for other parties due to differences in regulations, funding or environmental properties, for example.

Westland Infra is the utility of the Westland area, which is characterized by its greenhouses. These enterprises typically consume a lot of power, which makes energy efficiency rewarding for them. It also opens opportunities for the utility to optimize the electricity network. As a result, Westland Infra has been using smart grid technologies since before the term was coined. Asset management, demand response and smart metering have steadily advanced over the years, and Westland Infra maintains a Wide Area Network that spreads throughout their service area, and provides a robust network which connects many points in the grid.

This situation explains our focus on last-mile communications, since this is the only link that misses between the utility and residential customers and businesses that are not covered by the network yet, or for which the connection could be improved.

1.3 Introduction to smart metering

One of the smart grid technologies that gets a lot of attention is smart metering. Smart metering is the automated reading of electricity metering data without the physical intervention of a person near the electricity meter.

The benefits of smart metering are numerous. First of all, there is no need for an employee of the utility to physically interact with the electricity meter, which results in a cost reduction in the long run. It's also convenient for the occupants of homes where meters are located, because there is no need to stay home to let the utility employee in.

Another advantage is that meter readings can be performed more frequently, and also billing can be done monthly, instead of yearly. This gives consumers a better insight into their electricity use, and can stimulate them to conserve more energy because they are more frequently confronted with the amount of energy they use. This can even be taken further. Meter readings could be performed daily, hourly or even every 5 minutes. This data can then be presented to the consumer through an internet portal or on an in-home display. That way, consumers can see how much energy they are consuming at any time. If they are trying to save energy, they can also immediately see if it has effect, which can be very encouraging.

Smart metering also allows utilities to see when energy was consumed by the customers. This can be used to vary the pricing of electricity based on the time of the day, such that electricity is more costly during peak hours and less expensive during off-peak hours. We call this *dynamic pricing*. This stimulates consumers to use energy demanding appliances during off-peak hours, and not during peak-hours. As a result, less energy will be consumed during peak hours, resulting in *peak reduction*. When peaks are lower, this means that the equipment throughout the grid also require less capacity, which leads to cost reductions. Lower peaks also lead to less strain on the electricity network, which expands its lifespan. [2]

Smart metering also enables a better *demand response*. Demand response is reacting to changes in electricity use to stabilize the power signal and be able to supply the capacity that is demanded. When metering data can be retrieved in near real time, this gives information about how much energy is being consumed in the grid, and the supply can be adapted to exactly match that demand.

Smart metering also enables a reliable method of *outage detection*. When a building is not receiving any power, smart meters will immediately be aware of that and can report to the utility. This gives the utility an immediate indication where outages are occurring and which buildings are affected by the outage. They can use this data to determine where the outage is caused and quickly respond to fix the malfunction.

Lastly, smart metering is an important technology to stimulate the use of *Distributed Energy Generation* (DEG). DEG is the generation of energy at the place where the energy is used. The energy can be generated by solar cells or wind turbines, but DEG by fossil fuels is also more energy efficient than traditional generation and distribution. [3] Renewable energy can be used for DEG in a domestic situation as well, especially solar power. However, because of the variability of solar power, at certain times more power is generated that necessary, while at other times not enough energy can be generated by the solar cells. Although storing energy in batteries can be a partial solution, a lot of energy can still be lost, reducing the financial attractiveness of solar panels. Traditional power meters do not have the capability to register energy flowing back into the grid. Smart meters would allow residents to sell redundant energy back to the utility, which makes investing in solar panels a lot more attractive.

1.4 Introduction to streetlight control

Streetlights are important instruments to keep traffic safe and give people a feeling of safety in their neighbourhoods. They have been used since the old Roman and Greek civilizations, and cities without streetlights have become unimaginable. However, these lights do require a lot of power to burn every night, and they can also cause light pollution in certain areas, and the amount of light that needs to be generated by streetlights is often only required during rush hours.

As a solution to these problems, streetlights can be altered so that they vary their light intensity over time. This allows them to run on full capacity during rush hours, and save energy and reduce light pollution in the middle of the night. It is possible to equip streetlights with light sensors or timing mechanisms to determine how much illumination is needed at a certain time. However, due to the changing lengths of night and day over the year, the times and intensities may also change throughout the year. There can also be certain situations and events for which different requirements apply, such as festivals or calamities, which require more control over the streetlights' light intensities.

In order to realize this level of control, reliable communications with each lamppost will be required. But streetlights can be located throughout a large area and in different environments. Length of the lampposts, the distance between them and obstacles can all have an influence on the communications that are best suited to control the streetlights.

1.5 Thesis organization

After the introduction to smart grid and some of its technologies in this chapter, we will continue to formulate some problems that are encountered when realizing smart grid communications in chapter 2. In this chapter, we also examine related research that has been done on the subjects we discuss in this paper.

In chapter 3, we will discuss the system we developed to solve the problems introduced in chapter 2. We give a description of all the different components of the network and its topology.

Then we will describe a case study we performed to examine the capabilities of our network in chapter 4. We will explain the experiments we have performed, present and analyse the results, and then conclude what these results imply.

In chapter 5, we will conclude how well suited our system is for smart grid communications, reflecting back to the requirements we set in the beginning.

We will then continue with discussing possible future research that could be performed to expand upon our research in chapter 6.

After this, we include two appendices. Appendix A contains the script which we used to generate the graphs we use to present the results of our case study, as well as the raw result data files that are used to generate the graphs.

Appendix B contains all the source code of the program we developed to perform the experiments. This is a C++ program and consists of several header and implementation files.

2. Problem definition and available solutions

2.1 Problem definition

The European Union has set goals to counter environmental issues through smart grid technology. One of these goals is to equip 80% of all European households with smart meters. [4] This forces utilities to start rolling out smart meters in large numbers. However, retrieving metering information from these meters is a complicated process which has no clear best practice. One can think of many ways that theoretically enable communications with smart meters, from cable internet to wireless GSM communications. However, all these approaches have their advantages and disadvantages, and there isn't much public knowledge about the best way to realize smart metering communications.

Many utilities have rolled out smart meters with different communication technologies, but information about their costs and success are very hard to find, since these companies usually do not wish to share any of their knowledge with others. This became very clear when we contacted some utilities around the world that are known to be smart grid innovators and tried to obtain some information about the way they constructed their smart metering communications infrastructure. None of them provided us with any useful information whatsoever. However, we learned through informal channels about utilities that had rolled out smart grid technology, but were having problems with reliability and regulations that did not allow them to get maximum results. Official statements also make it clear that realizing a reliable smart grid network can lead to many problems and high costs. [5][6]

Another problem is the combination of various smart grid technologies over the same network. Most existing knowledge about smart grid communications deal with only one type of application communicating over the network, which makes it possible to create a network around the specific needs of that particular application. However, it is problematic to deploy a new network for every single smart grid application. If we are rolling out smart meters and a streetlight control system, and supply them both with a separate network, we will have double work and double costs. We should also expect more smart grid applications to be added in the future. It is not possible to supply every single smart grid technology with its own network, since this would create an enormous amount of management and maintenance. For this reason, a standardized and transparent approach must be taken. We must find a solution that allows different applications to use a similar infrastructure and use the available technologies in an appropriate way.

These are the major requirements that a smart grid communications network needs to fulfil:

- **Reliability**: Failure to communicate is devastating for smart grid applications, since the entire smart grid concept is based on reaction to changes elsewhere in the grid. Without successful communication between different parts of the grid, smart grid applications will fail to do what they are designed for. For some applications, such as demand response, forecasting models can be used to substitute live data and react according to these predictions in case of a failure. However, the advantage of smart grid over traditional operation is lost in these situations.
- **Latency and datarate**: Multiple smart grid applications can together generate a lot of data that is to be sent elsewhere in the grid. In order to allow swift delivery of this information, the communications infrastructure needs to provide enough bandwidth and throughput for these communications. Since smart grid is all about reaction to changes elsewhere in the grid, a low latency can also be an important requirement. In the smart grid design, future expansions and additions of new smart grid functionality should also be estimated and taken into account.
- **Security**: With new applications sending data all over the grid, cyber security is an important factor in smart grid designs. With smart metering, personal information about customers' energy use is being sent over the network, which has given cause to privacy concerns. Groups or individuals with malicious intent also pose a threat to the grid, as hackers could take down (part of) the grid if security not be tight.

- **Adaptability and upgradability**: No matter how good a network you've created, circumstances can always change. New applications may be added to the grid, resulting in higher or different requirements, such as a larger bandwidth or better range. Once in place, it is important that changes can be made to the communications infrastructure to keep it up to date and well-fitted for the grid. Utilities are not necessarily responsible for this themselves. Utilizing a public carrier service will most likely guarantee new upgrades and an up-to-date network. Private networks require utilities to do this themselves, but do give them control over the specific upgrades that are done.
- **Flexibility**: Circumstances can differ greatly within the service area of a utility. Some areas might be densely populated, whereas other areas are more rural in nature. High-rise blocks can require different communications than regular houses. A certain degree of flexibility is required to be able to provide communications in all these circumstances. The use of multiple technologies that can be combined and switched out is probably required.

Some utilities already maintain a large Wide Area Network (WAN), which spreads throughout their service area to connect stations and other facilities to each other. [7] This is mostly implemented through copper wire and optic fibre cables, which provide reliable communications and a high datarate. In this case, the challenge is to connect the communications infrastructure all the way to the end-point smart grid devices, such as smart meters, which are located at homes and businesses. In the next section we discuss various options to realize these last-mile smart grid communications.

2.2 Available technologies

Many existing technologies could be used for smart grid communications. All of these have some advantages and disadvantages. In this section, we will discuss the most appealing technologies and why none of them is the holy grail which can individually provide us with the communications requirements we need.

2.2.1 Power Line Carrier

Power Line Carrier (PLC) is a technique that many utilities are already familiar with. PLC is being used by utilities for a variety of purposes such as switching public lighting on and off. Utilities own the power cables, and they lead to all homes and businesses, as well as to substations and other control buildings where smart grid technologies could be deployed. This theoretically makes PLC an ideal candidate for smart grid communications. Many utilities already use PLC as part of their smart metering infrastructure. [8][9]

However, the reliability of PLC communications is not without question. Especially interference issues give rise to doubts among utilities about the feasibility of reliable PLC communications. [10][11][12][13] Another large disadvantage of PLC is the fact that communication fails during outages and electrically abnormal events, and it's exactly those situations in which the smart grid could provide a means to quickly diagnose and solve the problem. [14]

2.2.2 Cable internet

Internet Protocol based technologies have proved to enable a reliable network that can handle a high datarate. Extending a utility's existing WAN and directly connecting smart meters to this network would be ideal. However, placing cables leading to all residents and businesses can be very expensive. Even though the costs of cables have reduced over the years, they would still have to be placed, leading to every meter in the grid. [15][16] Also, the provided bandwidth would be overkill for simple communications such as smart metering. For these reasons, this option is not relevant to most utilities. Some utilities have laid optic fibre cables to homes, and justify the costs by also offering data, video and voice communications over these lines. [17][18] However, not all utilities are in a position to do so, mostly because of the separation of internet and electricity providers in many countries.

In this case, another option is to use the internet, to which virtually all businesses and residences have a connection already. Equipping smart grid devices with internet modems and connecting them to the internet would allow broadband communications without large infrastructural investments. However, sending smart grid information over the internet would pose some security threats, which could raise concerns from users who fear for their privacy. Internet subscription costs could also be considerable, and more than utilities want to spend on smart grid communications. Furthermore, the internet service provider as an additional stakeholder would make the situation more complex, and requires negotiation about who pays which portion of the connection costs, which can be quite hard to determine.

2.2.3 Wireless internet

As an alternative to cable internet, utilities can opt for a wireless internet solution. Since cables do not need to be laid, this saves a lot of initial costs, and can speed up the roll-out of smart meters and other equipment. The main choice for utilities when going for wireless communications, is between deploying a self-owned and maintained wireless network, or using the services of a public carrier. [19]

Many reasons exist why utilities would choose public carriers over a private network, such as good coverage and bandwidth, and no added responsibility to maintain the network. Carrier prices for data connections have dropped recent years, which makes this option more and more appealing. [20][21]

The reasons to choose a private network are plenty as well. Deploying a self-maintained network places all control in the hands of the utility. It can decide what type of network to use, where coverage needs to be realized, and when to upgrade their network, whereas a utility using a public carrier network heavily depends on the carrier for maintenance and upgrades. Another reason is that public carriers may provide far more bandwidth than needed for certain technologies such as smart metering, depending on what data the utility wants to receive, or how often to receive it. The maintenance costs for private networks might also be lower than the subscription costs to public networks. [21][22]

Another disadvantage of public networks, is the limitation of the amount of connections per base station. As a result, the combined datarate of all the connections will be much less than the sum of the achievable datarates by the single connections. To solve this problem, communications could be scheduled by turn, limiting the amount of connections at each moment. This means additional functionality is required to determine which connection should be switched. It also limits the ability to quickly react to important, unpredicted events, since an end-node would have to wait for its turn to be able to send urgent information.

There is a great variety of wireless internet solutions that can be chosen from. Choosing a public carrier to supply the service will most likely lead to GPRS or UMTS communications. When a large bandwidth is required, WiMAX is a 4G technology that could provide this. When thinking about private wireless internet, Wi-Fi is the first thing that comes to mind. However, this technology is mostly restricted to local networks. If a larger range is required, this leads us to wireless mesh grids.

2.2.4 Radio frequency mesh grid

Radio frequency (RF) mesh grids can provide a flexible, reliable and secure network that can span a considerably large area. This is achieved by *hopping*, which means that a message is passed down by multiple nodes until it reaches its destination. Once in place, operation is relatively cheap. The network is self-forming and self-healing, so if one node stops functioning, the rest of the network is not damaged. RF mesh technology can operate at a number of different frequencies, which results in a variable reach and datarate. This means that utilities can choose which frequency they use according to their needs.

In North-America, RF mesh is the most popular technology for smart metering. [23][24] In Europe, these networks have not yet emerged as much. This may be due to a variety of reasons. One important factor is the difference in regulations concerning unlicensed radio communications, which are more flexible in North-America than in Europe. Utilities that use radio communications above a certain power level need to apply for a license, usually for a different frequency, which brings extra costs to the project. [25] Available frequencies and costs can differ between countries.

Although large distances can be covered by hopping, this does require a certain density of nodes. When the distance between neighbouring nodes becomes too large, mesh grids will fail to deliver a robust and reliable network. There also is a limit to the amount of nodes that can exist in one RF mesh network. More hopping also means a larger reaction time.

2.3 Related research

Many papers have been written about smart grid and smart metering communications. Many discuss the requirements of a smart grid, and give recommendations about how this can be realized. This includes papers from the Electricity Advisory Committee, the U.S. Department of Energy and the European SmartGrids Technology Platform. [26][27][28] These papers explain the necessity of the smart grid, and discuss what it should be capable of doing, and what this would mean for the future.

These papers are supplemented by white papers from utilities, commercial enterprises and field experts, who project their vision of smart grid and discuss specific aspects of the smart grid [29], give a step-by-step plan to move towards a smart grid [30] or summarize recent smart grid pilot projects. [31] In other words, the amount of papers on smart grid and related matters is enormous, ranging from the fields of engineering and computer science to economics and management.

Last-mile communication is often shortly mentioned in papers, but few studies are fully dedicated to this subject. A. Clark and C.J. Pavlovski research mobile network services, utility operated WiMAX and mesh networks for last-mile communications. [32] Their main conclusion is that there is a "need to select several technologies and suppliers to ensure the 100 per cent coverage required by the electricity supplier", and "it is anticipated that a combination of several wireless solutions will provide breadth of coverage in an economical way". They do not consider wireless mesh networks a practical option, however. Their main reasons are:

- "It was not clear how meshing may evolve longer term"
- "Traffic from other mesh cells is routed through another household and the full implications of this may not be clearly understood from a security, privacy, and customer perspective"
- "A lack of proven ability to scale and offer telecommunications industry standards"
- "The electricity industry, regardless of who operates the network, is better positioned when deploying standard telecommunications technologies as opposed to introducing its own solutions"
- "By accepting market trends for telecommunications, longer term certainty is enhanced"

It should be noted though, that these arguments are not technological in nature, and only provide extra motivation to research mesh networking capabilities.

Deconinck gives an overview of smart metering possibilities for Flanders, Belgium. [13] Different technologies are compared, and the author concludes that "for future-proof smart meter applications, it is clear that no single communication means on its own will be able to meet all requirements", and "When it is necessary that a set of meters is reached within a given time span as for smart metering control applications (real time requirements), then the medium shall support a form of broadcasting or parallelism. This favours solutions such as PLC or RF, or other wireless solutions (PMR, UMTS)."

In a ZigBee for smart metering study, K. Liu concludes that "ZigBee fulfills the basic requirements of electricity meter applications". [33] "ZigBee has a fairly large transmission range within line of sight both indoor and outdoor", "it has an effective data rate" and "supports a large network size and provides 8 levels of security to ensure the integrity and confidentiality of metering data".

2.4 Research contributions

As we have shown in the previous section, multiple sources agree that a combination of technologies is probably the best solution for smart metering and other smart grid applications. However, studies that actually experiment with combinations of different technologies and applications are scarce, if existent at all.

This is what motivated us to our thesis, and we provide the following research contributions:

- Realization of last-mile smart grid communications using multiple technologies
- Realization of last-mile smart grid communications involving multiple applications
- Description of the development process to provide a transparent communications infrastructure
- Implementation of ZigBee in smart grid networks
- Examination of the reliability and capacity of the communications

3. Smart grid last-mile communications

3.1 Introduction to the river system model

In terms of dataflow, a smart grid can be constructed following the model of a river system. Small ‘streams’ of data emerge at the beginning of the system: smart meters, smart streetlights and other end-devices. These data streams can originate from a wide area, but do not usually account for a high datarate individually. These streams can be combined together, streaming in the direction of the control centre of the utility. They form a bigger stream, one which requires a higher datarate, combining the datarates of the streams before it and the data that is generated on that location. These streams can then again be combined, and so on, slowly forming the ‘river’ of data which will arrive at the control centre.

At any point, the communications should meet a combination of the requirements for everything further down the network, not only in terms of bandwidth, but also reliability and stability. A failure in the design at one point in the grid affects the communications from that point on, all the way upstream to the end-points. We can imagine that this model shows many similarities to the network of electricity cables itself, which also starts with a lot of power flowing through large cables, and slowly branching off into smaller cables with a lower capacity.

There is an exception to this model when communication is required between multiple parts of the grid, but not to the control centre. However, in this case, there is a sub-model in place following the same principles as discussed above. Also, communication to the control centre is still desirable, because a smart grid demands monitoring and control of all processes. It is also possible that data streams are processed at one point, such that the amount of data flowing onwards to the control centre is smaller than the combination of data that arrive at that point. However, the model is still accurate for the other requirements besides datarate.

It is also possible to deploy separate networks. For example, a single utility might have a network comprised of an optic fibre WAN, leading to copper wire cables and later into a WiMAX network. A second network could run via the internet, cellular services and a wireless mesh network.

Another thing to keep in mind is the fact that data does not necessarily flow in one direction like in a river system, but also in the opposite direction, towards the end-points. For some technologies, the properties of the up- and downstream and not identical, and this should be taken into account at design-time.

Taking our river system model in mind, this also leads us to a way to analyze whether an existing network is ready for new applications. We just follow the path from the place where the new system will be placed, back to the control centre, and check whether the network can handle requirements of the new application. For example, when planning implementation of advanced metering infrastructure (AMI), the entire network should be ready to handle the data from homes to the utility. If this is not the case, deploying such a system will probably not be a good idea.

Utilities should also keep in mind whether other smart grid upgrades will later be made as well. Maybe the network can handle AMI now, but congestion and bottlenecks may arise when other applications are added. For this and other reasons, it seems a good strategy to start with smart grid advancements closer to the utility in the network, instead of starting with AMI. [34][35]

Last-mile communications are analogous with the starting data streams of the river system model. Although the term can be interpreted with some flexibility, the last mile minimally includes the path from the end-point to the first node in the network at which data is bundled, separated, extracted or processed. However, when this first link is remarkably short in distance, which can be the case in RF mesh networks, paths further down the network can also be considered last-mile.

3.2 Used technologies and motivation

Smart grid applications require reliable communications, and also a certain form of flexibility. Some applications require more bandwidth than others, and their environments can change from location to location. For these reasons, we developed a communications framework for smart grid applications with the use of two communication technologies: ZigBee and UMTS. These two technologies are well suited to supplement each other in a network, because many of the disadvantages of one of these can be countered due to the characteristics of the other. We will describe the limitations of these technologies, and explain why the use of the other, or a combination of both counters these problems.

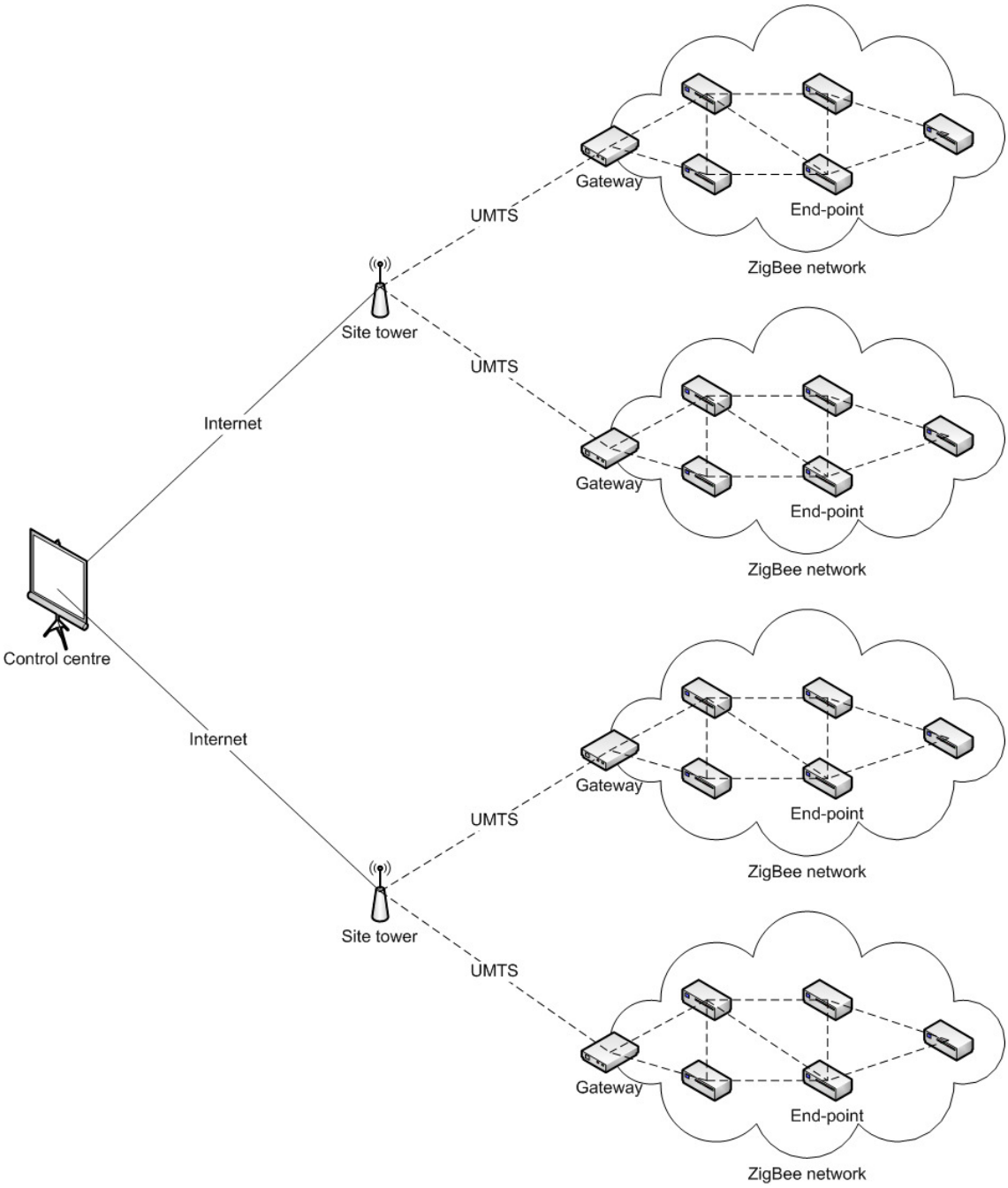
Problem: ZigBee has a limited range. Although ZigBee is suited for densely populated areas, problems may arise in rural areas.
Solution: Large coverage by the mobile network in one of the main merits of UMTS. In many places, almost any home can be reached by the mobile network.

Problem: When networks become too large, messages may take very long to travel to the control centre, while the smart grid requires near real-time communications.
Solution: A ZigBee network can easily be split up into multiple networks. These separate networks can then be individually managed over a UMTS connection. This allows swift delivery of information to the control centre.

Problem: Although subscription costs have dropped over recent years, communicating with each and every electricity meter and streetlight still brings a considerable amount of costs each month.
Solution: By splitting a UMTS connection over multiple smart grid end-points through a ZigBee network, subscription costs drop considerably. Only one public carrier subscription is needed to reach all end-points in the ZigBee network.

Problem: Although 3G technology allows more simultaneous connections than before, there still is a limit to the amount of devices that can connect via the same site tower. If all smart grid end-points have an individual UMTS connection, this requires some turn-taking regulation among these devices. This brings extra development costs and communication requirements.
Solution: By using ZigBee networks to split up UMTS connections over multiple smart grid devices, the amount of necessary simultaneous connections to site towers is greatly reduced. This allows any end-point device to be reached at any point in time.

Naturally, some additional technology is required to be able to route data over the public carrier network to a certain location, and vice versa. In order to realize this, we developed a gateway that allows transparent communication from the control centre, over the public carrier network via UMTS, up to a smart grid end-point device over the ZigBee network. Below is a simplified diagram that shows how the network is structured.


Control centre
Internet
Internet
Site tower
Site tower
UMTS
UMTS
UMTS
UMTS
Gateway
End-point
ZigBee network
Gateway
End-point
ZigBee network
Gateway
End-point
ZigBee network
Gateway
End-point
ZigBee network

3.3 Gateway description

In order to route data from smart grid end-points to the control centre over a ZigBee network, and then over a UMTS network, we need a gateway to send messages from the UMTS connection over the ZigBee network, and vice versa. In order to do this, we combined a UMTS modem and a microcontroller with a ZigBee module. The UMTS modem forwards the data it receives over UMTS to the microcontroller via an Ethernet cable.

We can now communicate with the microcontroller by sending a message to the IP-address of the UMTS modem. The modem automatically forwards any data it receives to the microcontroller.

In order to control the gateway, we developed a protocol in which commands can be given to the microcontroller. These include commands which control the ZigBee module of the microcontroller. This module is the coordinator of its corresponding ZigBee network, which means that it regulates network configuration, and end-point devices log into this coordinator when they join the network. The gateway protocol allows us to modify certain ZigBee network behaviour, such as changing encryption or discovering new ZigBee nodes that might have appeared in the network.

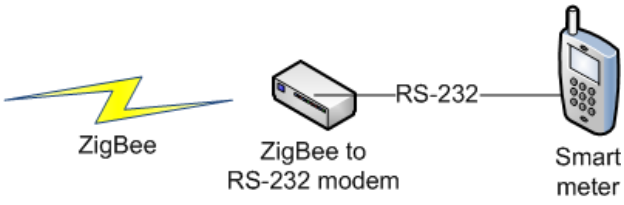
In order to communicate with an end-device in the network, we wrap the data we wish to send in a command to the gateway which specifies the ZigBee address of the end-point. The gateway then extracts the data, and wraps it in a ZigBee packet, which it then sends to the addressed ZigBee node.

Data which is sent from an end-point to the control centre works in a symmetrical manner. The gateway receives a message from an end-device, extracts the data and wraps it in a gateway protocol message, in which it also specifies the address of the ZigBee node from which the message originates.

Multiple ZigBee networks exist when multiple gateways are placed in an area. Which network is joined by a specific end-device is automatically determined. The ZigBee node determines to which coordinator it has the best connection, and joins that network. The gateway keeps track of which nodes have joined its network, and this list is periodically retrieved from the control centre. Using this data, we can determine to which gateway we need to connect when we want to communicate with an end-device. This also allows end-points to join a different network when a gateway malfunctions, as long as another gateway is in reach. This means that all end-devices can remain reachable when a gateway gets destroyed or is unreachable for any other reason.

3.4 Smart metering solution

Many different types of smart meters exist today, incorporating different types of protocols and communication technologies. However, we were not able to find any smart meters working with ZigBee, so we had to find another solution to communicate with smart meters through our ZigBee network. To realize this, we used a modem that extracts data it receives over ZigBee, and transparently sends it over a RS-232 port. Since most smart meters incorporate RS-232 or similar input, this is a very reusable method to enable ZigBee connectivity on smart meters.

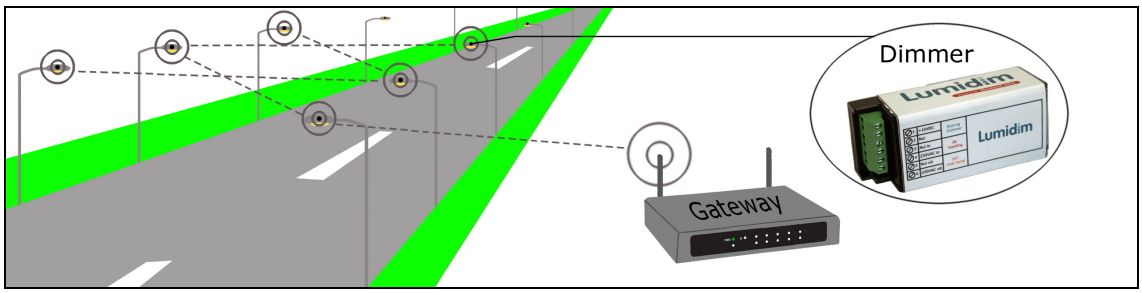


Since the data is transferred transparently to the smart meter, this gives us full control over the smart meter from the control centre. Smart meters use a variety of protocols, and we can simply wrap any message we want to send to the meter in the gateway protocol and address it to the ZigBee module of the RS-232 modem that is connected to the meter. This means that when new meters are installed, or a meter is replaced by one using a different protocol, no other hardware needs to be switched or modified. The control centre simply decides which protocol to use to communicate with the smart meter, and the data gets transparently passed along by the different devices in the network.

3.5 Streetlight control solution

There are several different ways and available technologies to remotely control public lighting. However, this is still a quite new development, and we did not find any solution suitable to our needs. For this reason, we developed our own system using the technologies we have discussed earlier.

A custom module was designed to be placed in every streetlight. These modules contain a ZigBee module to communicate with the gateway. A schematic representation of the system is shown below.



The module's main function is controlling the light intensity of the streetlight. The board is controlled by a PIC processor, which is a robust, low-cost microprocessor. The PIC processor manages all the activities that happen inside the control module. It also contains an EEPROM chip, where a diversity of information is stored.

3.5.1 IDE and programming

We programmed the PIC processor using the microC PRO for PIC software, developed by MikroElektronika. [36] This IDE allowed us to program the module entirely to suit our needs in the C programming language. We utilized two ways to load the binaries into the PIC processor. The first method is using a PICFlash2 to load the program directly from microC to the PIC via the USB port of the development PC. [37] As an alternative, we used the Kanda handheld PIC programmer. [38] This allows us to program streetlight modules on location, without the need to bring a laptop or other large device.

MikroElektronika PICFlash2

Kanda handheld PIC programmer

3.5.2 Basic operation

The module gets powered when the public lighting is switched on. At that moment, the light intensity starts at 100%, and remains that way for a period of 5 minutes. This is a physical requirement for a certain type of streetlight, and also allows the ZigBee module to join the network and notify the gateway. After these 5 minutes, the control module can gradually reduce its intensity according to a certain intensity plan. The module comes with a standard intensity plan, but a new one can also be sent from the control centre. During the operation time, a variety of commands can be sent to the module to control its operation or acquire information about the module and the streetlight. As the power goes off at the end of the night, some information is stored in the EEPROM, which then can be used the next night.

3.5.3 Communication

The PIC processor communicates with a ZigBee module that is also located on the board. The ZigBee module is configured in AT mode, which roughly means that the payload of the packets that are received by the ZigBee module are extracted and sent to the PIC. Similarly, the data that the PIC sends to the ZigBee module is automatically packetized and sent to the gateway. As such, it provides transparent communication with the gateway from the side of the control module, and the software does not have to deal with matters such as checksums and addressing.

When the module is powered, it connects to the gateway with the best signal, and remains in that network until the power goes off. The ZigBee module will also send a notification to the gateway when it joins its network. From that point on, all messages from the control module are automatically sent to that gateway.

3.5.4 Light intensity plan

The light intensity is regulated according to a plan that is sent to the control module. The plan contains times and light intensities for all seven nights of the week. Each night is divided into ten intervals. The plan specifies at which time these intervals begin, and what intensity percentage corresponds to that interval. When the power is turned on, and the ZigBee notifies the gateway about joining the network, the current time and day of the week are sent to the control module. With this information, the control module can follow the plan and dim the light accordingly throughout the night without any other communication.

3.5.5 Autonomous operation

When there is no connection with the gateway, or when transmission of intensity plan or day and time information fails, the control module will automatically follow a standard plan that is hardcoded into the module. This plan contains the same information as the week-based intensity plan, but just for one day.

However, when the module is powered, the module does not know yet what time it is, which it needs to determine when a certain interval begins. To solve this, the control module contains an advanced algorithm that is used to estimate the time it was turned on, using data collected over the past days. From that point on, the module uses this time to follow the standard plan.

This mechanism prevents unwanted situations, such as a reduction of light intensity at an unwanted time, for example during traffic rush hours. This is potentially dangerous and should be avoided. It also prevents different streetlights to emit light at a different intensity. This situation is unwanted, because it makes people call the utility when they notice the difference. If all streetlights emit light at a similar intensity, people do not tend to notice.

3.5.6 Current and voltage measurement

The control module contains a chip that measures the voltage and current of the streetlight. With this information, we can also calculate the apparent power, and add up these values every second to obtain an accurate estimation of the use of energy. This can be very useful information, because it allows the user to experiment with different intensity plans and find out how much energy was saved.

4. Case study

In order to evaluate our smart grid last-mile communications model using ZigBee and UMTS, we decided to perform some experiments and register how our system performs. To accomplish this, we developed a program that allows us to control the communications with a series of smart meters and smart streetlights on a detailed level. This software registers statistics such as the amount of messages sent, the success rate of the communications, etc. We will use this software to run some experiments and then analyze the results to examine how well our system performs.

4.1 Software

We developed a program in Visual C++, which provides a console interface to schedule communications to ZigBee end-points in our system. On the console, a number of commands can be entered to perform certain actions, such as adding a job, viewing which jobs are scheduled and what the results of the communications have been. Using the command line, the user can control jobs that are scheduled in the scheduler. The scheduler regulates which jobs need to run at which moment.

4.1.1 Job

A job is basically a collection of information about a certain message or series of messages that need to be sent to a specific end-point. A job can run once, or at a specified interval. Below is the job class definition.

```
class job {
  public:
    job(char m[256], Gateway *gw, bool per, bool sr, int intv, bool is_meter, string
mod); // Constructor
    void run(); // Main loop
    char message[256]; // Message to send
    Gateway *gateway; // Gateway to the module
    bool periodic; // Should job be repeated?
    bool should_reply; // Do we expect a reply from an end-point?
    bool pending; // indicates the message needs to be sent asap
    int interval; // Interval between jobs
    int count_down; // How long until next job?
    bool is_metering; // Is it a metering job?
    bool terminate; // Signal job to end
    string module; // End-point address
};
```

The job has only two functions; the constructor and the `run()` function. This last function is called once and keeps looping until the program stops. In this function, it determines when its messages should be sent. It does so by decrementing a timer, and setting `pending` to true once the timer reaches zero. This notifies the scheduler that the job needs to be performed. Below is the `run()` function.

```
void job::run() {
  unsigned long loopt;
  extern Scheduler *pSch; // Pointer to the scheduler
  extern bool pause; // Bool to pause operation
  count_down = interval * 10; // Initial countdown
  time_t tijd = time(NULL); // Now
  tm *local = localtime(&tijd); // Local time
  long oldsec = local->tm_sec;
  long newsec; // oldsec and newsec to detect second elapsed
  // Keep decrementing the count_down until it reaches 0
  // then pending is set to tue, and the counter is restarted
  while (!terminate) {
    Sleep(100);
    tijd = time(NULL); // Get the time again
    local = localtime(&tijd); // Create local time
    newsec = local->tm_sec; // Extract seconds
    if (newsec != oldsec) { // See if a second has passed
      oldsec = newsec; // Pass to oldsec
      if (!pause && pSch->InActiveTime()) {
        count_down -= 10; // Tick
        if (count_down <= 0) {
          // The job needs to be run
          if (pending) {
            // We are already pending, this means we are too slow: skip
            pSch->skipped++;
            if (strstr(message, "Transparant")) pSch->m_skipped++;
          }
          pending = true;
          count_down = interval * 10; // Reset countdown
        }
      }
    }
  }
}
```

4.1.2 Scheduler

The scheduler checks every job to see if it is pending, and then executes that job if it is. The scheduler also keeps track of the communications statistics, such as how many messages have been sent, how many have been answered, etc. The class definition is shown below.

```
class Scheduler {
  public:
    Scheduler(); // Constructor
    ~Scheduler(); // Destructor
    void run(); // The basic loop of the scheduler
    void add_job(char message[256], Gateway *gateway, bool periodic, bool
should_reply, int interval, bool metering = false, string module = ""); // Add a job
    bool rm_job(int n); // Remove a job
    void print_jobs(); // Print info about jobs
    void show_comm(); // Print communication on the screen
    void hide_comm(); // Don't print communication on the screen
    bool get_show(); // Should communication be printed?
    int get_max_job_length(); // Maximum length of a job message
    void rearrange(); // Rearrange jobs with similar intervals to spread them out
    unsigned char CalculateCrc(unsigned char pData[]); // Checksum calculation
    void DestroyAllSockets(); // Cleanup
    void SetActiveTimes(const char *start_time, const char *end_time); // Set when
communication ends / starts
    bool InActiveTime(); // Are we in the activity time?

    job *seljob; // The selected job

    vector <job*> jobs; // Vector of all the jobs
    // statistics for all messages (including metering messages)
    long totaltime; // Time we are running
    long msg_count; // Messages sent
    long msg_fail; // Failed messages
    long msg_success; // Successful messages
    long connectfail; // Connection failures
    long ackfail; // Acknowledgement failed
    long replyfail; // Reply failed
    long skipped; // Job was skipped
    // statistics for metering messages only
    long m_totaltime; // see above
    long m_job_count; // see above
    long m_job_fail; // see above
    long m_job_success; // see above
    long m_connectfail; // see above
    long m_ackfail; // see above
    long m_replyfail; // see above
    long m_skipped; // see above

    long socket_to; // Socket timeout in ms

    unsigned long SentBytes; // Total amount of bytes sent
    unsigned long RecvBytes; // Total received bytes
    unsigned long SentZigBytes; // Bytes sent by the ZigBee
    unsigned long RecvZigBytes; // Bytes received by the ZigBee

  private:
    void add_job(job* newjob); // Add a job

    // Send a message
    bool SendMsg(char message[], Gateway *to, bool should_reply, char *term_seq =
NULL);
    // Parse a received message
    bool ParseMessage(char msg[], char command[], char arg1[], char arg2[], int &i);
    string GetMod(int mod, char arg[]); // Find a module
    void prnt(string out); // Print a message to the screen and log file
    bool is_element(vector<int> vec, int val); // Check if exists in vector
    int num_occ(int val); // return number of occurrences of val in jobs intervals
    bool show; // Print communication to screen?
    char logl[1024]; // Line to print to log
```

```
    active_time start_t; // Time to start
    active_time end_t; // Time to end
    bool at_set; // Active Time Set?
};
```

The scheduler also contains a `run()` function in which it keeps looping as long as the program runs. This function is shown below. However, a large part of the function deals with executing a metering job. Because of the length of this piece, we abbreviated this by the theoretical `RunMeteringJob()` function, which does not actually exist in our program. What remains is a loop in which we loop over all the jobs that are contained in the scheduler. For each job, we check if it is pending, and if so, we execute the job by sending its message(s).

```
void Scheduler::run() {
  long mts, mte; // track time for jobs
  extern bool pause;
  bool success;
  while (true) {
    Sleep(100);
    EnterCriticalSection(&CriticalSection);
    if (!pause && InActiveTime()) {
      for (int i = 0; i < jobs.size(); i++) {
        seljob = jobs[i];
        success = false;
        if (jobs[i]->pending) {
          // We need to send the message
          jobs[i]->pending = false;

          // See if it's a metering job
          if (strstr(jobs[i]->message, "Transparant")) {
            RunMeteringJob();
          } // if Transparant
          else {
            SendMsg(jobs[i]->message, jobs[i]->gateway, jobs[i]->should_reply);
          } // else: no metering job
          // If not periodic, erase the job
          if (!jobs[i]->periodic) rm_job(i);
        }
      }
    }
    LeaveCriticalSection(&CriticalSection);
  }
}
```

4.1.3 Gateways and modules

The IP addresses and ports of gateways are defined in a text file. Because we are using UMTS modems, these really are the addresses of the UMTS modems, but it allows us to transparently connect to the gateways. The modules are also specified in a text file. They are defined by either only their ZigBee address for the streetlight modules, or by both the ZigBee address and the electricity meter identifier for the smart meters, which is required to communicate with the meter. Below is an example of a modules file that defines two smart meters and two streetlight modules.

```
0x0013A20040477C54, 87823121
0x0013A200405E0968, 87856549
0x0013A200405C821A
0x0013A200405E0989
```

These two files are automatically loaded by the program on start-up.

4.1.4 Commands and scripts

To control the behaviour of the program and to add or remove jobs, the user enters commands on the command line of the program. Below is a table of the available commands and a short description:

Command	Description
help / list	Print a list of commands
pause	Put all communication on hold
run	Resume communication
jobs	Show the table of jobs
gateways	Show the table of gateways
modules	Show the table of modules
selgw	Select a gateway
selmod	Select a module
exit / quit / q	Exit the program
addjob	Add a job to the scheduler
addmjob	Add a meter job to the scheduler
rmjob	Remove a job
rmall	Remove all jobs
setto	Set the ack and reply timeout in milliseconds
comm	Show communication with gateway. The command prompt will not be available. Press Ctrl+C to return
stats	Print success rate and average time
file	Execute commands stored in a file
arr	Arrange the coundown of the jobs, such that the timegaps are equal for similar interval times
sset	Set the start and end times - No messages will be sent outside of this interval Format: 'hh:mm' or 'hh:mm:ss'
stopt	Stop all communication after this period (minutes)

When a command is typed that needs some parameters, the user will be prompted for this information. However, it is also possible to type this data in the same line to immediately include the arguments.

The `file` command also enables the use of files in which commands are stored. This way, we can make scripts that initiate a certain experiment. Below is an example of such a script. The lines that start with a hash sign (#) are comments, and are not processed by the program.

```
pause
# set stop time to 15 minutes
stopt 15

# 1st meter
selmod 1
addmjob y 10

# 2nd meter
selmod 2
addmjob y 10

# 1st streetlight module
selmod 3
addjob Geef_Update y y 10

# 2nd streetlight module
selmod 4
addjob Geef_Update y y 10

# set timeout to 2 seconds
setto 2000
# rearrange the jobs in time
arr
# print the jobs on screen
jobs
# start the communications
run
# print communication on screen
comm
```

This script first pauses any scheduler activity to prevent the program from starting before all configurations have been made. It then sets the end-time to 15 minutes from now. Next it starts adding the jobs to the scheduler. First two metering jobs are added, and then two jobs are added to get a status update from two streetlights. These jobs all occur at an interval of 10 seconds. The timeout is set to 2 seconds.

The `arr` command rearranges the triggers of the jobs in such a way, that they are not triggered to be run at the exact same time. In other words, the intervals of the jobs stay the same, but they receive an offset such that they trigger distributed over time.

4.1.5 Statistics

The scheduler keeps track of all the statistics of the jobs throughout its operation. Every job that is initiated is either successful or it fails. Three reasons exist for a job to fail:

- Connection failure: the program could not connect to the gateway, so no messages could be sent.
- Acknowledgement failure: A message was sent to the gateway, but the gateway did not respond with an acknowledgement within the specified timeout time.
- Reply failure: A message was sent to the gateway and also acknowledged, but the ZigBee end-point did not reply to the message within the specified timeout time.

Only one job can be run at a specific time. If a previous job is still being handled while a new job is scheduled to commence, the new job will wait until the current job is completed. If jobs are too densely scheduled, a job might not be able to be initiated before the next occurrence of the same job is scheduled to take place. In this case, the job will not be performed, and be marked as a *skipped job*. If a job is initiated, but does not succeed for any of the mentioned three reasons, it will be marked as a *failed job*. Retries are not implemented, such that the rate of failed jobs gives an accurate representation of the reliability of the communications.

4.1.6 Challenges

During the development of our program, we encountered several challenges. One thing we struggled with, was communicating with the smart meters. We found out that this was caused by the fact that the meters were expecting a different parity on their serial port than the ZigBee RS-232 modem was outputting. The meters were expecting 7 data bits and 1 odd parity bit, whereas the ZigBee modems were outputting 8 data bits. It turned out parity was a setting in the ZigBee modules. However, the ZigBee firmware only supports 8 data bits, or 8 data bits and 1 parity bit, but not 7 data bits and 1 parity bit.

This meant that we either had to convert the parity of the data between the ZigBee modem and the smart meter, which would require additional hardware, or convert the data softwarematically. We chose to convert the data in the software, but this required some thinking, because the ZigBee gateway protocol could not be encoded in this way, as this would make it unreadable for the gateway. In other words, we had to change the parity of only the payload of transparent messages to the meters, and leave the rest of the messages in the standard 8 bit data format. Below are the functions we used for the parity conversion.

```
void Parity8to7(unsigned char str[]) {
  int i = 0;
  while (str[i] != '\0') {

    // Print the character in binary format in a string
    char *bits;
    bits = byte_to_binary(str[i]);

    // Count the parity of the character by counting the 1s in the string
    int ones = 0;
    for (int j = 0; j < 8; j++) if (bits[j] == '1') ones++;

    // Add the parity bit to the left
    if (ones % 2) str[i] += 128;
    i++;

    delete bits;
  }
  pSch->SentZigBytes += strlen((char *) str);
}

char *byte_to_binary(int x) {
  char *b;
  b = new char[9];
  for (int i = 7; i >= 0; i--) {
    if (x % 2) b[i] = '1';
    else b[i] = '0';
    x /= 2;
  }
  b[8] = '\0';
  return b;
}
```

What we are basically doing in this function, is converting every single character in the `str[]` array to a `char` array in which the character is represented by eight `'1'` or `'0'` characters. We do this by constantly checking whether the character is even or odd, which tells us whether the last bit is a 1 or a 0, and then dividing the character by 2, which basically just shifts the byte such that the second least significant byte is now the least significant. We repeat this for all the bits in the character. This allows us to count the amount of ones in the original character to determine whether the parity bit should be a one or a zero. If the amount of ones is even, the parity bit is added by adding 128 (`10000000`) to the character.

Please note that this function will fail when any of the characters are already higher or equal to 128, but these characters cannot be represented by 7 data bits and 1 parity bit anyway.

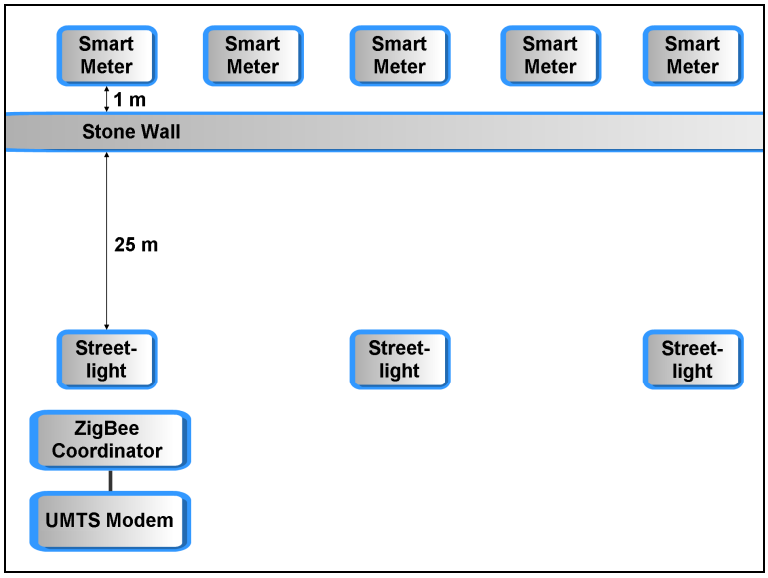
The next function converts incoming data back to a readable form.

```
void Parity7to8(unsigned char str[]) {
  int i = 0;
  while (str[i] != '\0') {
    // Remove the parity bit if it exists
    str[i] %= 128;
    if (str[i] < 32 || str[i] == 127) str[i] = '_';
    i++;
  }
}
```

This conversion is easy, because we only have to remove the parity bit if it is present. In other words, subtract 128 from a character if it is higher or equal to 128. This is the same as taking modulo 128 over the character. This function also replaces every non-printable character by an underscore (_). We purely do this because we will be printing the incoming data to the screen. This step should not be done if the converted data is actually processed later on.

4.2 Experiments

For all our experiments, we will use one gateway with a ZigBee coordinator, through which all ZigBee communications are routed. We connect to this coordinator via a UMTS modem. The network further consists of 3 ZigBee enabled streetlights and 5 ZigBee enabled smart meters, operating at a frequency of 2,4 GHz. The ZigBee coordinator and the streetlights are placed outside of a building. The distance between the streetlights and the building is 25 m. The smart meters are located inside this building, at a distance of 1 m from the wall. The 5:3 ratio effectively simulates a realistic situation, where smart meters are located inside the homes of customers, but there isn't a streetlight in front of every house. See the diagram below for a schematic representation of the setup.



The objective of our experiments is to determine how many jobs can be run per minute. In order to get a comprehensive view on the capabilities of the ZigBee network, we will be varying two different parameters during our tests:

1) **Applications**. We have two different applications which will communicate over the ZigBee network:
    a. Smart metering. This is the smart metering application, in which communications from the coordinator to the meters are required.
    b. Streetlight control. This only involves the streetlights and the coordinator. There is no communication with the meters.
2) **Jobs per minute**. We will test different time intervals between communication tasks of the application(s). In other words, we will vary the amount of jobs per minute that are scheduled to be performed.

A job for the streetlight control application consists of just one simple message over the ZigBee network from the coordinator to the streetlight, and one from the streetlight back to the coordinator. A smart metering job requires more communication; a total of six messages from coordinator to meter, and six from meter to coordinator. Because the smart meters need to transmit a lot more data per job than the streetlight control application, we will also test at different amounts of jobs per minute for the different applications. Besides separately, we will also perform experiments in which we run both applications simultaneously. We call these the *dual application* experiments. The scheduled amount of jobs for the smart metering application will always equal the scheduled amount of jobs for the streetlight control application.

The PC software registers many statistics during the experiments, which allows us to get a good insight into the performance of the applications. One of the things we are curious about, is the amount of attempted jobs that result in a successful reception of data. In other words, what is the effective success rate of the attempted jobs (S_{eff}). This is simply the percentage of jobs that succeed compared to the ones that are attempted.

Another interesting statistic is the amount of successfully performed jobs in relation to the amount of jobs that were originally scheduled (S_{sch}). This is calculated by the following formula:

$$S_{sch} = \frac{J_{suc}}{J_{sch}}$$

Where J_{suc} is the amount of successfully performed jobs, and J_{sch} is the amount of originally scheduled jobs.

In other words, S_{eff} is the success rate of the jobs that were actually initiated, S_{sch} is the success rate of the jobs that should have been run according to the schedule. We would expect S_{sch} to decline when we keep scheduling more jobs per minute, because there is only a fixed amount of jobs that can be performed in one minute, since we are only performing one job at the time. When we schedule more jobs than can be attempted, S_{sch} will drop. However, it is uncertain if S_{eff} will also decline when more jobs are scheduled. This could be the case, because more densely scheduled jobs might put more strain on the network, which could lead to a lower reliability. However, even if this is the case, it is also uncertain if this effect will be noticeable in our results.

The software also registers the average time it takes to perform a job. Using this statistic, we might be able to see if messages take longer to return when more jobs per minute are sent over the network. If we take the average of these jobtime averages, we get the total jobtime average over all experiments (t_{avg}). We would normally expect that the length of the experiment divided by t_{avg} would be equal to the amount of jobs, if we assume an S_{eff} of 100%. However, this number might be slightly off for individual experiments, because there may be some variance in the performance of the network between different times. Especially the UTMS network might perform differently at certain moments. In order to measure the variability of the network, it is useful to compare the results of individual experiments with the averages. We do this by mathematically deriving two averaging functions from our defined results statistics.

With the use of t_{avg}, we can calculate the value that S_{sch} should have for a specific experiment if the network performance was always constant, because we know how many jobs are scheduled to be performed in one minute: the amount of scheduled jobs per minute (jpm_{sch}), which also gives us the scheduled amount of jobs per second (jps_{sch}). If we use t_{avg} to calculate how many jobs can be performed in the total time of the experiment (t_{tot}), we can use this to define a function that gives the expected value of S_{sch} given a certain jps_{sch}. We get the following formula:

Averaging method 1:

$$S_{sch} = \frac{J_{suc}}{J_{sch}} = \frac{S_{eff} \cdot \frac{t_{tot}}{t_{avg}}}{jps_{sch} \cdot t_{tot}} = \frac{S_{eff} \cdot t_{tot}}{t_{avg} \cdot jps_{sch} \cdot t_{tot}} = \frac{S_{eff}}{t_{avg} \cdot jps_{sch}}$$

Another way to calculate S_{sch} for a given jps_{sch} is by using the average amount of jobs we were able to perform in the experiments. However, we should not include those experiments in which we had spare time between the jobs, because this will result in a lower maximum amount of jobs per minute, which will throw off our average. In other words, we should only take the average of all the *saturated* experiments, which means we could not have performed more jobs than we did for that experiment. This will give us the average maximum amount of jobs per second (jps_{max}). We get the following formula:

Averaging method 2:

$$S_{sch} = \frac{J_{suc}}{J_{sch}} = \frac{jps_{max} \cdot t_{tot}}{jps_{sch} \cdot t_{tot}} = \frac{jps_{max}}{jps_{sch}}$$

We will later use these methods to compare the individual results of the experiments to the averages and see if our results differ from these methods at certain points.

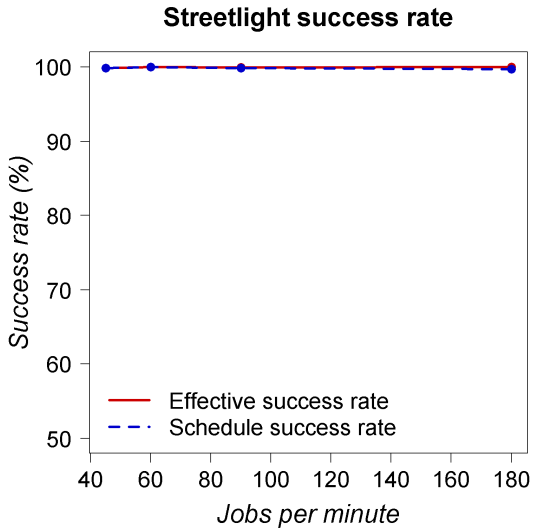
4.3 Results

We first performed experiments using the streetlight control application only. The results are shown in the table below.

jpm_{sch}	S_{eff}	S_{sch}	t_{avg} (messages)
45	99,852	99,852	0,493
60	100,00	100,00	0,185
90	99,926	99,851	0,433
180	99,963	99,703	0,189

The table shows the amount of scheduled jobs per minute, the effective success rate, the scheduled success rate and the average time of the messages that were sent.

The graph below shows the success rates of the streetlight experiments.



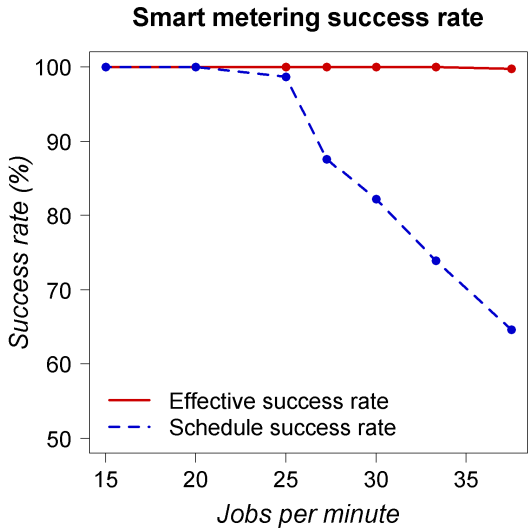
The data shows that we've been able to perform 180 requests per minute to the streetlights, with a success rate of close to 100%. This means that performing 180 jobs per minute was not a problem for the streetlight control application, and sending with a higher jpm_{sch} did not result in a lower success rate.

For the next experiments, we only sent requests to the smart meters. Please note, that the streetlights were involved in the network, so communications could be routed via the streetlights, but there was no application level interaction with the streetlights. The results are shown below.

jpm_{sch}	S_{eff}	S_{sch}	t_{avg} (messages)	J (metering)	t_{avg} (metering)
15	100,00	100,00	0,444	224	2,439
20	100,00	100,00	0,448	299	2,413
25	100,00	98.649	0,444	365	2,431
27,27	100,00	87.654	0,458	355	2,496
30	100,00	82.183	0,443	369	2,421
33,33	100,00	73.948	0,454	369	2,418
37,5	99,725	64.706	0,461	363	2,458

In this table, the second column from the right represents the total amount of metering jobs that were attempted in the experiment. The rightmost column is the average time of the metering jobs during the experiment.

The graph below shows the success rates during these experiments.

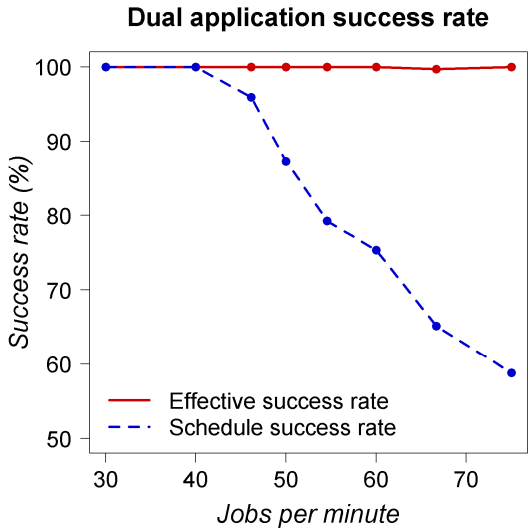


The results show that we were not able to perform 25 or more meter readouts in one minute. After that, the scheduled success rate steadily declined, as was to be expected. The effective success rate stayed at 100%, with exception of the last experiment at a jpm_{sch} of 37,5, but it was not significantly lower and might just be coincidental.

Finally, we performed the dual application experiments. Here we combined the smart metering and streetlight control applications. The amount of jobs were distributed evenly, which means that a similar amount of jobs were scheduled for both applications. The results are shown in the table below.

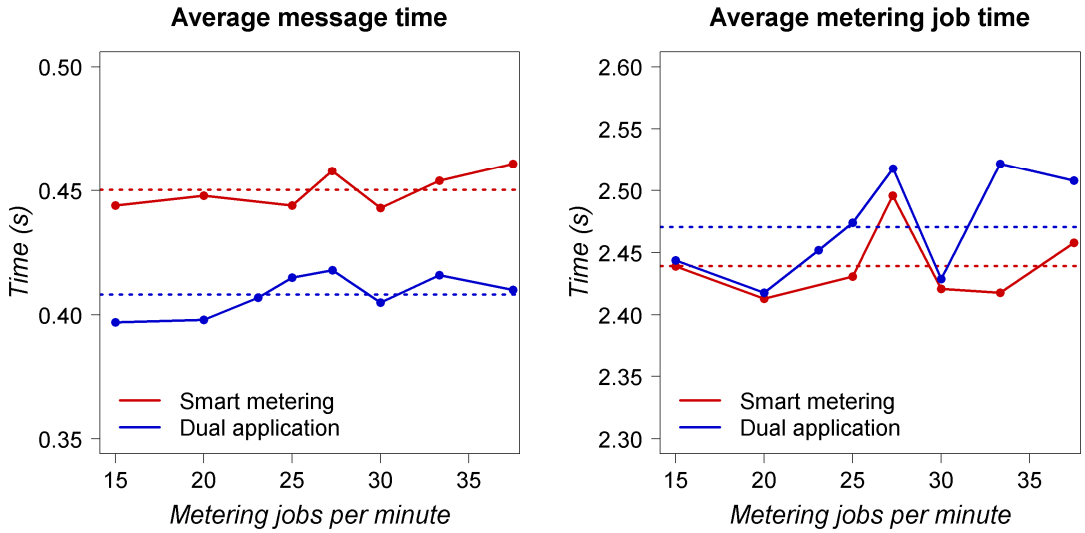
jpm_{sch}	S_{eff}	S_{sch}	t_{avg} (messages)	J (metering)	t_{avg} (metering)
30	100,00	100,00	0,397	224	2,444
40	100,00	100,00	0,398	229	2,418
46,15	100,00	95,942	0,407	331	2,452
50	100,00	87,366	0,415	325	2,474
54,55	100,00	79,268	0,418	325	2,518
60	100,00	75,333	0,405	339	2,429
66,67	99,692	65,191	0,416	324	2,522
75	100,00	58,824	0,410	330	2,508

The graph below shows the success rates for the smart metering jobs during these experiments.



This graph shows a similar shape as the previous one. The jpm_{max} seems to lie around 40 here. Since half of the jobs were metering jobs, we could maximally perform around 20 metering jobs per minute. This is slightly lower than in the previous experiments, which is straightforward, since we simultaneously had the streetlight control application running.

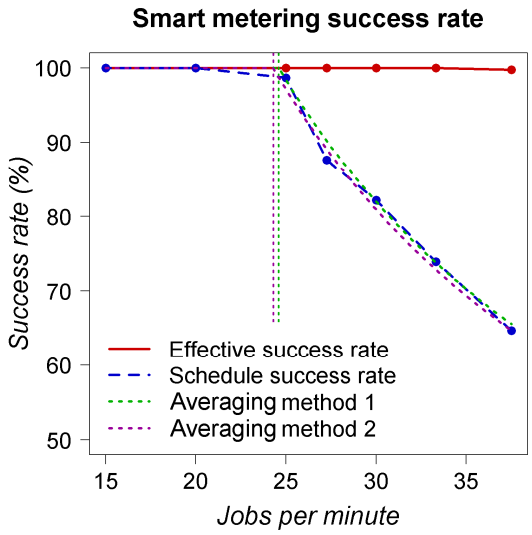
The next two graphs show the average message times and the average metering job times respectively, for both the smart metering experiments and dual application experiments. The dotted horizontal line represents the total average over all the experiments.



The first graph shows that the average message took longer for the smart metering application than for the dual application experiments. This is obviously caused by that fact that the streetlight control application messages are included in this average, which are faster than the smart metering messages. In the second graph, we see that for all experiments, the average metering job takes longer in the dual application experiment than in the smart metering experiment. This also clearly results in a higher overall average. Looking at both graphs, it gives the impression that the average time increases for a higher jpm_{sch}. The fact that in the second graph the dual application average is higher than the smart metering average also supports this idea.

4.4 Analysis

In this section we will apply the averaging methods we derived to the results of the experiments, and compare it to the individual experiment results. Since S_{eff} is 100% in almost all experiments, we will adopt this value in the averaging functions. In other words, we assume that all initiated jobs succeed. The following graph shows the two averaging methods for the smart metering experiments. The two vertical lines mark the intersections with the 100% line, and thus represent the average maximal jpm_{sch} which results in a S_{sch} of 100% according to the respective method.



We can see here that a minimal difference exists between the two averaging methods. Method 1 gives a higher S_{sch} for all saturated experiments. We also observe that none of the individual results differ greatly from the averages. Both methods give a good average of the actual results of the experiments, and neither can be easily identified as the best method.

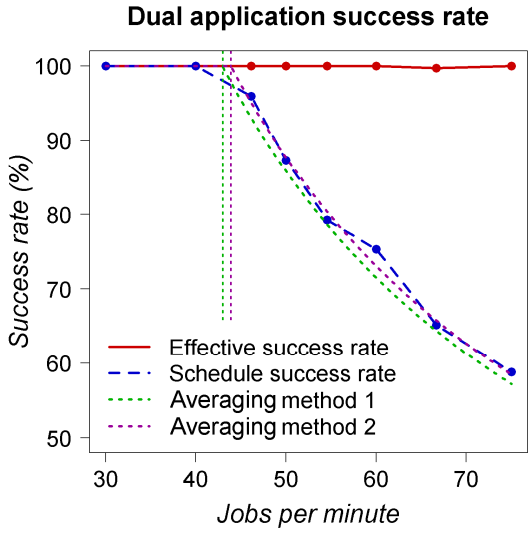
We will now proceed to calculate the jpm_{max} for the first averaging method, assuming an S_{eff} of 100%. The experiments reveal a t_{avg} of 2,439 for the smart metering experiments. This results in the following calculation:

$$\frac{S_{eff}}{t_{avg} \cdot jps_{sch}} = 1 \Rightarrow \frac{1}{2{,}439 \cdot jps_{sch}} = 1 \Rightarrow jps_{sch} = 0{,}4100$$

Multiplied by 60, this leads to an average jpm_{max} of 24,60.

The experiments resulted in an average of 364,2 jobs at a t_{tot} of 15 minutes, which means a jpm_{max} of 24,28. By definition, this is also the average jpm_{max} of averaging method 2.

The next graph shows the averaging methods for the dual application experiments.



Both averaging methods again show similar results. However, this time method 2 gives a higher S_{sch} for all saturated experiments. The actual results from the experiments are again quite close to the averages, with only the results at a jpm_{sch} of 60 differing significantly from both averaging methods.

The experiments resulted in a t_{avg} of 1,398. Again, we calculate jpm_{max} according to averaging method 1, assuming an S_{eff} of 100%:

$$\frac{S_{eff}}{t_{avg} \cdot jps_{sch}} = 1 \Rightarrow \frac{1}{1{,}398 \cdot jps_{sch}} = 1 \Rightarrow jps_{sch} = 0{,}7153$$

This results in a jpm_{max} of 42,92.

The experiments performed an average of 658,0 jobs at a t_{tot} of 15 minutes, which results in a jpm_{max} of 43,87. By definition, this is also the average jpm_{max} of averaging method 2.

4.5 Case study conclusions

Our experiments have shown that when jobs are handled one by one, nearly all jobs are completed successfully. Especially the streetlight control application was very reliable. It was also faster than the smart metering application, since a lot more jobs per minute could be completed. But the smart metering results also show reliable communication with the smart meters, which means that ZigBee has no problem communicating through a stone wall. Similarly, the dual application experiments resulted in a high effective success rate.

We formulated two methods to average the amount of jobs that can be completed relative to the amount of jobs that are scheduled, using the values we gathered from the experiments. Comparing the averages of the two methods reveals that they are quite similar. The differences between them can be explained by the fact we assumed an S_{eff} of 100% for the first averaging method, which is quiet accurate but not completely correct. Although method 1 resulted in a higher success rate than method 2 for the smart metering experiments, method 2 revealed a higher success rate than method 1 for the dual application experiments. By comparing the averaging methods to the actual results, we can conclude that the individual results do not differ greatly from the averages. This means that the performance of the network is quite stable throughout the experiments, which contributes to its reliability.

We also compared the average time of a metering job during the individual experiments, as well as the average time of individual messages. The results give the impression that the average jobtime increases as the number of performed jobs increase as well. This is both supported by the slight increase of message- and jobtime relative to the amount of jobs, and the fact that the dual application experiments had a higher smart metering jobtime average than the smart metering application. The latter observation supports the idea that jobs take longer when more jobs are scheduled, because the dual application experiments are communicating with the streetlights in addition to the smart metering jobs. This means that the dual application experiments had more data being transmitted over the network than the smart metering experiments with an equal amount of smart metering jobs. However, the gathered data is not convincing enough to conclude that this impression is valid. Additional research is needed to confirm or disprove our assumption.

5. Conclusions

In chapter 2, we summarized the requirements that smart grid communications networks need to meet. The first of these was reliability. In our case study we have shown that our system is capable of very reliable communications. Not only did we manage an effective success rate of nearly 100% is all cases, but we also showed that the performance of the network doesn't deviate so much, which means that the datarate and latency are stable.

The amount of messages we were able to send was more than enough for the amount of end-points we used, and is probably also enough when many more end-points are added. This of course depends on the requirements of the specific application, but when a higher datarate is required, we can simply add a gateway to create a new network, and thus double the datarate of the entire system.

We did not research the security of the network in this paper, but related research indicated that the security of ZigBee networks is tight [33], and public carrier networks are widely trusted as well.

Adaptability, upgradability and flexibility are the most important features of our solution. This is reflected in many aspects of the network. ZigBee networks are very flexible, because they are self-forming and self-healing. Relocating and adding end-points or coordinators can be done without any changes in the configuration, as long as there is a possible path from an end-node to a coordinator. Because gateways can be added at any point, this also enables us to adapt to new requirements of the applications and the network. It also allows us to adapt the networks to local requirements, without changing the functional behaviour of the system. Lastly, due to the transparent communications throughout the system, new applications can be easily added without having to change the software of the gateway or any other hardware in the field.

We have also shown that our system is well suited for the readout of smart electricity meters, which means that it is a viable means to reach the requirements set by the European Union to equip 80% of all European households with smart meters.

6. Future Research

We have experimented with a small-scale setup of smart meters and streetlights in a largely controlled environment. In a realistic situation, more meters and streetlights will be involved in the network, which will probably change the properties of the network as well. Researching the behaviour of larger networks could be useful to predict how many meters can be handled by one ZigBee network in combination with certain requirements.

Additionally, the distribution of communication with the streetlights could be varied, resulting in a better approximation of reality, in which communications can be demanding during short periods, but sparse for the rest of the time.

Sending messages simultaneously to different smart meters or streetlights could also be experimented with. In our experiments, we always waited for one job to complete before the next was initiated. It would be interesting to see what happens if multiple jobs are being handled simultaneously.

Also, the physical distances between the various objects in the setup could be altered and the effects examined. For example, increasing the distance between the streetlights and wall of the building might slow down communications or decrease effective success rate.

The obstruction of other objects and materials could also be experimented with, such as vegetation and different types of walls. High-rise blocks also form an interesting situation. Smart meters situated on multiple stories require hops through multiple floors and walls, which might burden communications significantly.

Other interesting research could be performed with other mesh grid protocols. For example, 6LoWPAN, which implements the IP protocol over wireless mesh grid, is based on the same IEEE standard as ZigBee. ZigBee is a new protocol for many utilities, whereas knowledge about IP technology is present in every major organisation. Besides this difference, 6LoWPAN shares many of the specifications of ZigBee, such as the frequency bands in which it operates. As a result, it also has a similar range and datarate. ZigBee is also developing an IP based variation which could be interesting.

Finally, other frequencies for ZigBee could be examined. In Europe, the 868 MHz radio band is also license-free. This frequency potentially provides a better reach, but a lower datarate.

7. Acknowledgements

For certain, I could not have written this paper without the help of a variety of people. First, I would like to thank all the people at Westland Infra who have helped me out by showing me around, assisting in realizing the communications with the meters, and conducting the experiments. Especially my internship supervisor, Jeroen Gronsveld, helped me out greatly by thinking with me and arranging anything I needed to complete my work.

Secondly, I would like to thank Todor Stafanov of Leiden University for guiding me through the process of writing this thesis, keeping me on the right track, yet giving me the freedom to explore and invest the things I wanted.

Lastly, I would like to thank the people from Delmation BV, the company at which we developed the streetlight control system, and where all this work started. I would especially like to thank my father, Kees van der Drift, for trusting in my skill and helping me with anything I needed to complete my research, and Peter Lorsheijd, who helped me a lot during the development phase of the system.

8. References

[1] Smart Grid: A comparison between Europe, the USA and Asia
B.J.C. van der Drift
Leiden Institute of Advanced Computer Science (LIACS) Leiden University

[2] SmartGridNews
http://www.smartgridnews.com/artman/publish/article_441.html

[3] U.S. distributed generation fuel cell program
M.C. Williamsa, J.P. Strakeyb and S. C. Singhal
Journal of Power Sources, Volume 131, Issues 1-2, 14 May 2004, Pages 79-85

[4] European Smart Metering Industry Group
http://www.esmig.eu/newsstor/press-release-european-parliament-expresses-its-broad-support-for-smart-metering

[5] SmartGridNews
http://www.smartgridnews.com/artman/publish/Business_Policy_Regulation_News/Boulder-SmartGridCity-Cost-Overruns-How-Bad-is-it-Really-1868.html

[6] TradingMarkets
http://www.tradingmarkets.com/news/stock-alert/xel_boulder-smart-grid-costs-blow-up-puc-orders-more-transparency-latest-rate-increase-charges-all-col-758721.html

[7] Implementing Smart Grid Communications: Managing Mountains of Data Opens Up New Challenges for Electric Utilities
J.G. Cupp, M.E. Beehler
Burns & McDonnell

[8] Vattenfall
http://www.e-control.at/portal/page/portal/medienbibliothek/presse/dokumente/pdfs/Vattenfall_experience_in_Smart_Metering_Iiro_Rinta-Joupp.pdf

[9] Électricité Réseau Distribution France
Smart meters, ERDF continues deploying Linky
http://www.erdfdistribution.fr/medias/dossiers_presse/DP_ERDF_210610_1_EN.pdf

[10] Power line carrier interference caused by DC electric arc furnaces
G.A. Franklin, S.M. Hsu
IEEE Power Engineering Society General Meeting, 2003

[11] Power line carrier interference from HVDC converter terminals
P.J. Tatro, K.A. Adamson, M.A. Eitzmann, M. Smead
IEEE Transactions on Power Delivery, vol. 8, issue 3, 1993

[12] Powerline Carrier (PLC) Communication Systems
K.H. Zuberi
Department of Microelectronics and Information Technology
Royal Institute of Technology
IT-Universitetet, Kista, Stockholm, Sweden

[13] G. Deconinck
An evaluation of two-way communication means for advanced metering in Flanders (Belgium)
K.U.Leuven
IEEE International Instrumentation and Measurement Technology Conference, May 12–15, 2008

[14] Sensus
Licensed versus Unlicensed Spectrum for Utility Communications
http://www.sensus.com/Module/Catalog/File?id=524

[15] SmartGridNews
http://www.smartgridnews.com/artman/publish/Business_Policy_Regulation_News/Boulder-SmartGridCity-Cost-Overruns-How-Bad-is-it-Really-1868.html

[16] TradingMarkets
http://www.tradingmarkets.com/news/stock-alert/xel_boulder-smart-grid-costs-blow-up-puc-orders-more-transparency-latest-rate-increase-charges-all-col-758721.html

[17] GigaOM Network
http://gigaom.com/cleantech/xcel%e2%80%99s-smartgridcity-can-thank-fiber-for-ballooning-costs/

[18] GigaOM Network
http://gigaom.com/cleantech/does-fiber-have-a-role-in-the-smart-grid-a-tennessee-utility-thinks-so/

[19] GigaOM Network
http://gigaom.com/cleantech/smart-grid-networks-the-public-vs-private-debate/

[20] GigaOM Network
http://gigaom.com/cleantech/10-reasons-why-utilities-want-to-use-public-networks/

[21] Electric Light & Power
http://www.elp.com/index/display/article-display/3427940556/articles/utility-automation-engineering-td/volume-15/Issue_5/Features/The_Smart_Grid_Debate_Public_vs_Private_Networks.html

[22] GigaOM Network
http://gigaom.com/cleantech/10-reasons-utilities-want-to-build-their-own-smart-grid-networks/

[23] Chartwell
http://www.energylibrary.com/index.cfm/ID/7/iNewsID/197/

[24] SmartGridNews
http://www.smartgridnews.com/artman/publish/Technologies_Communications_News/Mesh-Networks-Is-Communications-Winner-in-Utility-Survey-1645.html

[25] PennEnergy
http://www.pennenergy.com/index/power/metering/display/2702271845/articles/utility-automation-engineering-td/volume-15/Issue_5/Features/Toward_a_Global_Smart_Grid_-_The_US_vs_Europe.html

[26] Electric Advisory Committee
Smart Grid: Enabler of the New Energy Economy
http://www.oe.energy.gov/DocumentsandMedia/final-smart-grid-report.pdf

[27] U.S. Department of Energy
Exploring the imperative of revitalizing America's electric infrastructure
http://www.smartgrids.eu/documents/vision.pdf

[28] European SmartGrids Technology Platform
Vision and Strategy for Europe's Electricity Networks of the Future
http://www.smartgrids.eu/documents/vision.pdf

[29] Xcel Energy
Smart Grid: A White Paper
http://smartgridcity.xcelenergy.com/media/pdf/SmartGridWhitePaper.pdf

[30] S.E . Collier
Ten Steps To A Smarter Grid
http://www.milsoft.com/downloads/presentations/10%20Steps%20to%20a%20Smarter%20Grid%2001302009.pdf

[31] World Economic Forum
Accelerating Successful Smart Grid Pilots
http://www.greenbiz.com/sites/default/files/WEF_EN_SmartGrids_Pilots_Report_2010.pdf

[32] A. Clark and C.J. Pavlovski
Wireless Networks for the Smart Energy Grid: Application Aware Networks
Proceedings of the International MultiConference of Engineers and Computer Scientists 2010 Vol II

[33] K. Liu
Performance Evaluation of ZigBee Network for Embedded Electricity Meters
KTH Electrical Engineering

[34] SmartGridNews
http://www.smartgridnews.com/artman/publish/Delivery_Distribution_Automation_News/Are-We-Building-the-Grid-Backwards-2796.html

[35] BusinessGreen
http://www.businessgreen.com/bg/news/1933073/silver-spring-networks-uk-flawed-smart-grid-future

[36] MikroElektronika
mikroC PRO for PIC
http://www.mikroe.com/eng/products/view/7/mikroc-pro-for-pic/

[37] MikroElektronika
PICFlash2
http://www.mikroe.com/eng/products/view/392/picflash2/

[38] Kanda
Handheld PIC Programmer
http://www.kanda.com/pic-keyfob-handheld.html

Appendix A. R script and raw data files

We generated the graphs in this paper using the R programming language. Below is the script that reads the raw data from the experiments and generates the graphs. It also contains the calculations of the averaging methods.

plotresults.r

```
1.  ###########################################################################
2.
3.  global_res <- 256
4.  global_w <- 1600
5.  global_h <- global_w
6.  global_jpm_sm1
7.  global_jpm_sm2
8.  global_jpm_da1
9.  global_jpm_da2
10.
11. ###########################################################################
12.
13. sl_avg <- function() {
14.   data <- read.table("C:/R/sl_jpm.dat", header=T)
15.   if (length(data$time) > 0) sl_avg <- sum(data$time) / length(data$time)
16.   else sl_avg <- 0
17. }
18.
19. ###########################################################################
20.
21. plotresults <- function(input, output, title, expected = FALSE, duap = TRUE) {
22.
23. png(file=output, width=global_w, height=global_h, res=global_res)
24.
25. # Set size of window
26. windows.options(width=6, height=6)
27.
28. # Read data from file
29. data <- read.table(input, header=T)
30.
31. # Define colors to be used for lines
32. plot_colors <- c(rgb(r=0.8,g=0.0,b=0.0), rgb(r=0.0,g=0.0,b=0.8),
    rgb(r=0.0,g=0.7,b=0.0), rgb(r=0.6,g=0.0,b=0.6))
33. line_types <- c(1, 2, 3, 3)
34.
35. # Set margins
36. par(mar=c(5, 5, 4, 2))
37.
38. # Draw graph: 1 connected line and 1 dots
39. plot(data$s, data$sent, type="o", col=plot_colors[1], lwd=3,
40.    lty=line_types[1],
41.    ylim=range(50:100),
42.    xlim=range(data$s),
43.    ylab="% Successful",
44.    axes=FALSE, ann=FALSE, pch=19)
45.
```

```
# Draw second line
lines(data$s, data$total, type="o", lty=line_types[2], lwd=3, col=plot_colors[2], pch=19)

if (expected) {
  xdat <- seq(min(data$s),max(data$s),0.1)
  ydat <- seq(min(data$s),max(data$s),0.1)
  ydat2 <- seq(min(data$s),max(data$s),0.1)

  # Calculate expected success rate based on maximum jobs

  avgjobs <- 0
  div <- 0
  # Calculate average jobs in 15 min
  for (i in data$perf) {
    if (i > 300) {
      avgjobs <- avgjobs + i
      div <- div + 1
    }
  }
  avgjobs <- avgjobs / div

  is_found1 <- FALSE
  is_found2 <- FALSE
  tavg <- sum(data$jobtime) / length(data$jobtime)
  if (duap) tavg <- tavg + sl_avg()

  cat("tavg = ", as.character(tavg), "\n")
  cat("avgjobs = ", as.character(avgjobs), "\n")

  for (i in 1:length(xdat)) {
    # Averaging method 1
    ydat2[i] <- 100 / (tavg * (xdat[i] / 60))
    if (duap) ydat2[i] <- ydat2[i] * 2
    if (ydat2[i] > 100) ydat2[i] <- 100
    else {
      if (is_found2 == FALSE) {
        last <- xdat[i]
        cat("Intersection = ", as.character(last), "\n")
        abline(v=last, lty=line_types[3], lwd=2, col=plot_colors[3], pch=19)
        is_found2 <- TRUE
      }
    }
    # Averaging method 2
    sch_jobs <- (xdat[i] * 15)
    if (duap) sch_jobs <- sch_jobs / 2
    ydat[i] <- (avgjobs / sch_jobs) * 100
    if (ydat[i] > 100) ydat[i] <- 100
    else {
      if (is_found1 == FALSE) {
        last <- xdat[i]
        cat("Intersection = ", as.character(last), "\n")
        abline(v=last, lty=line_types[4], lwd=2, col=plot_colors[4], pch=19)
        is_found1 <- TRUE
      }
          }
         }
         # Draw the lines for the averages
         lines(xdat, ydat2, type="l", lty=line_types[3], lwd=3, col=plot_colors[3], pch=19)
         lines(xdat, ydat, type="l", lty=line_types[4], lwd=3, col=plot_colors[4], pch=19)
       }

       # Set the title
       title(main=title, cex.main=1.7, font.main=2)

       # Label the x and y axes with italic font
       title(xlab= "Jobs per minute", font.lab=3, cex.lab=1.7)
       title(ylab= "Success rate (%)", font.lab=3, cex.lab=1.7, line=3.4)

       # Create a legend
       if (expected) {
```

```
116.              legend("bottomleft", c("Effective success rate", "Schedule success
    rate", "Averaging method 1", "Averaging method 2"), cex=1.5, col=plot_colors,
117.                  lty=line_types, lwd=3, bty="o", box.lwd=0, box.lty=0)
118.       }
119.       else {
120.              legend("bottomleft", c("Effective success rate", "Schedule success
    rate"), cex=1.5, col=plot_colors,
121.                  lty=line_types, lwd=3, bty="o", box.lwd=0, box.lty=0)
122.       }
123.
124.       # Make x axis tick marks without labels
125.       axis(1, las=1, cex.axis=1.5)
126.
127.       # Plot y axis with smaller horizontal labels
128.       axis(2, las=1, cex.axis=1.5)
129.
130.       # Create box around plot
131.       box()
132.
133.       dev.off()
134.
135.         if (duap) {
136.           global_jpm_da1 <<- 2 * 60 / tavg
137.           global_jpm_da2 <<- 2 * avgjobs / 15
138.         }
139.         else {
140.           global_jpm_sm1 <<- 60 / tavg
141.           global_jpm_sm2 <<- avgjobs / 15
142.         }
143.       }
144.
145.       ##########################################################################
146.
147.       plottimes <- function() {
148.       # Read the files
149.       sm_data <- read.table("C:/R/sm_jpm.dat", header=T)
150.       da_data <- read.table("C:/R/da_jpm.dat", header=T)
151.
152.       # Divide jobs by two for dual application
153.       for (i in 1:length(da_data$s)) da_data$s[i] <- da_data$s[i] / 2
154.
155.       png(file="msg_times.png", width=global_w, height=global_h,
    res=global_res)
156.
157.       # Define colors to be used for lines
158.       plot_colors <- c(rgb(r=0.8,g=0.0,b=0.0), rgb(r=0.0,g=0.0,b=0.8),
    rgb(r=0.0,g=0.6,b=0.0))
159.
160.       # Set margins
161.       par(mar=c(5, 6, 4, 2))
162.
163.       # Draw graph
164.       plot(sm_data$s, sm_data$time, type="o", col=plot_colors[1], lwd=3,
165.          ylim=range(c(0.35,0.5)),
166.
    xlim=range(min(c(min(sm_data$s),min(da_data$s))):max(c(max(sm_data$s),max(da_data
    $s)))),
167.          ylab="Time (s)",
168.          axes=FALSE, ann=FALSE, pch=19)
169.       # Draw average line
170.       avg <- sum(sm_data$time) / length(sm_data$time)
171.       abline(h=avg, lty=3, lwd=3, col=plot_colors[1], pch=19)
172.
173.       # Draw second line
174.       lines(da_data$s, da_data$time, type="o", lty=1, lwd=3,
    col=plot_colors[2], pch=19)
175.       # Draw average line
176.       avg <- sum(da_data$time) / length(da_data$time)
177.       abline(h=avg, lty=3, lwd=3, col=plot_colors[2], pch=19)
178.
179.       # Make x axis tick marks
180.       axis(1, las=1, cex.axis=1.5)
181.
182.       # Plot y axis with smaller horizontal labels
```

```
183.       axis(2, las=1, cex.axis=1.5)
184.
185.       # Create box around plot
186.       box()
187.
188.       # Set the title
189.       title(main="Average message time", cex.main=1.7, font.main=2)
190.
191.       # Label the x and y axes with italic font
192.       title(xlab= "Metering jobs per minute", font.lab=3, cex.lab=1.7)
193.       title(ylab= "Time (s)", font.lab=3, cex.lab=1.7, line=4)
194.
195.       legend("bottomleft", c("Smart metering", "Dual application"), cex=1.5,
    col=plot_colors,
196.          lwd=3, bty="n");
197.
198.       dev.off()
199.       }
200.
201.       ########################################################################
202.
203.       plotjobtimes <- function() {
204.       # Read the files
205.       sm_data <- read.table("C:/R/sm_jpm.dat", header=T)
206.       da_data <- read.table("C:/R/da_jpm.dat", header=T)
207.
208.       # Divide jobs by two for dual application
209.       for (i in 1:length(da_data$s)) da_data$s[i] <- da_data$s[i] / 2
210.
211.       png(file="job_times.png", width=global_w, height=global_h,
    res=global_res)
212.
213.       # Define colors to be used for lines
214.       plot_colors <- c(rgb(r=0.8,g=0.0,b=0.0), rgb(r=0.0,g=0.0,b=0.8),
    rgb(r=0.0,g=0.6,b=0.0))
215.
216.       # Set margins
217.       par(mar=c(5, 6, 4, 2))
218.
219.       # Draw graph
220.       plot(sm_data$s, sm_data$jobtime, type="o", col=plot_colors[1], lwd=3,
221.          ylim=range(c(2.3,2.6)),
222.
    xlim=range(min(c(min(sm_data$s),min(da_data$s))):max(c(max(sm_data$s),max(da_data
    $s)))),
223.          ylab="Time (s)",
224.          axes=FALSE, ann=FALSE, pch=19)
225.       # Draw average line
226.       avg <- sum(sm_data$jobtime) / length(sm_data$jobtime)
227.       abline(h=avg, lty=3, lwd=3, col=plot_colors[1], pch=19)
228.
229.       # Draw second line
230.       lines(da_data$s, da_data$jobtime, type="o", lty=1, lwd=3,
    col=plot_colors[2], pch=19)
231.       # Draw average line
232.       avg <- sum(da_data$jobtime) / length(da_data$jobtime)
233.       abline(h=avg, lty=3, lwd=3, col=plot_colors[2], pch=19)
234.
235.       # Make x axis tick marks
236.       axis(1, las=1, cex.axis=1.5)
237.
238.       # Plot y axis with smaller horizontal labels
239.       axis(2, las=1, cex.axis=1.5)
240.
241.       # Create box around plot
242.       box()
243.
244.       # Set the title
245.       title(main="Average metering job time", cex.main=1.7, font.main=2)
246.
247.       # Label the x and y axes with italic font
248.       title(xlab= "Metering jobs per minute", font.lab=3, cex.lab=1.7)
249.       title(ylab= "Time (s)", font.lab=3, cex.lab=1.7, line=4)
250.
```

```
251.       legend("bottomleft", c("Smart metering", "Dual application"), cex=1.5,
    col=plot_colors,
252.          lwd=3, bty="n");
253.
254.       dev.off()
255.       }
256.
257.       ###########################################################################
258.
259.       plotresults("C:/R/sl_jpm.dat", "sl_jpm.png", "Streetlight success rate")
260.       plotresults("C:/R/sm_jpm.dat", "sm_jpm.png", "Smart metering success
    rate")
261.       plotresults("C:/R/da_jpm.dat", "da_jpm.png", "Dual application success
    rate")
262.       plotresults("C:/R/sm_jpm.dat", "est_sm_jpm.png", "Smart metering success
    rate", TRUE, FALSE)
263.       plotresults("C:/R/da_jpm.dat", "est_da_jpm.png", "Dual application
    success rate", TRUE)
264.       plottimes()
265.       plotjobtimes()
266.
267.       cat("Smart metering max JPM method 1: ", as.character(global_jpm_sm1),
    "\n")
268.       cat("Smart metering max JPM method 2: ", as.character(global_jpm_sm2),
    "\n")
269.       cat("Dual application max JPM method 1: ", as.character(global_jpm_da1),
    "\n")
270.       cat("Dual application max JPM method 2: ", as.character(global_jpm_da2),
    "\n")
271.
272.       ###########################################################################
```

sl_jpm.dat

This is the raw data of the streetlight experiments. The values under `s` represent jpm_{sch}, `sent` represents S_{eff}, `total` represents S_{sch}, and `time` is the average time of a single message.

```
s   sent    total   time
180 99.963  99.703  0.189
90  99.926  99.851  0.433
60  100.0   100.0   0.185
45  99.852  99.852  0.493
```

sm_jpm.dat

This is the raw data for the smart metering experiments. The first columns are the same as in the previous file. The values under `perf` represent the amount of attempted metering jobs, and `jobtime` is the average time it took to perform a metering job.

```
s          sent    total   time    perf    jobtime
37.5       99.725  64.706  0.461   363     2.458
33.33      100.0   73.948  0.454   369     2.418
30         100.0   82.183  0.443   369     2.421
27.27      100.0   87.654  0.458   355     2.496
25         100.0   98.649  0.444   365     2.431
20         100.0   100.0   0.448   299     2.413
15         100.0   100.0   0.444   224     2.439
```

da_jpm.dat

These are the results of the dual application experiments.

s	sent	total	time	perf	jobtime
30	100.0	100.0	0.397	224	2.444
40	100.0	100.0	0.398	229	2.418
46.15	100.0	95.942	0.407	331	2.452
50	100.0	87.366	0.415	325	2.474
54.55	100.0	79.268	0.418	325	2.518
60	100.0	75.333	0.405	339	2.429
66.67	99.692	65.191	0.416	324	2.522
75	100.0	58.824	0.410	330	2.508

Appendix B. C++ source code

This appendix contains the source code of the program that we developed to perform the experiments.

client.cpp

```
1.  // link WS2_32.lib
2.  #pragma comment(lib, "ws2_32.lib")
3.
4.  #include <iostream>
5.  #include <fstream>
6.  #include <string>
7.  #include <time.h>
8.  #include <vector>
9.  using namespace std;
10.
11. #include "gateway.h"
12. #include "meter.h"
13. #include "scheduler.h"
14.
15. #define DEF_GW "10.8.48.39"
16.
17. //-------------------------------------------------------------
18. // FUNCTION DEFINITIONS
19. //-------------------------------------------------------------
20.
21. void handle_cmd(istream *source);
22. void add_prompt(istream *source);
23. void add_m_prompt(istream *source);
24. void quit(int code = 0);
25. void load_gateways();
26. void print_gateways();
27. void load_modules();
28. void print_modules();
29. void add_module(char *mod);
30. void print_stats();
31. void Parity8to7(unsigned char str[]);
32. void Parity7to8(unsigned char str[]);
33. char *byte_to_binary(int x);
34.
35. //-------------------------------------------------------------
36. // GLOBAL VARIABLES
37. //-------------------------------------------------------------
38.
39. CRITICAL_SECTION CriticalSection;
40. Scheduler* pSch;
41. fstream gwf;
42. fstream mdf;
43. bool bexit;
44. vector<string> modules;
45. vector<string> meters;
46. string* selmod;
47. string* selmet;
48. vector<Gateway*> gateways;
```

```
49. Gateway* selgw;
50. bool pause;
51.
52. //-------------------------------------------------------
53. // FUNCTION IMPLEMENTATIONS
54. //-------------------------------------------------------
55.
56. // Start the scheduler thread
57. DWORD WINAPI Thread(LPVOID pParam) {
58.   Scheduler * pS;
59.   pS = (Scheduler *) pParam;
60.   pS->run();
61.   return 0;
62. }
63.
64. BOOL WINAPI ConsoleHandler(DWORD CEvent) { // To catch the Ctrl+C
65.   switch(CEvent) {
66.     case CTRL_C_EVENT:
67.       pSch->hide_comm();
68.       break;
69.     default:
70.       quit();
71.   }
72.   return TRUE;
73. }
74.
75. int main() {
76.   system("color f0");
77.   SetConsoleTitle("ZigBee Meter");
78.
79.   cout << "  _____________________________________________________________________" <<
    endl;
80.   cout << " |  ______ _      ______             ___  ___     _              |" <<
    endl;
81.   cout << " | |___  /(_)     | ___ \\            |  \\/  |    | |             |"
    << endl;
82.   cout << " |    / /  _  __ _| |_/ / ___  ___   | .  . | ___| |_  ___ _ __  |" <<
    endl;
83.   cout << " |   / /  | |/ _` | ___ \\/ _ \\/ _ \\  | |\\/| |/ _ \\ __|/ _ \\ '__|
    |" << endl;
84.   cout << " | ,/ /___| | (_| | |_/ |  __/  __/  | |  | |  __/ |_|  __/ |    |" <<
    endl;
85.   cout << " | \\_____/|_|\\__, |_____/\\___|\\___|  |_|  |_|\\___|\\__|\\___|_|
    |" << endl;
86.   cout << " |            __/ |                                               |" <<
    endl;
87.   cout << " |           |___/                                                |" <<
    endl;
88.   cout << " |_____________________________________________________________________|" <<
    endl << endl;
89.
90.   cout << endl << "ZigBee Meter - (c) Bart van der Drift, 2010" << endl;
91.   cout << "Leiden University - Westland Infra" << endl;
92.   cout << endl << "Type 'help' for a list of commands" << endl << endl;
93.   string command;
94.
95.   if (SetConsoleCtrlHandler( (PHANDLER_ROUTINE)ConsoleHandler,TRUE)==FALSE)
96.     cout << "Unable to install handler!\n";
97.
98.   pSch = new Scheduler;
99.
100.         load_gateways();
101.         load_modules();
102.
103.         InitializeCriticalSection(&CriticalSection);
104.
105.         // Start de scheduler
106.         HANDLE hThread = CreateThread(NULL, 0, Thread, pSch, 0, NULL);
107.
108.         pause = false;
109.         bexit = false;
110.
111.         while (!bexit) {
112.           handle_cmd(&cin);
```

```
113.         }
114.
115.         return 0;
116.       }
117.
118.       void handle_cmd(istream *source) {
119.         string command;
120.         string indicator = "--?> ";
121.         if (source == &cin) cout << endl << indicator;
122.         *source >> command;
123.         if (source != &cin && command[0] != '#') cout << endl << "[" << command
    << "] ";
124.         if (command == "list" || command == "help") {
125.           cout << "List of commands:" << endl;
126.           cout << " +---------------------+-------------------------------------
    ----------------+" << endl;
127.           cout << " | COMMAND             | DESCRIPTION
    |" << endl;
128.           cout << " +---------------------+-------------------------------------
    ----------------+" << endl;
129.           cout << " | help / list         | Print a list of commands
    |" << endl;
130.           cout << " +---------------------+-------------------------------------
    ----------------+" << endl;
131.           cout << " | pause               | Put all communication on hold
    |" << endl;
132.           cout << " +---------------------+-------------------------------------
    ----------------+" << endl;
133.           cout << " | run                 | Resume communication
    |" << endl;
134.           cout << " +---------------------+-------------------------------------
    ----------------+" << endl;
135.           cout << " | jobs                | Show the table of jobs
    |" << endl;
136.           cout << " +---------------------+-------------------------------------
    ----------------+" << endl;
137.           cout << " | gateways            | Show the table of gateways
    |" << endl;
138.           cout << " +---------------------+-------------------------------------
    ----------------+" << endl;
139.           cout << " | modules             | Show the table of modules
    |" << endl;
140.           cout << " +---------------------+-------------------------------------
    ----------------+" << endl;
141.           cout << " | selgw               | Select a gateway
    |" << endl;
142.           cout << " +---------------------+-------------------------------------
    ----------------+" << endl;
143.           cout << " | selmod              | Select a module
    |" << endl;
144.           cout << " +---------------------+-------------------------------------
    ----------------+" << endl;
145.           cout << " | exit / quit / q     | Exit the program
    |" << endl;
146.           cout << " +---------------------+-------------------------------------
    ----------------+" << endl;
147.           cout << " | addjob              | Add a job to the scheduler
    |" << endl;
148.           cout << " +---------------------+-------------------------------------
    ----------------+" << endl;
149.           cout << " | addmjob             | Add a meter job to the scheduler
    |" << endl;
150.           cout << " +---------------------+-------------------------------------
    ----------------+" << endl;
151.           cout << " | rmjob               | Remove a job
    |" << endl;
152.           cout << " +---------------------+-------------------------------------
    ----------------+" << endl;
153.           cout << " | rmall               | Remove all jobs
    |" << endl;
154.           cout << " +---------------------+-------------------------------------
    ----------------+" << endl;
155.           cout << " | setto               | Set the ack and reply timeout in
    milliseconds      |" << endl;
```

```
156.            cout << " +---------------------+-----------------------------------
    ----------------+" << endl;
157.            cout << " | comm                | Show communication with gateway.
    The command prompt|" << endl;
158.            cout << " |                     | will not be available. Press Ctrl+C
    to return       |" << endl;
159.            cout << " +---------------------+-----------------------------------
    ----------------+" << endl;
160.            cout << " | stats               | Print success rate and average time
    |" << endl;
161.            cout << " +---------------------+-----------------------------------
    ----------------+" << endl;
162.            cout << " | file                | Execute commands stored in a file
    |" << endl;
163.            cout << " +---------------------+-----------------------------------
    ----------------+" << endl;
164.            cout << " | arr                 | Arrange the coundown of the jobs,
    such that the     |" << endl;
165.            cout << " |                     | timegaps are equal for similar
    interval times       |" << endl;
166.            cout << " +---------------------+-----------------------------------
    ----------------+" << endl;
167.            cout << " |                     | Set the start and end times - No
    messages will be  |" << endl;
168.            cout << " | sset                | sent outside of this interval.
    |" << endl;
169.            cout << " |                     | Format: 'hh:mm' or 'hh:mm:ss'
    |" << endl;
170.            cout << " +---------------------+-----------------------------------
    ----------------+" << endl;
171.            cout << " | stopt               | Stop all communication after this
    period (minutes) |" << endl;
172.            cout << " +---------------------+-----------------------------------
    ----------------+" << endl;
173.          } else if (command == "pause") {
174.            cout << "Communication on hold!" << endl;
175.            EnterCriticalSection(&CriticalSection);
176.            pSch->DestroyAllSockets();
177.            LeaveCriticalSection(&CriticalSection);
178.            pause = true;
179.          } else if (command == "run") {
180.            cout << "Communication resumed!" << endl;
181.            pause = false;
182.          } else if (command == "jobs") {
183.            pSch->print_jobs();
184.          } else if (command == "gateways") {
185.            print_gateways();
186.          } else if (command == "selgw") {
187.            cout << "Enter gateway number to select: ";
188.            int num;
189.            *source >> num;
190.            if (num < 0 || num >= gateways.size()) cout << "Cannot select gateway
    " << num << endl;
191.            else {
192.              selgw = gateways[num];
193.              cout << "Gateway " << num << " selected!";
194.            }
195.          } else if (command == "selmod") {
196.            cout << "Enter module number to select: ";
197.            int num;
198.            *source >> num;
199.            if (num < 1 || num > modules.size()) cout << "Cannot select module "
    << num << endl;
200.            else {
201.              selmod = &modules[num - 1];
202.              selmet = &meters[num - 1];
203.              if (*selmet == "       -") cout << "Module " << num << "
    selected!";
204.              else cout << "Module and meter " << num << " selected!";
205.            }
206.          } else if (command == "modules") {
207.            print_modules();
208.          } else if (command == "exit" || command == "quit" || command == "q") {
209.            quit();
```

```
210.            bexit = true;
211.          } else if (command == "addjob") {
212.            add_prompt(source);
213.          } else if (command == "addmjob") {
214.            add_m_prompt(source);
215.          } else if (command == "rmjob") {
216.            cout << "Enter jobnumber to remove: ";
217.            int num;
218.            *source >> num;
219.            cout << "Removing job..." << endl;
220.            EnterCriticalSection(&CriticalSection);
221.            if (pSch->rm_job(num)) cout << "Job removed!" << endl;
222.            else cout << "Job could not be removed!" << endl;
223.            LeaveCriticalSection(&CriticalSection);
224.          } else if (command == "rmall") {
225.            int njobs = pSch->jobs.size();
226.            cout << "Removing jobs..." << endl;
227.            EnterCriticalSection(&CriticalSection);
228.            for (int i = 0; i < njobs; i++) pSch->rm_job(0);
229.            LeaveCriticalSection(&CriticalSection);
230.            cout << "All jobs removed!" << endl;
231.          } else if (command == "setto") {
232.            long to;
233.            do {
234.              cout << "Enter new timeout value in milliseconds: ";
235.              *source >> to;
236.            } while (to <= 0);
237.            pSch->socket_to = to;
238.          } else if (command == "comm") {
239.            cout << "Showing communication. Press Ctrl+C to return." << endl;
240.            pSch->show_comm();
241.            // loop until Ctrl+C is pressed
242.            while (pSch->get_show()) Sleep(50);
243.          } else if (command == "stats") {
244.            print_stats();
245.          } else if (command == "file") {
246.            string filename;
247.            cout << "Enter filename: ";
248.            *source >> filename;
249.            ifstream filestream;
250.            filestream.open(filename.c_str());
251.            if (filestream.is_open()) {
252.              while (!filestream.eof()) handle_cmd(&filestream);
253.              filestream.close();
254.            }
255.            else {
256.              cout << "Could not open file " << filename << "!" << endl;
257.            }
258.          } else if (command == "arr") {
259.            pSch->rearrange();
260.          } else if (command == "sset") {
261.            string t1, t2;
262.            cout << "Enter start time: ";
263.            *source >> t1;
264.            cout << "Enter end time: ";
265.            *source >> t2;
266.            pSch->SetActiveTimes(t1.c_str(), t2.c_str());
267.          } else if (command == "stopt") {
268.            int min;
269.            int h, m;
270.            char t1[10], t2[10];
271.            time_t t = time(NULL);
272.            tm *local = localtime(&t);
273.
274.            cout << "Enter time (min): ";
275.            *source >> min;
276.            m = local->tm_min + min;
277.            h = local->tm_hour;
278.            while (m >= 60) {
279.              h = (h + 1) % 24;
280.              m -= 60;
281.            }
282.            sprintf(t1, "%02d:%02d:%02d", local->tm_hour, local->tm_min, local-
    >tm_sec);
```

```
283.           sprintf(t2, "%02d:%02d:%02d", h, m, local->tm_sec);
284.           pSch->SetActiveTimes(t1, t2);
285.         } else if (command[0] == '#') {
286.           // The # is for commenting in the files - loop until linefeed is
    encountered
287.           char dum;
288.           do {
289.             dum = source->get();
290.           } while (dum != '\n');
291.           for (int i = 0; i < indicator.length(); i++) cout << '\b'; // remove
    indicator
292.         } else {
293.           cout << "Unknown command: " << command << endl;
294.         }
295.       }
296.
297.       void add_prompt(istream *source) {
298.         char mes[256];
299.         char instr[64];
300.         char inc;
301.         int ini;
302.         bool rep = false;
303.
304.         cout << "Enter command: ";
305.         *source >> instr;
306.
307.         do {
308.           cout << "Message to module? (y/n) ";
309.           *source >> inc;
310.         } while (inc != 'n' && inc != 'y');
311.
312.         if (inc == 'y') {
313.           sprintf(mes, "##%s&%s$", instr, selmod->c_str());
314.           rep = true;
315.         }
316.         else sprintf(mes, "##%s$", instr);
317.
318.         do {
319.           cout << "Repeat message periodically? (y/n) ";
320.           *source >> inc;
321.         } while (inc != 'n' && inc != 'y');
322.
323.         if (inc == 'y') {
324.           cout << "Repetition period in seconds: ";
325.           *source >> ini;
326.           pSch->add_job(mes, selgw, true, rep, ini);
327.         }
328.         else pSch->add_job(mes, selgw, false, rep, 0);
329.         cout << "Job added!" << endl;
330.       }
331.
332.       void add_m_prompt(istream *source) {
333.         string mID;
334.         char msg[256];
335.         char meterm[256];
336.         unsigned char umeterm[256];
337.
338.         // The opening message
339.         sprintf(meterm, "/?%s!!%c%c", selmet->c_str(), 0x0D, 0x0A);
340.         for (int i = 0; i < 256; i++) {
341.           umeterm[i] = meterm[i];
342.         }
343.         Parity8to7(umeterm);
344.         sprintf(msg, "##Transparant&%s&%s$", selmod->c_str(), umeterm);
345.
346.         // Periodical
347.         char inc;
348.         int ini;
349.         do {
350.           cout << "Repeat message periodically? (y/n) ";
351.           *source >> inc;
352.         } while (inc != 'n' && inc != 'y');
353.         if (inc == 'y') {
354.           cout << "Repetition period in seconds: ";
```

```
355.           *source >> ini;
356.           pSch->add_job(msg, selgw, true, true, ini, true, *selmod);
357.         }
358.         else pSch->add_job(msg, selgw, false, true, 0, true, *selmod);
359.         cout << "Job added!" << endl;
360.       }
361.
362.       void print_stats() {
363.         // Stats for all messages
364.         cout << "-----------------------------" << endl;
365.         cout << " Statistics for all messages" << endl;
366.         cout << "-----------------------------" << endl;
367.         cout << "Total messages: " << pSch->msg_count << " (" << pSch-
    >msg_count + pSch->skipped << ")"<< endl;
368.         cout << "Successful messages: " << pSch->msg_success << endl;
369.         cout << "Failed messages: " << pSch->msg_fail << endl;
370.         cout << "Connect fails: " << pSch->connectfail << endl;
371.         cout << "Ack fails: " << pSch->ackfail << endl;
372.         cout << "Reply fails: " << pSch->replyfail << endl;
373.         cout << "Skipped messages: " << pSch->skipped << endl;
374.         if (pSch->msg_count)
375.           cout << "Success rate: " << 100 * pSch->msg_success / pSch->msg_count
    << "%" << endl;
376.         else
377.           cout << "Success rate: " << "-" << endl;
378.         if (pSch->msg_success)
379.           cout << "Average time: " << pSch->totaltime / pSch->msg_success << "
    ms" << endl;
380.         else
381.           cout << "Average time: " << "-" << endl;
382.
383.         // Stats for metering jobs
384.         cout << endl << "-----------------------------" << endl;
385.         cout << " Statistics for metering jobs" << endl;
386.         cout << "-----------------------------" << endl;
387.         cout << "Total jobs: " << pSch->m_job_count << " (" << pSch-
    >m_job_count + pSch->m_skipped << ")"<< endl;
388.         cout << "Successful jobs: " << pSch->m_job_success << endl;
389.         cout << "Failed jobs: " << pSch->m_job_fail << endl;
390.         cout << "Connect fails: " << pSch->m_connectfail << endl;
391.         cout << "Ack fails: " << pSch->m_ackfail << endl;
392.         cout << "Reply fails: " << pSch->m_replyfail << endl;
393.         cout << "Skipped messages: " << pSch->m_skipped << endl;
394.         if (pSch->m_job_count)
395.           cout << "Success rate: " << 100 * pSch->m_job_success / pSch-
    >m_job_count << "%" << endl;
396.         else
397.           cout << "Success rate: " << "-" << endl;
398.         if (pSch->m_job_success)
399.           cout << "Average time: " << pSch->m_totaltime / pSch->m_job_success
    << " ms" << endl;
400.         else
401.           cout << "Average time: " << "-" << endl;
402.         cout << endl;
403.         cout << "-----------------------------" << endl;
404.         cout << endl;
405.         cout << "Total sent bytes: " << pSch->SentBytes << endl;
406.         cout << "Total received bytes: " << pSch->RecvBytes << endl;
407.         cout << endl;
408.         cout << "Sent bytes over ZigBee: " << pSch->SentZigBytes << endl;
409.         cout << "Received bytes over ZigBee: " << pSch->RecvZigBytes << endl;
410.       }
411.
412.       void quit(int code) {
413.         delete pSch;
414.         DeleteCriticalSection(&CriticalSection);
415.         cout << "See ya! :-)" << endl;
416.         exit(code);
417.       }
418.
419.       void load_gateways() {
420.         char ip[32];
421.         int port;
422.
```

```
423.          gwf.open("gateways.txt");
424.          if (!gwf) {
425.            cout << "Cannot load gateways from file" << endl;
426.            selgw = NULL;
427.          }
428.          else {
429.            // Load all the gateways
430.            while (!gwf.eof()) {
431.              gwf >> ip >> port;
432.              Gateway *ng = new Gateway(ip, port);
433.              gateways.push_back(ng);
434.            }
435.            gwf.close();
436.            if (gateways.size()) selgw = gateways[gateways.size() - 1];
437.            else selgw = NULL;
438.          }
439.        }
440.
441.        void print_gateways() {
442.          if (gateways.size()) {
443.            cout << "+----+-----------------+-------+" << endl;
444.            cout << "|  # | IP Address      | Port  |" << endl;
445.            cout << "+----+-----------------+-------+" << endl;
446.            for (int i = 0; i < gateways.size(); i++) {
447.              if (selgw == gateways[i])
448.                printf("|>%2d<| %15s | %5d |\n", i, gateways[i]->ip, gateways[i]-
    >port);
449.              else
450.                printf("| %2d | %15s | %5d |\n", i, gateways[i]->ip, gateways[i]-
    >port);
451.            }
452.            cout << "+----+-----------------+-------+" << endl;
453.          }
454.          else cout << "No gateways!" << endl;
455.        }
456.
457.        void load_modules() {
458.          mdf.open("modules.txt");
459.          if (!mdf) {
460.            cout << "Cannot load modules from file" << endl;
461.            selmod = NULL;
462.            selmet = NULL;
463.          }
464.          else {
465.            // Load all the modules
466.            while (!mdf.eof()) {
467.              string mod;
468.              string meter;
469.              string line;
470.              getline(mdf, line); // Get line from the file
471.              if (line.length() > 18) {
472.                // There's also a meter defined
473.                mod = line.substr(0, 18);
474.                modules.push_back(mod);
475.                // Now find the first character of the meter
476.                int mstart = 18;
477.                while ((line[mstart] < '0' || line[mstart] > '9') && line[mstart]
    != '\0') mstart++;
478.                meter = line.substr(mstart, 8);
479.                meters.push_back(meter);
480.              }
481.              else {
482.                modules.push_back(line);
483.                meters.push_back("       -");
484.              }
485.            }
486.            mdf.close();
487.          }
488.          if (modules.size()) {
489.            selmod = &modules[0];
490.            selmet = &meters[0];
491.          }
492.          else selmod = NULL;
493.        }
```

```
494.
495.       void print_modules() {
496.         if (modules.size()) {
497.           cout << "+----+--------------------+----------+" << endl;
498.           cout << "|  # | Module             | Meter    |" << endl;
499.           cout << "+----+--------------------+----------+" << endl;
500.           for (int i = 0; i < modules.size(); i++) {
501.             if (&modules[i] == selmod) printf("|>%2d<| %s | %s |\n", i+1,
    modules[i].c_str(), meters[i].c_str());
502.             else printf("| %2d | %s | %s |\n", i+1, modules[i].c_str(),
    meters[i].c_str());
503.           }
504.           cout << "+----+--------------------+----------+" << endl;
505.         }
506.         else cout << "No modules!" << endl;
507.       }
508.
509.       void add_module(string mod) {
510.         ofstream modf;
511.         modf.open("modules.txt", ios::app);
512.         if (!modf) cout << "cannot open module file" << endl;
513.         else {
514.           modf << endl << mod.c_str();
515.           modf.close();
516.           cout << "ädded" << endl;
517.         }
518.         modules.push_back(mod);
519.       }
520.
521.       void Parity8to7(unsigned char str[]) {
522.         int i = 0;
523.         while (str[i] != '\0') {
524.
525.           // Print the character in binary format in a string
526.           char *bits;
527.           bits = byte_to_binary(str[i]);
528.
529.           // Count the parity of the character by counting the 1s in the string
530.           int ones = 0;
531.           for (int j = 0; j < 8; j++) if (bits[j] == '1') ones++;
532.
533.           // Add the parity bit to the left
534.           if (ones % 2) str[i] += 128;
535.           i++;
536.
537.           delete bits;
538.         }
539.         pSch->SentZigBytes += strlen((char *) str);
540.       }
541.
542.       void Parity7to8(unsigned char str[]) {
543.         int i = 0;
544.         while (str[i] != '\0') {
545.           // Remove the parity bit if it exists
546.           str[i] %= 128;
547.           if (str[i] < 32 || str[i] == 127) str[i] = '_';
548.           i++;
549.         }
550.       }
551.
552.       char *byte_to_binary(int x) {
553.         char *b;
554.         b = new char[9];
555.         for (int i = 7; i >= 0; i--) {
556.           if (x % 2) b[i] = '1';
557.           else b[i] = '0';
558.           x /= 2;
559.         }
560.         b[8] = '\0';
561.         return b;
562.       }
```

gateway.h

```
1.  #pragma once
2.
3.  #include <vector>
4.  using namespace std;
5.
6.  #include "Socket.h"
7.
8.  class Gateway {
9.    public:
10.     Gateway(char ip_adres[32], int port_num);
11.     void DestroySocket();
12.     SocketClient *s;
13.     char ip[32];
14.     int port;
15. };
```

gateway.cpp

```
1.  #include "gateway.h"
2.
3.  Gateway::Gateway(char ip_adres[32], int port_num) {
4.    s = NULL;
5.    strcpy(ip, ip_adres);
6.    port = port_num;
7.  }
8.
9.  void Gateway::DestroySocket() {
10.   s->Close();
11.   delete s;
12.   s = NULL;
13. }
```

scheduler.h

```
1.  #pragma once
2.
3.  #include <WinSock2.h>
4.  #include <windows.h>
5.  #include <string>
6.  #include <fstream>
7.  #include <iostream>
8.  using namespace std;
9.
10. #include "gateway.h"
11.
12. class job {
13.   public:
14.     // Constructor
15.     job(char m[256], Gateway *gw, bool per, bool sr, int intv, bool is_meter,
    string mod);
16.     void run(); // Main loop
17.     char message[256]; // Message to send
18.     Gateway *gateway; // Gateway to the module
19.     bool periodic; // Should job be repeated?
20.     bool should_reply; // Do we expect a reply from an end-point?
21.     bool pending; // indicates the message needs to be sent asap
22.     int interval; // Interval between jobs
23.     int count_down; // How long until next job?
24.     bool is_metering; // Is it a metering job?
25.     bool terminate; // Signal job to end
26.     string module; // End-point address
27. };
```

```
28.
29. class active_time {
30.   public:
31.     int hr;
32.     int min;
33.     int sec;
34.     bool operator>(active_time& t2) const;
35.     bool operator>=(active_time& t2) const;
36.     bool operator<(active_time& t2) const;
37.     bool operator<=(active_time& t2) const;
38. };
39.
40. class Scheduler {
41.   public:
42.     Scheduler(); // Constructor
43.     ~Scheduler(); // Destructor
44.     void run(); // The basic loop of the scheduler
45.     void add_job(char message[256], Gateway *gateway, bool periodic, bool
    should_reply, int interval, bool metering = false, string module = ""); // Add a
    job
46.     bool rm_job(int n); // Remove a job
47.     void print_jobs(); // Print info about jobs
48.     void show_comm(); // Print communication on the screen
49.     void hide_comm(); // Don't print communication on the screen
50.     bool get_show(); // Should communication be printed?
51.     int get_max_job_length(); // Maximum length of a job message
52.     void rearrange(); // Rearrange jobs with similar intervals to spread them out
53.     unsigned char CalculateCrc(unsigned char pData[]); // Checksum calculation
54.     void DestroyAllSockets(); // Cleanup
55.     void SetActiveTimes(const char *start_time, const char *end_time); // Set
    when communication ends / starts
56.     bool InActiveTime(); // Are we in the activity time?
57.
58.     job *seljob; // The selected job
59.
60.     vector <job*> jobs; // Vector of all the jobs
61.     // statistics for all messages (including metering messages)
62.     long totaltime; // Time we are running
63.     long msg_count; // Messages sent
64.     long msg_fail; // Failed messages
65.     long msg_success; // Successful messages
66.     long connectfail; // Connection failures
67.     long ackfail; // Acknowledgement failed
68.     long replyfail; // Reply failed
69.     long skipped; // Job was skipped
70.     // statistics for metering messages only
71.     long m_totaltime; // see above
72.     long m_job_count; // see above
73.     long m_job_fail; // see above
74.     long m_job_success; // see above
75.     long m_connectfail; // see above
76.     long m_ackfail; // see above
77.     long m_replyfail; // see above
78.     long m_skipped; // see above
79.
80.     long socket_to; // Socket timeout in ms
81.
82.     unsigned long SentBytes; // Total amount of bytes sent
83.     unsigned long RecvBytes; // Total received bytes
84.     unsigned long SentZigBytes; // Bytes sent by the ZigBee
85.     unsigned long RecvZigBytes; // Bytes received by the ZigBee
86.
87.   private:
88.     void add_job(job* newjob); // Add a job
89.
90.     // Send a message
91.     bool SendMsg(char message[], Gateway *to, bool should_reply, char *term_seq =
    NULL);
92.     // Parse a received message
93.     bool ParseMessage(char msg[], char command[], char arg1[], char arg2[], int
    &i);
94.     string GetMod(int mod, char arg[]); // Find a module
95.     void prnt(string out); // Print a message to the screen and log file
96.     bool is_element(vector<int> vec, int val); // Check if exists in vector
```

```
97.     int num_occ(int val); // return number of occurrences of val in jobs
    intervals
98.     bool show; // Print communication to screen?
99.     char logl[1024]; // Line to print to log
100.
101.          active_time start_t; // Time to start
102.          active_time end_t; // Time to end
103.          bool at_set; // Active Time Set?
104.       };
105.
106.       extern CRITICAL_SECTION CriticalSection;
```

scheduler.cpp

```
1.  #include "scheduler.h"
    #include <time.h>
2.
3.  string space(int n);
4.
5.  extern void Parity8to7(unsigned char str[]);
6.  extern void Parity7to8(unsigned char str[]);
7.
8.  DWORD WINAPI StartJobThread(LPVOID pParam) {
9.    job * pJ;
10.   pJ = (job *) pParam;
11.   pJ->run();
12.   return 0;
13. }
14.
15. bool is_num(char c) {
16.   if (c < '0') return false;
17.   if (c > '9') return false;
18.   return true;
19. }
20.
21. job::job(char m[256], Gateway *gw, bool per, bool sr, int intv, bool is_meter,
    string mod) {
22.   strcpy(message, m);
23.   gateway = gw;
24.   periodic = per;
25.   should_reply = sr;
26.   interval = intv;
27.   terminate = false;
28.   if (per) pending = false;
29.   else pending = true;
30.   count_down = 0;
31.   is_metering = is_meter;
32.   module = mod;
33.
34.   // Only start a thread if we need one
35.   if (periodic) CreateThread(NULL, 0, StartJobThread, this, 0, NULL);
36. }
37.
38. void job::run() {
39.   unsigned long loopt;
40.   extern Scheduler *pSch;
41.   extern bool pause;
42.   count_down = interval * 10;
43.   time_t tijd = time(NULL);
44.   tm *local = localtime(&tijd);
45.   long oldsec = local->tm_sec;
46.   long newsec;
47.   // Keep decrementing the count_down until it reaches 0
48.   // then pending is set to tue, and the counter is restarted
49.   while (!terminate) {
50.     Sleep(100);
51.     tijd = time(NULL);
52.   local = localtime(&tijd);
53.     newsec = local->tm_sec;
54.     if (newsec != oldsec) { // See if a second has passed
```

```
55.       oldsec = newsec;
56.       if (!pause && pSch->InActiveTime()) {
57.         count_down -= 10;
58.         if (count_down <= 0) {
59.           if (pending) {
60.             pSch->skipped++;
61.             if (strstr(message, "Transparant")) pSch->m_skipped++;
62.           }
63.           pending = true;
64.           count_down = interval * 10;
65.         }
66.       }
67.     }
68.   }
69. }
70.
71. Scheduler::Scheduler() {
72.   show = false;
73.
74.   at_set = false;
75.
76.   msg_count = 0;
77.   msg_success = 0;
78.   msg_fail = 0;
79.   totaltime = 0;
80.   connectfail = 0;
81.   ackfail = 0;
82.   replyfail = 0;
83.   skipped = 0;
84.
85.   m_job_count = 0;
86.   m_job_success = 0;
87.   m_job_fail = 0;
88.   m_totaltime = 0;
89.   m_connectfail = 0;
90.   m_ackfail = 0;
91.   m_replyfail = 0;
92.   m_skipped = 0;
93.
94.   socket_to = 2000;
95.
96.   SentBytes = 0;
97.   RecvBytes = 0;
98.   SentZigBytes = 0;
99.   RecvZigBytes = 0;
100.      }
101.
102.      Scheduler::~Scheduler() {
103.        DestroyAllSockets();
104.      }
105.
106.      void Scheduler::show_comm() {
107.        show = true;
108.      }
109.
110.      void Scheduler::hide_comm() {
111.        show = false;
112.      }
113.
114.      bool Scheduler::get_show() {
115.        return show;
116.      }
117.
118.      void Scheduler::run() {
119.        long mts, mte; // track time for jobs
120.        extern bool pause;
121.        bool success;
122.        while (true) {
123.          Sleep(100);
124.          EnterCriticalSection(&CriticalSection);
125.          if (!pause && InActiveTime()) {
126.            for (int i = 0; i < jobs.size(); i++) {
127.              seljob = jobs[i];
128.              success = false;
```

```
129.               if (jobs[i]->pending) {
130.                 // We need to send the message
131.                 jobs[i]->pending = false;
132.
133.                 // See if it's a metering job
134.                 if (strstr(jobs[i]->message, "Transparant")) {
135.                   m_job_count++;
136.                   prnt("\n");
137.                   prnt("------------------------------ METERING JOB -----------
    -------------------");
138.                   prnt("\n");
139.
140.                   char *ss = new char[256];
141.                   unsigned char *us = new unsigned char[256];
142.                   char nmsg[256];
143.
144.                   // First Message: addressing
145.                   mts = GetTickCount();
146.                   if (SendMsg(jobs[i]->message, jobs[i]->gateway, jobs[i]-
    >should_reply)) {
147.
148.                     // Second message: baudrate
149.                     sprintf(ss, "%c051%c%c", 0x06, 0x0d, 0x0a);
150.                     for (int j = 0; j < 256; j++) us[j] = ss[j];
151.                     Parity8to7(us);
152.                     sprintf(nmsg, "##Transparant&%s&%s$", jobs[i]-
    >module.c_str(), us);
153.                     if (SendMsg(nmsg, jobs[i]->gateway, true)) {
154.
155.                       // Third message: password
156.                       sprintf(ss, "%cP1%c(00000000)%ca", 0x01, 0x02, 0x03);
157.                       for (int k = 0; k < 256; k++) us[k] = ss[k];
158.                       Parity8to7(us);
159.                       sprintf(nmsg, "##Transparant&%s&%s$", jobs[i]-
    >module.c_str(), us);
160.                       if (SendMsg(nmsg, jobs[i]->gateway, true)) {
161.
162.                         // Fourth message: Read Data
163.                         //sprintf(ss, "%cR2%cC003()%c%c", 0x01, 0x02, 0x03,
    0x10); // Ask time
164.
165.                         // Generate string for times. THIS FAILS AROUND
    MIDNIGHT!!!
166.                         int h1, h2, m1, m2;
167.                         time_t tijd = time(NULL);
168.                         tm *local = localtime(&tijd);
169.                         h2 = local->tm_hour;
170.                         h1 = local->tm_hour;
171.                         m2 = (local->tm_min / 5) * 5; // Floor to a fivefold
172.                         m1 = m2 - 5;
173.                         if (m1 < 0) {
174.                           h1--;
175.                           m1 += 60;
176.                         }
177.                         char timestr[32];
178.                         sprintf(timestr,
    "0%02d%02d%02d%02d%02d;0%02d%02d%02d%02d%02d",
179.                           local->tm_year % 100,       /* 1 year */
180.                           local->tm_mon + 1,          /* 1 month */
181.                           local->tm_mday,             /* 1 day */
182.                           h1,                         /* 1 hour */
183.                           m1,                         /* 1 minutes */
184.                           local->tm_year % 100,       /* 2 year */
185.                           local->tm_mon + 1,          /* 2 month */
186.                           local->tm_mday,             /* 2 day */
187.                           h2,                         /* 2 hour */
188.                           m2                          /* 2 minutes */
189.                           );
190.                         sprintf(ss, "%cR5%cP.01(%s)%c", 0x01, 0x02, timestr,
    0x03); // Metering data
191.                         for (int j = 0; j < 256; j++) us[j] = ss[j];
192.                         char crc = CalculateCrc(us);
193.                         int cnull = 0;
```

```
194.                          while (us[cnull] != '\0') cnull++; // Find the end of
    the string
195.                          us[cnull] = crc;
196.                          us[cnull+1] = '\0';
197.                          Parity8to7(us);
198.
199.                          sprintf(nmsg, "##Transparant&%s&%s$", jobs[i]-
    >module.c_str(), us);
200.                          char terms[2] = {3, 0}; // Carriage return, End of
    text, Null
201.                          if (SendMsg(nmsg, jobs[i]->gateway, true, terms))
    success = true;
202.                        } // M3
203.                      } // M2
204.
205.                      // Fifth message: There is an open connection, so we close
    it
206.                      sprintf(ss, "%cB0%c%c", 0x01, 0x03, 0x71);
207.                      for (int j = 0; j < 256; j++) us[j] = ss[j];
208.                      Parity8to7(us);
209.                      sprintf(nmsg, "##Transparant&%s&%s$", jobs[i]-
    >module.c_str(), us);
210.                      SendMsg(nmsg, jobs[i]->gateway, false); // Will not reply
211.
212.                      if (success) {
213.                        // Save the elapsed time if the message was successful
214.                        mte = GetTickCount();
215.                      }
216.                    } // M1
217.
218.                    delete ss;
219.                    delete us;
220.                    ss = NULL;
221.                    us = NULL;
222.                    if (success) {
223.                      m_job_success++;
224.                      m_totaltime += mte - mts;
225.                      sprintf(logl, "Total time of job: %ld ms\n", mte - mts);
226.                      prnt(logl);
227.                    }
228.                    else m_job_fail++;
229.                  } // if Transparant
230.                  else {
231.                    prnt("\n");
232.                    prnt("------------------------------------------------------------
    --------------------");
233.                    prnt("\n");
234.                    SendMsg(jobs[i]->message, jobs[i]->gateway, jobs[i]-
    >should_reply);
235.                  } // else: no metering job
236.                  // If not periodic, erase the job
237.                  if (!jobs[i]->periodic) rm_job(i);
238.                }
239.              } // for loop
240.            } // if pause
241.            LeaveCriticalSection(&CriticalSection);
242.          } // while
243.        } // void run
244.
245.        void Scheduler::add_job(job* newjob) {
246.          jobs.push_back(newjob);
247.        }
248.
249.        void Scheduler::add_job(char message[], Gateway *gateway, bool periodic,
    bool should_reply, int interval, bool metering, string module) {
250.          job *j = new job(message, gateway, periodic, should_reply, interval,
    metering, module);
251.          add_job(j);
252.        }
253.
254.        bool Scheduler::rm_job(int n) {
255.          if (n < 0 || n >= jobs.size()) return false;
256.          // Tell the thread to terminate
257.          jobs[n]->terminate = true;
```

```
258.
259.         // Wait until we're sure the thread has terminated
260.         Sleep(200);
261.
262.         // Delete the job instance from the vector
263.         jobs.erase(jobs.begin() + n);
264.         return true;
265.       }
266.
267.       void Scheduler::print_jobs() {
268.         int j;
269.         if (jobs.size()) {
270.           int maxlength = get_max_job_length();
271.           cout << endl;
272.           cout << "+-----+-"; for(j=0;j<maxlength+1;j++)cout<<"-"; cout << "+--
    ---------------+-------+--------------+-------+" << endl;
273.           cout << "|   # | Message" << space(maxlength - 7).c_str() << " | IP
    address     "  << " | Intv. " << "| Next trigger | Reply |" << endl;
274.           cout << "+-----+-"; for(j=0;j<maxlength+1;j++)cout<<"-"; cout << "+--
    ---------------+-------+--------------+-------+" << endl;
275.           for (int i = 0; i < jobs.size(); i++) {
276.             char msg[256];
277.             // If it is a metering message, we do not print the content of
    jobs[i]->message because of linefeeds etc
278.             if (strstr(jobs[i]->message, "Transparant")) sprintf(msg,
    "<Metering Message>");
279.             else strcpy(msg, jobs[i]->message);
280.             if (jobs[i]->periodic) {
281.               printf("| %3d | %s%s | %15s | %5d | %12d |",
282.                 i,
283.                 msg,
284.                 space(maxlength-strlen(msg)).c_str(),
285.                 jobs[i]->gateway->ip,
286.                 jobs[i]->interval,
287.                 jobs[i]->count_down / 10);
288.             }
289.             else {
290.               printf("| %3d | %s%s | %15s |     - |            - |",
291.                 i,
292.                 msg,
293.                 space(maxlength-strlen(msg)).c_str(),
294.                 jobs[i]->gateway->ip);
295.             }
296.             if (jobs[i]->should_reply) cout << "   YES |";
297.             else cout << "    NO |";
298.             cout << endl;
299.           }
300.           cout << "+-----+-"; for(j=0;j<maxlength+1;j++)cout<<"-"; cout << "+--
    ---------------+-------+--------------+-------+" << endl;
301.         }
302.         else cout << "No jobs!" << endl;
303.       }
304.
305.       int Scheduler::get_max_job_length() {
306.         int max = 0;
307.         for (int i = 0; i < jobs.size(); i++) {
308.           int len = strlen(jobs[i]->message);
309.           if (len > max) max = len;
310.         }
311.         return max;
312.       }
313.
314.       string space(int n) {
315.         string s = "";
316.         for (int i = 0; i < n; i++) s += " ";
317.         return s;
318.       }
319.
320.       bool Scheduler::SendMsg(char message[], Gateway *to, bool should_reply,
    char *term_seq) {
321.         long te, ts;
322.         char recv[4096];
323.         char c[1024];
324.         char a1[1024];
```

```
325.          char a2[2048];
326.          int start;
327.          bool success = false;
328.          bool retval = true;
329.          bool retry = true;
330.
331.          bool meterjob = strstr(message, "Transparant");
332.
333.          msg_count++;
334.
335.        lbl_retry:
336.          try {
337.            if (to->s == NULL) {
338.              // Create a new socket
339.              to->s = new SocketClient(to->ip, to->port);
340.            }
341.
342.            // Connect
343.            int ret_val = to->s->Connect();
344.            if (!ret_val) {
345.              sprintf(logl, "Connected to %s on port %d", to->ip, to->port);
346.              prnt(logl);
347.            }
348.
349.            if (retry) sprintf(logl, "Sending message: %s", message);
350.            else sprintf(logl, "Retry to send message: %s", message);
351.            Parity7to8((unsigned char*) logl);
352.            prnt("\n");
353.            prnt(logl);
354.            prnt("\n");
355.
356.            ts = GetTickCount();
357.            to->s->SendLine(message);
358.            SentBytes += strlen(message);
359.
360.            // Wait for the ACK
361.            string l = "";
362.            success = to->s->ReceiveLine(l);
363.            if (!should_reply) te = GetTickCount();
364.            if (success) {
365.              strcpy(recv, l.c_str());
366.
367.              // parse the message we received
368.              start = 0;
369.              ParseMessage(recv, c, a1, a2, start);
370.
371.              unsigned char *us = new unsigned char[1024];
372.              for (int i = 0; i < 1024; i++) us[i] = a2[i];
373.
374.              Parity7to8(us);
375.              if (strcmp(a2, "")) sprintf(logl, "%s, %s, %s\n", c, a1, us);
376.              else sprintf(logl, "%s, %s\n", c, a1);
377.              prnt(logl);
378.
379.              if (strstr(message, "Geef_Modules")) {
380.                extern vector<string> modules;
381.                // We gaan de gevonden modules extracten
382.                ofstream modfile;
383.                int Nmodules = (strlen(a2) + 1) / 19; // Een adres is 18
    karakters, en een ampersand
384.                if (Nmodules) {
385.                  modfile.open("modules.txt", fstream::app);
386.                }
387.                if (modfile.is_open()) {
388.
389.                  for (int m = 0; m < Nmodules; m++) {
390.                    string adr = GetMod(m, a2);
391.                    bool newmod = true;
392.
393.                    for (int i = 0; i < modules.size(); i++)
394.                      if (modules[i] == adr) newmod = false;
395.
396.                    if (newmod){
397.                      // The module is new: add it to the modules.txt file
```

```
398.                     modfile.seekp(0, ios::end);
399.                     modfile.put('\n');
400.                     modfile << adr.c_str();
401.                     // Also add to vector
402.                     modules.push_back(adr);
403.                   }
404.                 }
405.                 modfile.close();
406.               }
407.             }
408.
409.             // Wait for reply
410.             if (should_reply) {
411.               success = false;
412.               vector<string> recvec;
413.               string l = "";
414.               success = to->s->ReceiveLine(l);
415.               recvec.push_back(l);
416.               if (term_seq) {
417.                 // Keep receiving until we receive the terminating sequence
418.                 while (success && !strstr(l.c_str(), term_seq)) {
419.                   l = "";
420.                   success = to->s->ReceiveLine(l);
421.                   recvec.push_back(l);
422.                 }
423.               }
424.               if (success) {
425.                 te = GetTickCount();
426.                 for (int i = 0; i < recvec.size(); i++) {
427.                   l = recvec[i];
428.                   strcpy(recv, l.c_str());
429.
430.                   // Parse the reply
431.                   start = 0;
432.                   ParseMessage(recv, c, a1, a2, start);
433.
434.                   if (strcmp(a2, "")) {
435.                     if (meterjob) {
436.                       unsigned char *ua2 = new unsigned char[2048];
437.                       for (int i = 0; i < 2048; i++) ua2[i] = a2[i];
438.                       Parity7to8(ua2);
439.                       sprintf(logl, "%s, %s, %s\n", c, a1, ua2);
440.                     }
441.                     else sprintf(logl, "%s, %s, %s\n", c, a1, a2);
442.                   }
443.                   else sprintf(logl, "%s, %s\n", c, a1);
444.                   prnt(logl);
445.                   if (show) cout.flush();
446.                 }
447.                 sprintf(logl, "Received after %ld ms\n", te - ts);
448.                 prnt(logl);
449.
450.                 totaltime += te - ts;
451.               }
452.               else {
453.                 replyfail++;
454.                 if (meterjob) m_replyfail++;
455.                 sprintf(logl, "Failed: no reply received!\n");
456.                 prnt(logl);
457.               }
458.             }
459.             else {
460.               if (show) cout.flush();
461.               sprintf(logl, "Received after %ld ms\n", te - ts);
462.               prnt(logl);
463.             }
464.           }
465.           else {
466.             if (retry) {
467.               retry = false;
468.               delete to->s;
469.               to->s = NULL;
470.               goto lbl_retry;
471.             }
```

```
472.             ackfail++;
473.             if (meterjob) m_ackfail++;
474.             sprintf(logl, "Failed: no ack received!\n");
475.             prnt(logl);
476.           }
477.         }
478.         catch (const char* s) {
479.           sprintf(logl, "Failed: '%s'\n", s);
480.           prnt(logl);
481.           connectfail++;
482.           if (meterjob) m_connectfail++;
483.           retval = false;
484.           to->DestroySocket();
485.         }
486.         catch (std::string s) {
487.           sprintf(logl, "Failed: '%s'\n", s.c_str());
488.           prnt(logl);
489.           connectfail++;
490.           if (meterjob) m_connectfail++;
491.           retval = false;
492.           to->DestroySocket();
493.         }
494.         catch (int i) {
495.           sprintf(logl, "Failed: errorcode %d\n", i);
496.           prnt(logl);
497.           connectfail++;
498.           if (meterjob) m_connectfail++;
499.           retval = false;
500.           to->DestroySocket();
501.         }
502.         catch (...) {
503.           sprintf(logl, "Failed: unhandled exception\n");
504.           prnt(logl);
505.           connectfail++;
506.           if (meterjob) m_connectfail++;
507.           retval = false;
508.           to->DestroySocket();
509.         }
510.         /*if (disconnect) {
511.           to->s->Close();
512.           prnt("Disconnected!\n");
513.           delete s;
514.         }*/
515.
516.         if(!success) msg_fail++;
517.         else msg_success++;
518.
519.         return success;
520.       }
521.
522.       string Scheduler::GetMod(int mod, char arg[]) {
523.              // Get the XBee address of the 'mod'-th module in arg
524.              string m;
525.              m = "";
526.              for (int i = mod * 19; i < mod * 19 + 18; i++) {
527.                     m += arg[i];
528.              }
529.              return m;
530.       }
531.       //------------------------------------------------------------------------
    ----
532.
533.
534.       void Scheduler::prnt(string out) {
535.         time_t nu = time(NULL);
536.              tm *local = localtime(&nu);
537.         if (out != "\n") {
538.                char t[32];
539.                strftime(t, 32, "[%x - %X] ", local);
540.           string time = t;
541.           out = time + out;
542.         }
543.
544.         ofstream log;
```

```
545.          if (show) cout << out;
546.          char filename[32];
547.          strftime(filename, 32, "log/log %Y-%m-%d.txt", local);
548.          log.open(filename, ios::app);
549.          if (!log && show) cout << "Cannot open log file" << endl;
550.          else log << out;
551.        }
552.
553.        bool Scheduler::ParseMessage(char msg[], char command[], char arg1[],
    char arg2[],   int &i) {
554.                // Hier gaan we een string zoals hij over ethernet binnenkomt,
555.                // parsen en stoppen hem in command, arg1 en arg2
556.
557.          // Eerst tellen we de ontvangen bytes op
558.          RecvBytes += strlen(msg);
559.
560.                int index;
561.                while (msg[i] != '#' && i < 4096) i++; // Stop if i becomes too
    large
562.                if (msg[i] == '#' && msg[i+1] == '#') {
563.                        index = 0;
564.                        // Zoek het eerste scheidingsteken en plaats alles ervoor
    in command
565.                        for (i += 2; msg[i] != '&' && msg[i] != '$'; i++) {
566.                                // Loop tot we & of $ vinden
567.                                if (msg[i] == '!') {i++; index++;} // Escape
568.                                command[index] = msg[i];
569.                                index++;
570.                        }
571.                        command[index] = '\0';
572.                        if (msg[i] == '&') {
573.                                // Er is een argument
574.                                index = 0;
575.                                for (i++; msg[i] != '&' && msg[i] != '$'; i++) {
576.                                        // Loop tot we & of $ vinden
577.                                        if (msg[i] == '!') {i++; index++;} // Escape
578.                                        arg1[index] = msg[i];
579.                                        index++;
580.                                }
581.                                arg1[index] = '\0';
582.                                if (msg[i] == '&') {
583.                                        // Er is nog een argument
584.                                        index = 0;
585.                                        for (i++; msg[i] != '$'; i++) {
586.                                                if (msg[i] == '!') {i++; index++;}
    // Escape
587.                                                // Loop tot we $ vinden
588.                                                arg2[index] = msg[i];
589.                                                index++;
590.                                        }
591.                                        arg2[index] = '\0';
592.                                }
593.                                else arg2[0] = '\0';
594.                        }
595.                        else arg1[0] = '\0';
596.
597.           // Register sent and received bytes
598.           if (!strcmp(command, "Transparant")) {
599.           RecvZigBytes += strlen(arg2);
600.           }
601.           else if (strcmp(command, "ACK")){
602.             RecvZigBytes += 1 + 2 + 2 + 2 + 4 + 4;
603.             SentZigBytes += 1 + 2 + 2 + 2 + 4 + 4;
604.           }
605.               }
606.               else {
607.                       return false;
608.               }
609.               return true;
610.        }
611.
612.        void Scheduler::rearrange() {
613.          int i;
614.          vector<int> exist_intv;
```

```
615.
616.          // First we check which intervals are present in the jobtable
617.          for (i = 0; i < jobs.size(); i++) {
618.            if (!is_element(exist_intv, jobs[i]->interval)) {
619.              exist_intv.push_back(jobs[i]->interval);
620.              cout << "Added value to vector: " << jobs[i]->interval << endl;
621.            }
622.          }
623.
624.          cout << "Arranging..." << endl;
625.
626.          // Here we start looping over the different intervals
627.          for (i = 0; i < exist_intv.size(); i++) {
628.            cout << "Arranging jobs with interval " << exist_intv[i] << endl;
629.
630.            // Calculate the number of jobs with this interval
631.            int njobs = num_occ(exist_intv[i]);
632.            cout << "This interval has " << njobs << " occurrences" << endl;
633.
634.            // Reassign the countdown value for all jobs with this interval
635.            int job_idx = 0;
636.            for (int j = 0; j < njobs; j++) {
637.
638.              // Find the next job with this interval
639.              while (jobs[job_idx]->interval != exist_intv[i]) job_idx++;
640.
641.              // Reassign countdown value
642.              jobs[job_idx]->count_down = (10 * exist_intv[i] * (j + 1)) / njobs;
643.              cout << "Reassigned countdown to " << jobs[job_idx]->count_down /
    10 << " for job " << job_idx << endl;
644.              job_idx++;
645.            }
646.          }
647.        }
648.
649.        bool Scheduler::is_element(vector<int> vec, int val) {
650.          for (int i = 0; i < vec.size(); i++) {
651.            if (vec[i] == val) return true;
652.          }
653.          return false;
654.        }
655.
656.        int Scheduler::num_occ(int val) {
657.          int retv = 0;
658.          for (int i = 0; i < jobs.size(); i++) {
659.            if (jobs[i]->interval == val) retv++;
660.          }
661.          return retv;
662.        }
663.
664.        unsigned char Scheduler::CalculateCrc(unsigned char pData[]) {
665.                // Calculate the BCC of the specified data block
666.          int idx = 0;
667.
668.          // Skip Start of Heading and Start of Text
669.          while ((pData[idx] == 1 || pData[idx] == 2) && idx < 9999) idx++;
670.
671.          unsigned char cCrc = pData[idx];
672.          idx++;
673.
674.          // Loop until the string is terminated by '\0'
675.          while (pData[idx] != 0 && idx < 9999) {
676.                cCrc ^= pData[idx];
677.            idx++;
678.          }
679.          return cCrc;
680.        }
681.
682.        void Scheduler::DestroyAllSockets() {
683.          extern vector<Gateway*> gateways;
684.          for (int i = 0; i < gateways.size(); i++) gateways[i]->DestroySocket();
685.        }
686.
```

```
687.       void Scheduler::SetActiveTimes(const char *start_time, const char
    *end_time) {
688.         int h1, h2, m1, m2, s1, s2;
689.         // First some error checks
690.         if (!start_time || !end_time) cout << "Start/End times cannot be set:
    NULL string" << endl;
691.         else if (strlen(start_time) != strlen(end_time)) cout << "Start/End
    times cannot be set: strings of different length" << endl;
692.         else if (strlen(start_time) != 5 && strlen(start_time) != 8) cout <<
    "Start/End times cannot be set: incompatible string" << endl;
693.
694.         // Hours and minutes specified
695.         else if (strlen(start_time) == 5) {
696.           if (!is_num(start_time[0]) || !is_num(start_time[1]) ||
    !is_num(start_time[3]) || !is_num(start_time[4]) ||
697.                 !is_num(end_time[0]) ||   !is_num(end_time[1]) ||
    !is_num(end_time[3]) ||   !is_num(end_time[4]))
698.             cout << "Start/End times cannot be set: strings should follow
    format 'hh:mm' or 'hh:mm:ss'" << endl;
699.           else {
700.             // convert the string to times
701.             h1 = 10 * (start_time[0] - '0') + (start_time[1] - '0');
702.             h2 = 10 * (end_time[0] - '0')   + (end_time[1] - '0');
703.             m1 = 10 * (start_time[3] - '0') + (start_time[4] - '0');
704.             m2 = 10 * (end_time[3] - '0')   + (end_time[4] - '0');
705.
706.             // Two more checks
707.             if (h1 > 23 || h2 > 23 || m1 > 59 || m2 > 59) {
708.               cout << "Start/End times cannot be set: times are out of bounds"
    << endl;
709.               return;
710.             }
711.             if (h1 == h2 && m1 == m2) {
712.               cout << "Start/End times cannot be set: times are equal" << endl;
713.               return;
714.             }
715.
716.             // We passed all checks: set the active times
717.             start_t.hr = h1;
718.             start_t.min = m1;
719.             start_t.sec = 0;
720.             end_t.hr = h2;
721.             end_t.min = m2;
722.             end_t.sec = 0;
723.             at_set = true;
724.
725.             cout << "Start and end times successfully set to '" <<
726.               start_t.hr << ":" << start_t.min << ":" << start_t.sec << "' and
    '" <<
727.               end_t.hr << ":" << end_t.min << ":" << end_t.sec << "'" << endl;
728.           }
729.         }
730.
731.         // Hours, minutes and seconds specified
732.         else if (strlen(start_time) == 8) {
733.           if (!is_num(start_time[0]) || !is_num(start_time[1]) ||
    !is_num(start_time[3]) || !is_num(start_time[4]) ||
734.                 !is_num(end_time[0]) ||   !is_num(end_time[1]) ||
    !is_num(end_time[3]) ||   !is_num(end_time[4]) ||
735.                 !is_num(end_time[6]) ||   !is_num(end_time[7]) ||
    !is_num(start_time[6]) ||   !is_num(start_time[7]))
736.             cout << "Start/End times cannot be set: strings should follow
    format 'hh:mm' or 'hh:mm:ss'" << endl;
737.           else {
738.             // convert the string to times
739.             h1 = 10 * (start_time[0] - '0') + (start_time[1] - '0');
740.             h2 = 10 * (end_time[0] - '0')   + (end_time[1] - '0');
741.             m1 = 10 * (start_time[3] - '0') + (start_time[4] - '0');
742.             m2 = 10 * (end_time[3] - '0')   + (end_time[4] - '0');
743.             s1 = 10 * (start_time[6] - '0') + (start_time[7] - '0');
744.             s2 = 10 * (end_time[6] - '0') + (end_time[7] - '0');
745.
746.             // Two more checks
```

```
747.             if (h1 > 23 || h2 > 23 || m1 > 59 || m2 > 59 || s1 > 59 || s2 > 59)
    {
748.               cout << "Start/End times cannot be set: times are out of bounds"
    << endl;
749.               return;
750.             }
751.             if (h1 == h2 && m1 == m2 && s1 == s2) {
752.               cout << "Start/End times cannot be set: times are equal" << endl;
753.               return;
754.             }
755.
756.             // We passed all checks: set the active times
757.             start_t.hr = h1;
758.             start_t.min = m1;
759.             start_t.sec = s1;
760.             end_t.hr = h2;
761.             end_t.min = m2;
762.             end_t.sec = s2;
763.             at_set = true;
764.
765.             printf("Start and end times successfully set to '%02d:%02d:%02d'
    and '%02d:%02d:%02d'\n",
766.                 start_t.hr, start_t.min, start_t.sec, end_t.hr, end_t.min,
    end_t.sec);
767.           }
768.         }
769.       }
770.
771.       bool Scheduler::InActiveTime() {
772.         // Activetime not set: always true
773.         if (!at_set) return true;
774.
775.         time_t t = time(NULL);
776.         tm *local = localtime(&t);
777.         active_time now;
778.         now.hr = local->tm_hour;
779.         now.min = local->tm_min;
780.         now.sec = local->tm_sec;
781.
782.         if (end_t < start_t) {
783.           // De eindtijd ligt VOOR de starttijd
784.           if (now >= start_t) return true; // Het is na de begintijd (of
    gelijk)
785.           if (now < end_t) return true; // Het is voor de eindtijd
786.         }
787.         else {
788.           // De eindtijd ligt NA de starttijd
789.           if (start_t <= now && now < end_t) return true;
790.         }
791.         return false;
792.       }
793.
794.       bool active_time::operator>(active_time& t2) const
795.       {
796.         if (hr > t2.hr) return true;
797.         if (hr == t2.hr) {
798.           if (min > t2.min) return true;
799.           if (min == t2.min && sec > t2.sec) return true;
800.         }
801.         return false;
802.       }
803.
804.       bool active_time::operator>=(active_time& t2) const
805.       {
806.         if (hr > t2.hr) return true;
807.         if (hr == t2.hr) {
808.           if (min > t2.min) return true;
809.           if (min == t2.min && sec >= t2.sec) return true;
810.         }
811.         return false;
812.       }
813.
814.       bool active_time::operator<(active_time& t2) const
815.       {
```

```
816.         if (hr < t2.hr) return true;
817.         if (hr == t2.hr) {
818.           if (min < t2.min) return true;
819.           if (min == t2.min && sec < t2.sec) return true;
820.         }
821.         return false;
822.       }
823.
824.       bool active_time::operator<=(active_time& t2) const
825.       {
826.         if (hr < t2.hr) return true;
827.         if (hr == t2.hr) {
828.           if (min < t2.min) return true;
829.           if (min == t2.min && sec <= t2.sec) return true;
830.         }
831.         return false;
832.       }
```

Socket.h

The following code was completely written by René Nyffenegger, and was not altered for our program.

```
1.  /*
2.     Socket.h
3.
4.     Copyright (C) 2002-2004 René Nyffenegger
5.
6.     This source code is provided 'as-is', without any express or implied
7.     warranty. In no event will the author be held liable for any damages
8.     arising from the use of this software.
9.
10.    Permission is granted to anyone to use this software for any purpose,
11.    including commercial applications, and to alter it and redistribute it
12.    freely, subject to the following restrictions:
13.
14.    1. The origin of this source code must not be misrepresented; you must not
15.       claim that you wrote the original source code. If you use this source code
16.       in a product, an acknowledgment in the product documentation would be
17.       appreciated but is not required.
18.
19.    2. Altered source versions must be plainly marked as such, and must not be
20.       misrepresented as being the original source code.
21.
22.    3. This notice may not be removed or altered from any source distribution.
23.
24.    René Nyffenegger rene.nyffenegger@adp-gmbh.ch
25. */
26.
27. #pragma once
28.
29. #ifndef SOCKET_H
30. #define SOCKET_H
31.
32. #include <WinSock2.h>
33. #include <string>
34. #include <iostream>
35. using namespace std;
36.
37. enum TypeSocket {BlockingSocket, NonBlockingSocket};
38.
39. class Socket {
40. public:
41.
42.   virtual ~Socket();
43.   Socket(const Socket&);
44.   Socket& operator=(Socket&);
```

```
45.
46.   bool ReceiveLine(string &s_out);
47.   string ReceiveBytes();
48.
49.   void   Close();
50.
51.   // The parameter of SendLine is not a const reference
52.   // because SendLine modifes the std::string passed.
53.   void   SendLine (string);
54.
55.   // The parameter of SendBytes is a const reference
56.   // because SendBytes does not modify the std::string passed
57.   // (in contrast to SendLine).
58.   void   SendBytes(const string&);
59.
60.   unsigned long SetTimeout(unsigned long t_ms);
61.
62. protected:
63.   friend class SocketServer;
64.   friend class SocketSelect;
65.
66.   Socket(SOCKET s);
67.   Socket();
68.
69.
70.   SOCKET s_;
71.
72.   int* refCounter_;
73.
74.   unsigned long timeout;
75.
76. private:
77.   static void Start();
78.   static void End();
79.   static int  nofSockets_;
80. };
81.
82. class SocketClient : public Socket {
83. public:
84.   SocketClient(const string& host, int port);
85.   int Connect();
86. private:
87.   sockaddr_in addr;
88.   std::string error;
89. };
90.
91. class SocketServer : public Socket {
92. public:
93.   SocketServer(int port, int connections, TypeSocket type=BlockingSocket);
94.
95.   Socket* Accept();
96. };
97.
98. // http://msdn.microsoft.com/library/default.asp?url=/library/en-
    us/winsock/wsapiref_2tiq.asp
99. class SocketSelect {
100.         public:
101.           SocketSelect(Socket const * const s1, Socket const * const s2=NULL,
    TypeSocket type=BlockingSocket);
102.
103.           bool Readable(Socket const * const s);
104.
105.         private:
106.           fd_set fds_;
107.       };
108.
109.
110.
111.       #endif
```

Socket.cpp

The following code was completely written by René Nyffenegger. We only altered the character in `ReceiveLine()` that represents the end of a line. This originally is a linefeed (`'\n'`), but the gateway protocol messages end with a dollar sign (`'$'`), so we changed this in line 154.

```
1.  /*
2.     Socket.cpp
3.
4.     Copyright (C) 2002-2004 René Nyffenegger
5.
6.     This source code is provided 'as-is', without any express or implied
7.     warranty. In no event will the author be held liable for any damages
8.     arising from the use of this software.
9.
10.    Permission is granted to anyone to use this software for any purpose,
11.    including commercial applications, and to alter it and redistribute it
12.    freely, subject to the following restrictions:
13.
14.    1. The origin of this source code must not be misrepresented; you must not
15.       claim that you wrote the original source code. If you use this source code
16.       in a product, an acknowledgment in the product documentation would be
17.       appreciated but is not required.
18.
19.    2. Altered source versions must be plainly marked as such, and must not be
20.       misrepresented as being the original source code.
21.
22.    3. This notice may not be removed or altered from any source distribution.
23.
24.    René Nyffenegger rene.nyffenegger@adp-gmbh.ch
25. */
26.
27.
28. #include "Socket.h"
29. #include <WinSock2.h>
30.
31. int Socket::nofSockets_= 0;
32.
33. void Socket::Start() {
34.   if (!nofSockets_) {
35.     WSADATA info;
36.     if (WSAStartup(MAKEWORD(2,0), &info)) {
37.       throw "Could not start WSA";
38.     }
39.   }
40.   ++nofSockets_;
41. }
42.
43. void Socket::End() {
44.   WSACleanup();
45. }
46.
47. Socket::Socket() : s_(0) {
48.   Start();
49.   // UDP: use SOCK_DGRAM instead of SOCK_STREAM
50.   s_ = socket(AF_INET,SOCK_STREAM,0);
51.
52.   if (s_ == INVALID_SOCKET) {
53.     throw "INVALID_SOCKET";
54.   }
55.
56.   refCounter_ = new int(1);
57.
58.   timeout = 2000;
59. }
60.
61. Socket::Socket(SOCKET s) : s_(s) {
62.   Start();
```

```
63.   refCounter_ = new int(1);
64. };
65.
66. Socket::~Socket() {
67.   if (! --(*refCounter_)) {
68.     Close();
69.     delete refCounter_;
70.   }
71.
72.   --nofSockets_;
73.   if (!nofSockets_) End();
74. }
75.
76. Socket::Socket(const Socket& o) {
77.   refCounter_=o.refCounter_;
78.   (*refCounter_)++;
79.   s_         =o.s_;
80.
81.   nofSockets_++;
82. }
83.
84. Socket& Socket::operator=(Socket& o) {
85.   (*o.refCounter_)++;
86.
87.   refCounter_=o.refCounter_;
88.   s_         =o.s_;
89.
90.   nofSockets_++;
91.
92.   return *this;
93. }
94.
95. void Socket::Close() {
96.   try {
97.     closesocket(s_);
98.   }
99.   catch (...) {;}
100.       }
101.
102.       std::string Socket::ReceiveBytes() {
103.         std::string ret;
104.         char buf[1024];
105.
106.         while (1) {
107.           u_long arg = 0;
108.           if (ioctlsocket(s_, FIONREAD, &arg) != 0)
109.             break;
110.
111.           if (arg == 0)
112.             break;
113.
114.           if (arg > 1024) arg = 1024;
115.
116.           int rv = recv (s_, buf, arg, 0);
117.           if (rv <= 0) break;
118.
119.           std::string t;
120.
121.           t.assign (buf, rv);
122.           ret += t;
123.         }
124.
125.         return ret;
126.       }
127.
128.       bool Socket::ReceiveLine(std::string &s_out) {
129.         s_out = "";
130.         struct timeval tv;
131.         tv.tv_sec = timeout;
132.         if (setsockopt(s_, SOL_SOCKET, SO_RCVTIMEO,(char *)&tv,sizeof(struct
    timeval)))
133.           cerr << "Couldn't set socket timeout" << endl;
134.         while (1) {
135.           char r;
```

```
136.
137.           switch(recv(s_, &r, 1, 0)) {
138.             case 0: // not connected anymore;
139.                     // ... but last line sent
140.                     // might not end in \n,
141.                     // so return ret anyway.
142.               return false;
143.             case -1:
144.               return false;
145.       //      if (errno == EAGAIN) {
146.       //        return ret;
147.       //      } else {
148.       //      // not connected anymore
149.       //      return "";
150.       //      }
151.           }
152.
153.           s_out += r;
154.           if (r == '$') { // Terminating character is dollar sign ($)
155.             return true; // Terminating character changed to '$'
156.           }
157.         }
158.       }
159.
160.       void Socket::SendLine(std::string s) {
161.         s += '\n';
162.         send(s_,s.c_str(),s.length(),0);
163.       }
164.
165.       void Socket::SendBytes(const std::string& s) {
166.         send(s_,s.c_str(),s.length(),0);
167.       }
168.
169.       SocketServer::SocketServer(int port, int connections, TypeSocket type) {
170.         sockaddr_in sa;
171.
172.         memset(&sa, 0, sizeof(sa));
173.
174.         sa.sin_family = PF_INET;
175.         sa.sin_port = htons(port);
176.         s_ = socket(AF_INET, SOCK_STREAM, 0);
177.         if (s_ == INVALID_SOCKET) {
178.           throw "INVALID_SOCKET";
179.         }
180.
181.         if(type==NonBlockingSocket) {
182.           u_long arg = 1;
183.           ioctlsocket(s_, FIONBIO, &arg);
184.         }
185.
186.         /* bind the socket to the internet address */
187.         if (bind(s_, (sockaddr *)&sa, sizeof(sockaddr_in)) == SOCKET_ERROR) {
188.           closesocket(s_);
189.           throw "INVALID_SOCKET";
190.         }
191.
192.         listen(s_, connections);
193.       }
194.
195.       Socket* SocketServer::Accept() {
196.         SOCKET new_sock = accept(s_, 0, 0);
197.         if (new_sock == INVALID_SOCKET) {
198.           int rc = WSAGetLastError();
199.           if(rc==WSAEWOULDBLOCK) {
200.             return 0; // non-blocking call, no request pending
201.           }
202.           else {
203.             throw "Invalid Socket";
204.           }
205.         }
206.
207.         Socket* r = new Socket(new_sock);
208.         return r;
209.       }
```

```
210.
211.        SocketClient::SocketClient(const std::string& host, int port) : Socket()
    {
212.          hostent *he;
213.          if ((he = gethostbyname(host.c_str())) == 0) {
214.            error = strerror(errno);
215.            throw error;
216.          }
217.
218.          addr.sin_family = AF_INET;
219.          addr.sin_port = htons(port);
220.          addr.sin_addr = *((in_addr *)he->h_addr);
221.          memset(&(addr.sin_zero), 0, 8);
222.        }
223.
224.        int SocketClient::Connect() {
225.          int ret_val = ::connect(s_, (sockaddr *) &addr, sizeof(sockaddr));
226.          if (ret_val) {
227.            // Return value is not 0, but socket might already be connected
228.            int nerr = WSAGetLastError();
229.            if (nerr != WSAEISCONN) {
230.              // Significant error
231.              error = strerror(nerr);
232.              if (error == "Unknown error") throw nerr;
233.              throw error;
234.            }
235.          }
236.
237.          // Set the keep alive option
238.          char optval = 0xFF;
239.          setsockopt(
240.            s_,           // descriptor identifying a socket
241.            SOL_SOCKET,   // level
242.            SO_KEEPALIVE, // optname
243.            &optval,      // input buffer,
244.            sizeof(char)  // size of input buffer
245.          );
246.
247.          return ret_val;
248.        }
249.
250.        SocketSelect::SocketSelect(Socket const * const s1, Socket const * const
    s2, TypeSocket type) {
251.          FD_ZERO(&fds_);
252.          FD_SET(const_cast<Socket*>(s1)->s_,&fds_);
253.          if(s2) {
254.            FD_SET(const_cast<Socket*>(s2)->s_,&fds_);
255.          }
256.
257.          TIMEVAL tval;
258.          tval.tv_sec  = 0;
259.          tval.tv_usec = 1;
260.
261.          TIMEVAL *ptval;
262.          if(type==NonBlockingSocket) {
263.            ptval = &tval;
264.          }
265.          else {
266.            ptval = 0;
267.          }
268.
269.          if (select (0, &fds_, (fd_set*) 0, (fd_set*) 0, ptval) == SOCKET_ERROR)
270.            throw "Error in select";
271.        }
272.
273.        bool SocketSelect::Readable(Socket const* const s) {
274.          if (FD_ISSET(s->s_,&fds_)) return true;
275.          return false;
276.        }
277.
278.        unsigned long Socket::SetTimeout(unsigned long t_ms) {
279.          unsigned long retval = timeout;
280.          timeout = t_ms;
281.          struct timeval tv;
```

```
282.          tv.tv_sec = timeout;
283.          if (setsockopt(s_, SOL_SOCKET, SO_RCVTIMEO,(char *)&tv,sizeof(struct
    timeval)))
284.            cerr << "Couldn't set socket timeout" << endl;
285.          return timeout;
286.        }
```