HiveProtocol: A Communication Protocol Used to Centralize Sensor Data

David Carlsson
Adam Inersjö
Joel Westerlund
Abstract

HiveProtocol: A Communication Protocol Used to Centralize Sensor Data

David Carlsson
Adam Inersjö
Joel Westerlund

The area of Internet of Things (IoT) is a rapidly growing field within computer science and the ability to effortlessly gather data using sensor networks is increasing in demand. Collection of sensor data can be cumbersome since the sensors by themselves often are unable to communicate with the outside world through the Internet. To facilitate the need to interact with sensor nodes a transport protocol was developed to enable data, collected by sensors, to be propagated to a computer with network capabilities. Our protocol has the ability to send messages over an RS-485 network, with tests showing that all transmitted packets were received at the intended destination, however, a much lower bandwidth usage than desired was observed. Comparing the results of different testing scenarios indicates that, in order to mitigate collisions, a lot of the transmission time is wasted ensuring that the physical cable is not being used by someone else. The developed protocol is considered good enough for demonstrating proof of concept and to be used as a solid foundation for future development.
Sammanfattning

1 Introduction

An estimation showed that there where roughly 8.4 billion devices connected to the Internet in 2017, this is expected to rise to a staggering 20.4 billion devices by 2020 [Gar17]. Gartner also states that one of the more common use cases for connected devices is to gather data. Flexible and easy access to real-time sensor data will allow for an abundance of new applications using the data to provide services to users. One of these applications could be a parking system where a user could enter the intended destination on a GPS locator and upon arrival receiving the location of the closest available parking spot. By reducing the time spent driving around searching for a parking spot, time will be saved and emissions reduced. Other sensors could measure the amount of carbon dioxide in the air allowing an overview of the long-term impact of emission-reducing policies.

The collection of data is often done by analog sensors. However, before the data can be stored and interpreted, the gathered data needs to be converted digitally and transmitted to a computer. Although this can be done by connecting the sensors directly to a computer, using one cable per sensor makes it difficult to scale due to the limited amount of communication ports on a computer.

To accommodate the data collection each sensor could be connected to a sensor node in the form of primitive hardware, such as a microcontroller without direct access to the Internet. The sensor node then needs to transmit the collected data to a computer with the ability to store and interpret the data. To enable collection of sensor data, two approaches were considered. The choice was to either transmit the data using wireless communication or by using a physical cable. The two different approaches have their own advantages and disadvantages. The quality of service is higher when transmitting the data over a physical cable compared to wireless communication, the initial cost of a system using a physical cable is also lower. When using wireless communication, on the other hand, the installation is much easier, and a wireless network can cover a larger area by simply adding more wireless base stations [Wor12].

To support cheap and reliable transfer of data, we chose, in agreement with the stakeholders, to implement the protocol using a physical cable. Utilizing a physical cable instead of wireless communication would allow for a low-cost development of the protocol, that is meant to work as a proof of concept, while allowing for wireless communication as a future implementation. This protocol was developed to provide reliable message passing between hosts over a serial connection. The developed system allows for individual addressing and communication with connected sensor nodes. The system enables data collection by a computer with capabilities to store, interpret and forward the gathered information.
2 Background

Following is an introduction to the subjects of sensors and serial communication. First, we describe the basics of network communication to provide the reader with an introduction to what a protocol is. Then we describe what is meant by a sensor, history of sensors and general information about how sensor networks are used. We proceed with describing the basics of serial communication and its classifications, as our first implementation of the protocol will communicate over a serial link. Lastly, we present our stakeholders and how our work is contributing to the overall goal of a complete infrastructure for collection of sensor data.

2.1 Network Communication Basics

Communication over a network must follow a specific set of rules, called a protocol, to ensure that all devices correctly transmit and receive data [Axe07, p. 7]. In the book *Computer Networking, A Top-Down Approach* a protocol is defined as “the format and the order of messages exchanged between two or more communicating entities, as well as the actions taken on the transmission and/or receipt of a message or other event“ [KR13, p. 168].

As a result of two decades of research, the Internet emerged in the early 1970s. A key part of the research was the development of the Internet Protocol [Cou12, p. 106]. The Internet Protocol was designed to enable transmission of data from sources to destinations in computer networks [Pos81].

The Internet Protocol stack is organized into five different layers: *application, transport, network, link* and *physical layer*. By structuring the protocol into layers, each layer is able to achieve its responsibilities by using the services supplied by the layer directly underneath [KR13, p. 49-50]. As an example, the physical layer is responsible for transmitting single bits and bytes, while the link layer handles groups of bytes that forms what is called a *packet* [KR13, p. 80]. The most common protocols used on the Internet are called the *Transmission Control Protocol* (TCP) and the *Internet Protocol* (IP), collectively known as *TCP/IP* [KR13, p. 33].

A *gateway* is a device that allows for different networks to communicate, the gateway receives a packet containing data and information about sender and recipient. The gateway interprets the recipient’s address and propagates the data to another device allowing the packet to reach the intended recipient [KR13, p. 424].
2.2 Sensors

A sensor is a device that measures and reacts to physical or chemical properties. Sensors convert a property into analog electrical signals that can be accessed by computers with the use of analog-digital converters [SM16, p. 525, 804]. As sensors need a power source to measure their surroundings they are often connected to a sensor node. A sensor node is a small computer that can provide a sensor with power, read and store the measurements made by sensors and transmit and receive data [DP10, p. 47]. A single sensor node can have several sensors connected to it.

2.2.1 History of Sensors

In 1883 Warren S. Johnson invented what can be called the first sensor in the form of a thermostat [MPPA10], marking the birth of electric measuring devices. Since then a multitude of sensor categories have been created to meet the needs of industry, governments and the general public. The categories include infrared, ultrasonic and humidity sensors [Cou01].

The use of sensors has evolved from direct control of systems, as in the case of the thermostat used for controlling room temperatures developed by Johnson [MPPA10], into the collection and storage of large amounts of data about the world. Sensors are today still used in the control of systems, such as in autonomous vehicles [GLPL14] and automatic sustainable agriculture [KS14], but they are more and more used in areas such as environmental monitoring both underwater [VKR’05] and on land [KKA’15].

2.2.2 Sensor Networks

With the shift in sensors usage, from control to monitoring, the data acquired has to be collectible for further processing and storage. To interpret the data it has to be centralized in locations often far away from the sensors themselves. This centralization of data can be aided by using the existing global system for transferring information in the form of the Internet. To gain access to the Internet sensor nodes have to communicate with gateways. When multiple sensor nodes are connected to each other and a gateway they form what is called a sensor network [DP10, p. 7]. There exist both wireless and wired sensor networks [CK03] and the terminology differs between them. In wireless sensor networks the middle hand between the sensor nodes and the Internet is called a base station [DP10, p. 7], but to make the discussion more general we will from now one use the word gateway for both wireless base stations and wired gateways.
According to Roman and Lopez, there are three main solutions that can connect sensor nodes to the Internet: front-end proxy, gateway and TCP/IP overlay [RL09]. The three solutions can be seen in Figure 1, in common with all these solutions is that the sensor nodes connect to a gateway on their way to the Internet. The solutions differ in how the communication between the sensor node and the gateway is implemented.

In front-end proxy, the gateway and sensor node have both their own communication protocol and data representation. For the server to get data from the sensor node the gateway must translate the data and change protocol.

The gateway solution, on the other hand, uses the same data representation between all devices. Here no data translation is needed but the communication between sensor node and gateway still uses a different protocol than TCP/IP.

Finally, the TCP/IP overlay solution uses the same protocol and data representation for all communication. This solution requires the least amount of work to be done by the gateway as it only has to pass packets between the sensor node and the server without processing [RL09].

The communication protocols used on the Internet were implemented to work on devices without unlimited access to electricity [SYDZ16], but to make the deployment of sensors feasible they need to be battery driven to allow for placement in general areas. Therefore Shang et al. describe the need to completely re-architect the communication
between sensors and the Internet [SYDZ16].

2.3 Serial Communication

According to Jan Axelson author of the book *Serial port complete: COM ports, USB virtual COM ports, and ports for embedded systems* [Axe07, p. 1] “a serial port is a computer interface that transmits data one bit at a time”. The serial port may transmit data over a physical cable that connects several devices together, known as a bus [The18].

When using serial communication to transmit data in two directions, a duplex system can be used [PM03, p. 178]. According to Park and Mackay, the duplex system is divided into full-duplex and half-duplex. The full-duplex allows data to be transferred in both directions simultaneously while half-duplex only allows for data to be transmitted in one direction at the time.

2.4 Stakeholders

This project was a part of a joint venture between FZ Elektronik AB and Triangela AB, called *HiveNet*. The goal was to create an infrastructure for gathering and giving users access to sensor data. The infrastructure can be used to create a wide variety of applications such as car parking monitoring and mailboxes that gives alerts when new mail has arrived.

FZ Elektronik creates the sensors and sensors nodes while Triangela focus lies on the user interface and database design. Our protocol is meant to bridge their implementations creating a modular Internet of Things (IoT) solution. FZ Elektronik AB is a small Uppsala based company. The main focus of FZ Elektronik is consulting other companies, providing expertise and knowledge about design and implementation of embedded systems [FZ 18]. Triangela is a software development company specialized in creating advanced customized web applications and database systems [Tri18].

3 Purpose, Aims and Motivation

The purpose of this project was to facilitate the collection of sensor data by developing a communication protocol to be used when sending data between sensor nodes and a gateway. This protocol would be a first step in centralizing the data gathered by the sensor nodes.
Our protocol is intended to work independent of the underlying hardware infrastructure, whether it is wireless or physical cable. The goal was to create a protocol that would guarantee delivery and detect errors that occurred during transmission. The guaranteed delivery means that when sending data between devices the sender will resend the data either until the data has been successfully received or until a certain amount of attempts has been done. If the data was not able to be sent the protocol will indicate an error.

The protocol was implemented using RS-485 adapters, a physical layer bus allowing for serial communications. To test the protocol a gateway was implemented that can use the protocol and evaluate its performance.

When evaluating the performance we hoped to answer the following questions: How long will it take to transfer packets with varying data sizes between a sender and a receiver? How large is the amount of correctly delivered packets when sending packets over the network? Since most data transmitted will be small, around 7 bytes, will we able to utilize 80% of the total bandwidth when using the stakeholders requested data transfer rate of 115 200 bits per second (bps) using half-duplex? Since the sensor nodes will share the same physical layer, can we ensure that only the intended recipient accepts the transmitted data? When the data arrives, how do we know that the data has not been corrupted? How much longer will it take to send 1024 packets of the smallest possible data size containing 1 byte of data as opposed to sending one large packet with 1024 bytes of data?

In contrast to the multitude of other network protocols available for sensor communication, our protocol was not designed for open and general use, but instead for the needs of our stakeholders. This allowed us to create a protocol meeting the specific needs of the joint venture HiveNet and gave us the ability to make assumptions on its use. By removing the need for general purpose functionality found in open protocols, which requires them to be adapted to work on a variety of platforms, the protocol could be designed simpler and allowing the functionality to be adapted to the needs of the company HiveNet.

Centralization of sensor data facilitates for online applications to effortlessly access the data. When creating a proprietary protocol, to allow data centralization, sensor nodes can be designed in collaboration with the design of the protocol, allowing sensor nodes to both have longer battery life and to be used in areas previously not economically viable.
3.1 Delimitations

To be able to create a robust minimum viable product (MVP) we had to impose limitations. Some of the limitations involve the lack of secure communication by not encrypting the packages being sent and only implementing the protocol for one physical layer, the RS-485.

Initially, the plan was to implement the protocol to work on both a wired and wireless physical layer but the idea was abandoned in order to focus on other functionality and features of the protocol. The testing of the end product was limited to using RS-485 serial cables for communication between the sensor nodes and a gateway.

The intention was to make the protocol as hardware independent as possible, enabling the source code to be run on different types of hardware, for example on personal computers as well as microcontrollers. Because of time constraints and limited access to hardware our protocol was developed and tested only on personal computers using the Linux operating system (OS).

Secure communication is vital for all distributed systems, however, since this prototype will not be deployed in commercial use, transferring correct data instead of secure encrypted data was prioritized.

Broadcast messages were not implemented in this version of the protocol. The need to broadcast messages was not necessary for an MVP. Due to the time constraints of the project and the low priority of broadcast messaging, broadcast messages were not implemented.

This protocol will not be able to handle communication on more than one bus. Initially, communication over several buses connected to the same device was planned. This limitation was also due to the project’s time constraints.

4 Related Work

In this section, we examine other protocol implementations of both larger and smaller scale. The protocols characteristics are then discussed and compared to our implementation.
4.1 Link Communication in the Internet Protocol Stack

The Point-to-Point Protocol (PPP) is a communications protocol used in the link layer of the Internet Protocol stack [Sim94a]. PPP is named point-to-point because it is used for communication between two devices connected directly with a cable, often routers in the Internet. PPP is an Internet standard that was released in 1994 and provides multiple services including packet encapsulation and its own protocol for controlling the communication over a link [Sim94a].

In comparison with PPP, our protocol will allow for multiple devices to communicate over the same physical line, called all-to-all communication. Furthermore, our protocol will not implement as wide a variety of services as PPP does. The inspiration for our encapsulation came from PPP, which will be further discussed in Section 9.1.

4.2 Sensor Node Communication

Modbus Plus is a serial communication protocol enabling communication between what we call sensor nodes [MOD96]. The sensor nodes communicate using peer-to-peer (P2P) technique, where any device can initiate communication with other devices. This P2P technique is similar to our protocol’s use of a multi-master technique that allows for the sensor nodes to initialize communication. Using the multi-master technique allows for instant updates if changes occur in sensor data in contrast to occasional polling of the sensors. The Modbus Plus protocol’s network layer also frames the data before transmission in a similar fashion as our link layer does. The link layers framing of data is described in Section 9.

4.3 Effective Bus Usage

A resource effective bus protocol named ”Tiny Controller Network” was created by Åhman [Åh15]. His multi-master implementation was intended to provide communication between 8-bit microprocessors over an RS-485 network. In his work he focused on the ability for usage on simple hardware sending only small amounts of data and implemented a time synchronization function adapted for microcontrollers.

In contrast, the intention of our protocol is to function on more advanced hardware, with the ability to send messages of various sizes and to also be independent of the underlying physical communications layer. Furthermore, we did not implement time synchronization in our design.
5 Implementation Method

The following section aims to describe the methods and tools used in the development of the project. Firstly, to substantiate our use of programming language, three potential languages were compared and analyzed. We then explain how we validated the functionality of our system and describe our approach to test a system that is dependent on peripheral devices to function. Lastly, we compare different serial communication interfaces and how we decided on the communication standard used.

5.1 Choice of Programming Language

Our project focused on the implementation of a serial communication protocol working on the Linux OS. The requirements of the programming language to develop this protocol were that it would be fast, be able to access the serial ports in Linux, allow for low level data manipulation and work on devices with limited performance and battery life.

Although most modern programming languages fit the above characteristics, the following analysis compares three languages: C, C++ and Rust.

C is a high-level general-purpose programming language that was originally designed for the UNIX operating system by Dennis Ritchie [KR88, p. 1]. It was designed to map its functionality directly to that of the underlying hardware, giving programmers full access to the inner workings of the computer and trying not to control how programmers think or write their programs [KR03].

C++ was released in 1985 and was meant to be a successor of C [Str03]. It was focused on simplifying communication between developers, supporting object-oriented design and programming, while still keeping the efficiency and flexibility of C. C++ was designed both for creating everyday applications and lower level programs that control computers. C++ has also been used to create operating systems [CIRM93]. A comparison done in 2008 for programming in bioinformatics showed that C and C++ have a comparable speed and memory usage [FG08]. This means that although C++ was designed from the point of view of communication between programmers [Str03] it has not lost significant performance in comparison with C.

Rust is a programming language that, just like C++, is used for the development of applications and lower level programs, allowing for many kinds of programming methodologies [Rus18]. Rust was released in 2010 with the focus on building from programming languages from the 1980s and 90s, including C and C++ [Hoa10]. One reason for the
release of Rust was to create a safer language that wouldn’t allow common problems such as memory errors and data races [MK14]. A comparison by Perez showed for a specific algorithm that Rust was faster than C++ even though Rust was developed with more safety in mind [Per17]. In general, these kinds of comparisons are no proof of which language is faster, but it can be used as an indication.

Although both Rust and C++ have more functionality and were designed with simpler expression of ideas than C while still allowing for the same access to the serial ports and OS functions in Linux, the protocol was implemented in C. This was done to facilitate the integration of the bigger system that our the protocol will be a part of, implemented in C.

5.2 Versions of Testing

To ensure that the developed protocol was working properly rigorous testing was needed. Following are the techniques used to achieve this. During the entire testing process, a unit testing framework called CUnit was used. CUnit is a framework that allows for validation of single units of code and functionality [CUn15].

5.2.1 Testing for memory leaks using Valgrind

Valgrind is a framework that helps with debugging and profiling code [Val17]. The main use of Valgrind for this project was to eliminate memory leaks, that is ensuring that allocated memory is returned to the system when no longer needed. If the memory allocations are not returned when not needed anymore, memory resources will be wasted. Deallocation of memory is usually done automatically by the programming languages, however, this feature is not available in C. Valgrind keeps track of all allocation of memory and helps the developer to manually free all allocated memory.

5.2.2 Testing as a Development Practice

During the design and implementation phase of a program, there are two main techniques when it comes to testing, namely test-first, and test-last [EMT05]. In test-first the tests are written before the actual implementation and is a part of the design process, while in test-last the tests are used to validate the behavior of the already implemented code.

Test-driven development (TDD) is a methodology that applies the test-first technique [JS05].
5 Implementation Method

TDD starts by writing automated tests and when a test is completed the corresponding function is implemented and the correctness of the function can be validated immediately. A study following a software development group from IBM showed that TDD decreased defects in the code while the robustness of the code and the smoothness of code integration increased [WMV03]. For this reason, we chose to apply TDD during the implementation of our protocol.

5.2.3 Testing in Isolation with Mock Functions

When developing a programming project with several interconnecting modules, or when using input-output operations that depend on physical devices, it can be difficult to apply unit testing. This is because when testing the functionality of a module its dependent modules will be called, and the result of the test relies on the underlying modules to function correctly. This problem can be seen in the left part of Figure 2 where a function, that is to be tested, depends on two other functions, who in turn depend on both further functions as well as physical devices. To mitigate this problem of dependencies, and getting back to testing individual units, a technique called mocking can be used.

Mock functions work by creating replacement functions during the testing, which will be called by the function to be tested. The behavior of the replacement functions can be controlled during testing to both return certain values, but also to check that they were called as expected. Using mock functions in this way assumes knowledge of the function and its dependencies. The right side of Figure 2 shows the dependency graph when mock functions have been created. The two functions can be completely controlled, making the validation of the function unit much simpler. Although the practice of using mocks first was conceptualized for object-oriented programming to be used in a
programming methodology called *Extreme Programming* [MFC00], they are also useful in the context of imperative programming in C by applying the technique on functions instead of objects [KWBF07].

Mock functions were used during the development to simplify our unit tests and remove dependencies between our own code modules, as well as external function calls. We chose a framework called *Fake Function Framework* (FFF) [Lon18]. This framework was chosen over others such as *CMock* [Thr18] and *C-Mock Google Mock Extension* [Jag18] because it is implemented fully in C, not requiring the use of any extra programs as the other frameworks did. Another reason was that FFF does not provide its own testing framework, making it possible to use inside of CUnit.

### 5.3 Serial Communication Considerations

Three different communication interfaces, the RS-232, RS-485 and USB were considered for this project. As stated in Section 3 the interface needs to allow for more than two devices to communicate over the same medium using half-duplex communication with a required data transfer rate of 115 200 bps.

The RS-232 interface uses full-duplex serial communication [ARC17], allowing for 2 devices to communicate simultaneously over a distance of up to 100 feet translating to about 30 m, allowing a maximum data transfer rate of 20 Kbps [Axe07, p. 3].

RS-485 allows for 32 units to communicate over the same bus [Axe07, p. 3]. A cable of length up to 4 000 feet corresponding to 1 200 m can be utilized by the RS-485 interface. Transfer rates of up to 35 Mbps can be achieved when using shorter cables around 12 m. RS-485 operates with half-duplex [Bie17a].

The Universal Serial Bus allows for high data transfer rates, reaching up to 10 Gpbs using the USB 3.1 Gen 2 standard [USB11]. However, the cable length of the USB is limited, allowing for distances of up to 5 m for USB 2.0. This can be expanded up to 30 m by repeating the signal every 5 m using USB hubs [Bie17b].

For this project RS-485 was chosen as the communication standard. The low data transfer rate and ability to only allow communication between two devices eliminates the choice of RS-232. The data transfer speed provided by USB is much higher than the one of the RS-485, however, for the collection of sensor data, transfer speeds at 35 Mbps utilizing half-duplex are more than sufficient. Since we cannot guarantee that the sensor nodes will be located within 5 m of the gateway we must allow for a longer cable to be used leaving only RS-485 as an eligible communication interface.
6 System Structure

Our aims with the project were to develop a protocol to structure and provide reliable communication between hardware in form of a computer and sensor nodes. The goal was to make data, collected by sensor nodes, available for online user applications. As the sensor nodes often do not have direct access to the Internet, some sort of middleware is required providing the necessary capabilities, for example, a PC.

The protocol structure is intended to provide the possibility for easy switching between communication media. As an example, it should be easy to replace a physical link, connecting devices, into wireless dongles. To accommodate for this the protocol was divided into several layers as shown in Figure 3. The layers have an interface in between each layer, in this way the protocol can be easily adapted to different application requirements as long as the implementation adheres to the interface requirements. If the implementation of the protocol would not be divided into layers, any small change in functionality would require that the whole or at least large parts of the protocol had to be rewritten.

In the same way as the Internet Protocol is structured into different layers, our implementation follows a similar layering concept. By doing so we are supporting our requirement for portability. There is no need to build such a complex protocol as the Internet Protocol for our small-scale application. Hence our protocol was decided to consist of only application layer, transport layer and link layer.

![Diagram](image)

Figure 3: The layers implemented in our protocol stack and the propagation of a message from sender to receiver.
The application layer was only implemented with basic functionality to provide means to test and demonstrate the functionality of the underlying layers. A command line interface was developed so that a user could send and receive messages through a menu. The intention of the stakeholders is to later develop this layer to include functionality such as database storage and Internet communication, but that is outside the scope of this project.

When sending a message, as illustrated in Figure 3, the message is passed from the application layer down to the transport layer along with the address of the intended message recipient. When a message arrives at the transport layer a packet is produced by wrapping the message with information such as sender and receiver address. The packet is then sent to the link layer where all the packet information and the message itself is placed inside a buffer before being sent over the physical layer.

When receiving a message the received bits are placed in a buffer by the link layer and the packet is reconstructed by the receiver's transport layer. The packet is then stored in a queue awaiting collection by the application layer. When the application layer collects a packet from the queue the message and the sender address are extracted from the packet.

To provide the ability to test our protocol we, as a first implementation, connected our computers over a serial connection with RS-485 USB-adapters. One of the computers was acting as the gateway computer, and the other computers acting as sensor nodes. A sketch overview of the system can be seen in Figure 4.

As several sensor nodes can be connected to the same physical layer there is no way of knowing who the sender and receiver of a packet are without some sort of addressing scheme. When a message is sent over the bus it will be received by all the connected computers. Therefore our protocol has a sender and receiver address incorporated in the packet format. In this way messages, collected by a different computer then the intended message recipient, can be disregarded.
7 Requirements and Evaluation Methods

In this section, we will specify the requirements of the protocol, while also explaining how we intend to evaluate if we managed to satisfy those requirements. Firstly, to be able to use the protocol efficiently, more than one node must be allowed to operate at the same time on the shared physical cable. Another requirement is that before the data is accepted, the data must be validated, otherwise the faulty data might corrupt the system causing undefined behavior. Another important requirement is that only the intended receiver should handle the transmitted data, as it would not be efficient if all devices would consider all data transmitted over the shared physical cable. Finally, all memory allocations must be tracked and freed so that memory leaks are avoided, which otherwise could cause the system to run out of memory. Below we list the requirements and the evaluation results can be found in Section 10.

**Connectivity**: At least 10 sensor nodes must be able to simultaneously use the same physical cable. Due to lack of hardware, we could not create 10 sensor nodes. Instead, we connected two sensor nodes and allowed them to simultaneously transmit 1000 packets. The average transmission time of the packet was calculated and the number of packets that had to be resent was measured.

**Performance**: The protocol needs to utilize at least 80% of the bandwidth when using a data transfer rate of 115 200 bps. We tried to find the highest possible usage of bandwidth utilization by connecting two devices in form of regular computers. One of
the computers would transmit 1000 packets. The bandwidth utilization was then calculated by dividing the average bandwidth used for each packet with the full bandwidth of the communications channel. The average bandwidth was in turn calculated according to Equation 1, where Total Packet Size is the size of both the data and packet overhead. The total size of a packet was used in the calculations because we were interested in the average utilization of the total bandwidth, regardless of what type of data being sent.

\[
\text{Average Bandwidth} = \frac{\text{Total Packet Size [bits]}}{\text{Average Time [seconds]}}
\] (1)

**Data validation:** When transmitting data, the protocol must ensure that the data received is the same as the data transmitted. Validation of data is important to ensure that the received data is not corrupted, causing undefined behavior in the gateway. Automated tests were implemented using a number of predefined packets on both the sensor node and the gateway. We then measured how many faulty packets were detected, discarded and resent.

**Addressing:** When data is sent, only the intended receiver is allowed to process the data. To test the addressing, manual tests, using three sensor nodes with different addresses were connected to a gateway. If the wrong receiver accepts the transmitted data, the protocols integrity will be lost.

**Portability:** Since the protocol is intended to enable communication between a gateway and sensor nodes there will be hardware specific differences to take into consideration. The stakeholders required the implementation to separate the hardware dependencies allowing the protocol to be ported to less advanced hardware such as a microcontroller. To test the portability of our protocol we would need to implement it on another hardware platform than the one we used during development.

**Memory deallocation:** All memory allocated must be deallocated when not needed by the system anymore. If the allocated memory is not returned to a computer running the protocol might run out of memory and will not be able to receive anymore packets. A tool called Valgrind described in Section 5.2.1 was used notifying the developer if any memory leaks occurred with information about what part of the code was responsible for the leak. We would consider this demand to be met if no leaks are found by Valgrind.

## 8 Transport Layer

For a message to be sent to another process a sequence of things has to be accomplished. When a message is sent, the message is first passed to the transport layer along with the
address to the intended recipient of the message. The message is then encapsulated within a packet and information, such as sender address, receiver address, and checksums are added to the message. The packet is then passed on to the underlying layers for further transport. This section handles the details of our transport layer implementation and its services.

### 8.1 Sequence Numbers

The transport layer keeps track of ongoing communication with other processes. Every time a new sender or receiver address is encountered a random sequence number is generated and associated with that address. The sequence number is then attached to the next packet that is to be sent and incremented each time a packet is successfully transmitted.

A packet could be received more than once if an acknowledgment was lost. Acknowledgments are explained in Section 8.2. Therefore, when a packet is received, the sequence number of the packet is compared to the previously received sequence number to verify that the packet has not, in fact, already been received. When the same sequence number is encountered twice the latter packet will be discarded, thus avoiding propagation of duplicate messages to the application layer. If a completely new sequence number is received from an address the receiver will assume that it is the first packet received or that the sending process has been restarted, in both cases the packet is accepted.

### 8.2 Acknowledgments

For a sender to know whether a packet is received by the process associated with the destination address, and to give the sender a chance to resend the packet if anything goes wrong, an acknowledgment packet, called ACK, is returned when a data packet is received by the transport layer. The acknowledgment packet is generated by reversing the sender and receiver address and attaching the same sequence number as the sequence number of the received packet. Hence the sender gets a confirmation that the sent packet is, in fact, received at the destination. A timeout of 10 ms is used at the sender and if no acknowledgement has been received within that time the packet will be resent. If no acknowledgment is received after five tries the sender gives up and produces a notice indicating that the transmission failed. According to the Linux manual [Ker17], the TCP protocol has a default setting of three retries and a maximum of fifteen, therefore we decided to go with five retries in our implementation as a reasonable initial configuration.

In the case where a receiver takes delivery of an out of order sequence number, there
is no reason for the receiver to acknowledge the last correctly received packet. This is because the packets are sent one by one and there is no storing of old packets at the sender process. Hence, the sender is unable to resend a previous packet if the next packet has already been sent. When an out of order sequence number is received the packet is accepted as described in Section 8.1.

The 10 ms timeout was chosen by analyzing the round-trip time (RTT) of a packet. To calculate the RTT a formula for calculating the end-to-end time of a packet was adapted from Kurose and Ross [KR13, p. 69]. In Equation 2, $d_{end-end}$ is the time it takes for a packet to reach its destination, $d_{proc}$ is the processing time at both sender and receiver, $d_{trans}$ is the time it takes for a sender to transmit a packet and $d_{prop}$ is the time it takes for a packet to propagate from one device to another through the physical cable. $N$ is the number of devices that the packet will reach on its way to the receiver. In our case there are no intermediate devices between the sender and receiver so $N$ is equal to 1.

$$d_{end-end} = N \times (d_{proc} + d_{trans} + d_{prop}) \quad (2)$$

From the above equation, the RTT of a packet in our protocol was calculated. The calculation is separated into two parts, the end-to-end time for the packet and the end-to-end time for the acknowledgment, as can be seen in Equation 3 and 4. In these equations $d_{pkt-...}$ and $d_{ack-...}$ are the corresponding values as in Equation 2 but for the cases of a packet or acknowledgment.

$$d_{pkt} = d_{pkt-proc} + d_{pkt-trans} + d_{prop} \quad (3)$$

$$d_{ack} = d_{ack-proc} + d_{ack-trans} + d_{prop} \quad (4)$$

In the protocol, we chose to start the acknowledgment wait timer when all the data had been transmitted, which means that we could ignore $d_{pkt-trans}$ in the calculation of $d_{pkt}$ in Equation 3. The size of an ACK is 14 bytes = 112 bits, so with the transfer-rate of 115 200 bps, as mentioned in Section 3, the transmit time is $d_{ack-trans} = 112 \text{ bit}/115 \text{ 200 bps} \approx 1 \text{ ms}.$

The processing of the packet and acknowledgment were negligible, but in computers running the Linux OS, there is a latency timer on serial ports that dictates when data is sent [Fut06]. This timer can be set in the program, but the lowest latency is 1 ms. This timer only applies when transmitting to the serial link, not when receiving, so both $d_{pkt-proc}$ and $d_{ack-proc}$ were set to 1 ms.
To find $d_{prop}$ we used the maximum length of an RS-485 cable which is 1 200 m, as described in Section 5.3. Assuming that the propagation velocity is 78% of the speed of light, as described in [Kug16], the propagation time is $d_{prop} = \frac{1 200}{0.76 \times 299 792 458} \approx 5 \mu s$. Combining these calculations in Equation 5 we get the RTT $3.01 \text{ ms}$. From this number we chose a timeout of 10 ms to give flexibility for other hardware implementations.

$$d_{pkt} = 1.005 \text{ ms} \quad d_{ack} = 2.005 \text{ ms} \quad RTT = d_{pkt} + d_{ack} = 3.01 \text{ ms} \quad (5)$$

### 8.3 Collision Avoidance

In our transport protocol multiple sensor nodes are going to communicate over the same physical bus to reach the gateway. When two or more devices try to communicate over the same bus at the same time the data on the receiving side will be incomprehensible, leading to both senders trying to resend when no acknowledgment has been received. Without a collision avoidance scheme, this would continue to happen until both devices gave up and indicate an error.

To mitigate the collision of packets we implemented a collision avoidance scheme where all devices wait before sending any packets. When a packet is handed to the transport layer from the application layer the program will start listening on the bus for incoming packets. If no packet has been received during a random interval between 50 and 70 ms the sender begins to transmit the packet. The RS-485 bus does not allow for simultaneous reads and writes of the bus, as it is half-duplex. This means that regular collision detection algorithms dependent on simultaneous reads and write cannot be used. Therefore we implemented a fixed wait interval and the limits were chosen with generous margins that could later be optimized. The wait time is random to avoid two devices getting locked in trying to resend packets at the same time over and over again. If both try to send at the same time once, there is only a 1/20 chance that they will try to send at the same time again.

### 8.4 Queue

As the transport layer is unable to control the actions of the application layer there is no way of knowing when a received packet will be collected by the application layer. To prevent stalling the entire transport layer while waiting for the packet to be collected, the packet is stored in a queue. The queue is implemented so that the first packet that
enters the queue will be the first packet collected by the application layer. Thus the correct order of messages is maintained.

## 8.5 Ensuring Data Integrity Using CRC and Checksums

There are several ways to minimize the amount of faulty data being received and handled. One way is to use a parity bit where an extra bit is appended to the end of the data. For even parity schemes the appended bit is chosen such that the amount of 1’s in the bit string becomes even, for odd parity the parity bit is set such that the amount of 1’s becomes odd. Parity schemes are according to Kurose and Ross [KR13, p.474] probably the simplest way to perform error detection, however, the parity bit cannot account for even amounts of bit errors.

Another, more reliable method would be to use an 8-bit checksum that can be implemented by first dividing the data into bytes. The bytes would then be added together and the individual bits of the sum inverted producing the checksum. The checksum will allow a guaranteed error detection when two faulty bits are spaced less than 8-bits apart from each other in the original data [Bar99].

Cyclic redundancy check (CRC) is a much stronger data integrity check than both the checksum and parity bit. However, it requires more computational power [Bar99]. To calculate an N-bit CRC number a generator polynomial, that is a string of N bits, is used. The computation of the CRC is done by using modulo division of the data using the generator polynomial as a divisor. The quotient is discarded and the CRC is set equal to the remainder [Wil93]. We implemented a 32-bit CRC allowing for $2^{32}$ different values. However, since there are a lot more than $2^{32}$ possible data messages, different messages can have the same CRC. The risk of two different data messages having the same 32-bit CRC is only $2^{-32}$ [Bar99].

Due to the limitations of the parity bit not being able to detect even numbers of bit errors, the protocol does not use parity bit checking. The protocol instead utilizes an 8-bit checksum to verify that the information about the packet length has not been corrupted when transmitted, while the 32-bit CRC controls that the data is correct when receiving the packet. When sending a message, both the checksum and CRC is calculated and appended to the packet. At the receiver’s end, after the message is received, the checksum and CRC are once more calculated using the received packet’s data. The calculated checksum and CRC are then compared to the checksum and CRC that was appended to the packet. If the checksum and CRC match the data of the packet is considered validated, but if the checksum or CRC differ the packet is discarded and the transmitter has to resend the packet.
8.6 Illustration of a Transmission

An example of how data is transmitted over the network between two hosts is illustrated in Figure 5. Before sending data on the bus the sender waits for a random time as explained in Section 8.3. As soon as the data has been transmitted to the bus the sender proceeds to start a timer in order to give the receiver a reasonable time to send an acknowledgment and to prevent stalling execution of the sender process indefinitely in case no answer is received. Before sending the next packet the sender, again, waits a random time in order to avoid collisions.

![Figure 5: Illustration of a transmission showing the timings when sending two consecutive packets. $d_{\text{trans}}$, $d_{\text{prop}}$, and $d_{\text{proc}}$ are explained in Section 8.2.](image)

9 Link Layer

This section describes the functionality of the link layer in our protocol, stack. First, the general notion of a link layer is described, followed by our design of the layer.

According to Kurose and Ross [KR13, p. 470-471] the link layers in the Internet Protocol stack are responsible for sending and receiving packets of data between two connected devices. Kurose and Ross specify several services that can be provided by the link layer to aid in the transfer of a packet, including error detection, guaranteed delivery and the avoidance of collision. The link layer takes the packet to be sent and inserts
its own overhead to it, creating a so-called frame.

As our transport protocol handles the error detection, guaranteed delivery and collision avoidance, our link layer protocol only has to act as an intermediary layer between the transport layer and the physical RS-485 cable. Our link layer only makes sure that the receiving side understands how big the packet sent is and handles the synchronization of sending data. To keep track of which bytes that belong to which packet a technique called framing is used.

### 9.1 Framing

To transmit packets of data over the physical layer we used a combination of start and stop flags and byte stuffing. Start and stop flags are predefined byte values used to indicate the beginning and end of a frame. Before sending the packet a start flag is sent and a stop flag is transmitted after the rest of the packet. Figure 6a shows the data to be sent and Figure 6b shows the data with start and stop flags added. When the receiver gets the data it begins by looking for a start flag, discarding all bytes that it finds before the flag. Once the start flag is found the receiver reads and stores the data while at the same time looking for a stop flag. When a stop flag is found the receiver knows that the entire frame has been received.

![Diagram of framing process](image)

Figure 6: Illustration over how overhead is added to data in the framing process

The data that is passed to the link layer to be sent may contain any byte values. This is a problem when using start and stop flags because if the data contains the same byte that is used as a stop flag the receiver of the frame will believe that the frame has been completely received, ignoring following data. To avoid this another predefined byte value can be used, called an escape flag. This flag is sent before any data byte that is not
allowed, i.e. has the same value as a stop, start or escape flag. This process is called byte stuffing and can be seen in Figure 6c. When the receiver finds an escape flag it knows that the following byte should be treated as data instead of a flag. The combination of the two techniques can be seen in Figure 6d.

We chose to use a framing schema very similar to the one used in the Point-to-Point Protocol (PPP). PPP uses start, stop and escape flags, but calls the flags octets [Sim94b]. In the PPP definition, the hexadecimal values for the start and stop flags are the same, 0x7E, while the escape flag is 0x7D. PPP also uses a technique where it masks the escaped data by applying exclusive or with the value 0x20 before sending. When receiving an escape flag the receiver does the same operation as the sender to retrieve the original data byte. This masking is needed to make sure that the flagged data byte is not interpreted as a start byte if the receiver has missed the original start byte. The only modification that we did compare to PPP was to implement different start and stop flags to simplify the differentiation of the beginning and end of a frame. We used the byte value of 0x7E as the start flag, just as in PPP, and 0x7F as the stop flag.

9.2 Hardware Abstraction

The Link layer is the layer in our protocol that is closest to the hardware. It handles all the sending and receiving of data as well as interacting with the underlying hardware to initiate the physical ports used. To keep our protocol as independent as possible from the hardware that it was developed on we created what is called a Hardware Abstraction Layer (HAL). A HAL is used to hide the specifics of the platform that is used, instead of allowing the rest of the code to use the underlying functionality through an interface. We chose to abstract the sending and receiving of single bytes and the initiation of the physical layer into our HAL. The abstraction made the rest of the code independent of both the hardware and the physical layer.

If we were to change our physical layer to use wireless communication instead of RS-485 we would only have to change the implementation of the HAL interface. If we later chose to also change the target platform to another operating system or computer architecture we still would not need to change the main functionality of the code.

10 Evaluation Results

In this section, we evaluate the requirements of the protocol stated in Section 7 showing that besides from failing the bandwidth utilization our protocol meets the required
demands.

Connectivity: Connecting two sensor nodes, forcing them to send 1000 packets with data of 7 bytes resulted in a 1.15% utilization of the average bandwidth of 115 200 bps. 297 of the 1000 packets had to be resent due to the sender not receiving an acknowledgment of the sent packet. These results are shown in the left-hand side of the graphs in Figure 7b and c.

Performance: We connected two devices, in this case, regular computers, and sent 1 000 packets from one of the computers with a preset data size. The transmission was done with 7, 70 and 700 bytes of data and the average transmission time and bandwidth utilization were calculated. From the results of the test, shown in Figure 8a, we can see that when sending 7 bytes of data, the average transmission time was roughly 70 ms per packet. The transmission times when sending a packet of 70 bytes did not differ much from sending 7 bytes, and had an average time of 75 ms. When sending a large packet of 700 bytes the transmission time increased to 148 ms. The best bandwidth utilization was reached when sending packets containing 700 bytes of data, leading to a 33% utilization of the bandwidth, as seen in Figure 8b. When the data size was 7 bytes the utilized bandwidth diminished to approximately 2%. The bandwidth utilization was found by calculating the average bandwidth when sending packets and dividing it with the total bandwidth, according to Section 7.

Data validation: All automated tests of the checksum and 32-bit CRC showed that if any data bit was inverted, the checksum and CRC would differ, indicating an error. If an error in either the checksum or CRC was found the faulty packet was always discarded.
11 Results and Discussion

Our stakeholders had a list of demands mentioned in Section 7 that the protocol needed to satisfy. However, our calculations and measurements showed that not all of the stakeholders demands could be met. Most transmissions will utilize small packets, with a data size of roughly 7 bytes. When transmitting these small packets the utilization of 80% of the bandwidth was impossible to achieve. The utilization of the bandwidth is much lower than 80%, close to 2% as shown in Section 10 and the reason is believed to be due to the randomized 50-70 ms wait time that occurs between each sent packet. The wait time was intended to work as a collision avoidance system. Another presum-
ably contributing factor for the low utilization of bandwidth was the time waiting for an acknowledgment before sending another packet.

Because the wait time to send of 50-70 ms was an initial design choice, tests were made to see how the utilization, transmission time and the amount of resent packets would be affected with a lower wait time of 25-45 ms. The evaluation was done using two devices trying to send 1,000 packets, each containing 70 bytes of data, at the same time. This was attempted for both 50-70 ms and 25-45 ms wait time. The results of this test, all shown in Figure 7, showed that although the number of packets that had to be resent increased with lower wait time, from 297 to 345, the average transmission time per packet was reduced from 127 ms to 90.5 ms. This reduction in transmission time lead to an increase in the bandwidth utilization from 1.15% to 1.61%. The bandwidth utilization with the standard wait time of 50-70 ms was lower than in previous tests because these results only show the utilization of one device. At the same time as one of the devices try and succeed to send, the other is doing the same. When testing with two devices sending at the same time each device had to wait when it noticed that the other device was sending data, in accordance with our collision avoidance schema described in Section 8.3. The reduction in transmission time and increase in bandwidth utilization showed us that our initial choice of 50-70 ms can be lowered to optimize the protocol.

The time it would take for the protocol to send 1,000 packets was measured using different packet sizes. This comparison is shown in Figure 8a. For small packets with 7 bytes of data, the average sending time was around 70 ms. This is to be expected considering that the protocol waits roughly 50 ms before starting to transmit the packet and then has to wait until the acknowledgment has been received. When sending a packet with a data of 70 bytes, the average transmission time does not differ much from the smaller packet implying that most of the overhead is indeed the time spent waiting to transmit. Even with an increase of the data size to 700 bytes, the average time spent transmitting a packet remains close to the transmission time of a small packet.

In Section 3 we asked the question of how much the transmission time would differ when sending 1,024 bytes of data as one large packet or as 1,024 packets containing one byte of data. The resulting transmission times can be seen in Figure 9. Sending the data in 1,024 packets took in total about 400 times longer than sending the data in one large packet. When sending 1,024 packets, in addition to sending the data, another 14,336 bytes of packet information such as addresses and checksums must be transmitted. The transmission time of the 14,336 bytes is only 1 second and does not have a large impact on the total transfer time. Having a transmission time that is 400 times larger when sending single bytes shows how much of an overhead the collision avoidance scheme is. Waiting more than 50 ms between each transmission when the actual transmission time of a packet containing one byte data is only 1 ms needs to be reconsidered. A better approach would be to calculate the expected time before the ac-
12 Conclusions

knowledgment is received and adding a small time margin to avoid re-transmission of too many packets.

![Figure 9: Results when sending 1024 packets each containing 1 byte, compared with the average time and utilization when sending packets containing 1024 bytes each, from one device to another.](image)

When sending a packet over the RS-485 cable, any computer connected will receive the data. By connecting four different computers we could see that the addressing of the packets worked as intended since only the intended recipient could read the sent message while the packets addressed to another recipient was discarded at all times.

12 Conclusions

We have simplified the task of collecting sensor data for our stakeholders by providing the ability to send and receive data using our protocol. Our protocol will be an integral part of our stakeholders’ IoT solution allowing for both Triangela and FZ Elektronik to focus on their main areas of expertise. Allowing the stakeholders to focus on their main areas will pave the way for an earlier launch of their IoT solution.

The protocol was able to meet most of the demands that our stakeholders had. We did, however, not come close to the request to be able to utilize 80% of the bandwidth. After a discussion with our stakeholder we could conclude that less than 100 ms, on average, when sending a small packet containing 7 bytes of data is considered good enough, leading to a utilization of approximately 1.5% of the total bandwidth. The transmission
times can, however, be reduced by scaling down the time waiting to transmit and the
time waiting for an acknowledgment, although using too short wait times could cause
data collision and is not desired. Since the stakeholders are satisfied with the current
data transmission speed the protocol will keep its intended wait times, allowing for
collision avoidance. We were able to conclude that sending data in larger packets will
reduce the overall transmission time drastically, but since most of the data sent will be
small, around 7 bytes, the sender would have to buffer several messages and transmit
them wrapped in a larger packet to improve the bandwidth utilization.

As a whole, we consider the project successful in that we were able to create a minimum
viable product of the desired protocol meeting the stakeholders’ requirements, even the
requirement specifying the minimum average amount of bandwidth to be utilized was
met after the demand was revised.

13 Future Work

The following are improvements and features that we suggest to further improve the
functionality of the protocol.

Addressing: In the prototype developed we manually edited the source code to set the
address of the hardware running the code. An improvement would be to have the gate-
way automatically address the nodes connected to the network, keeping track of the
addresses currently in use. This would reduce the overhead of manual addressing when
deployed in commercial use, and reduce the possibility of someone configuring an ad-
dress already in use.

Polling: To keep track of what sensor nodes are currently attached to the gateway and
to mitigate the downtime of sensors, the gateway could individually poll the attached
sensor nodes in defined time intervals. When no answer is received an error could be
sent to the application layer indicating that something has happened to the unreachable
node.

Broadcast: The current implementation of the protocol has no feature to address multi-
ple targets. Even though a host connected to the network listens to all traffic on the bus,
the packets are discarded once the receiver realizes the packet was intended for someone
else. This could be improved by implementing a specific packet type indicating that the
packet should be considered by all recipients.

Security: As of now, there is no security implementation in effect. Hence, the infor-
mation transmitted could be intercepted and interpreted by anyone with access to the
data traffic. Knowing the specific packet format the messages could be extracted from the packets. To prevent eavesdropping the data sent could first be encrypted and then decrypted by the recipient.

*Synchronization:* Improvements could be made in the way communication is initialized between a sender and receiver. At present, the sender produces a random sequence number without notifying the receiver and the receiver has to accept whatever sequence number generated. The initiation of communication could be done in a more orderly manner, where a special data packet would indicate that communication is about to start and what sequence number should be expected. Also, the current implementation has a small chance of packet loss if the sender is suddenly restarted and the same random sequence number is generated as the last sequence number sent before the restart, this would result in the receiver thinking the packet was already received and the packet would be discarded without the sender knowing.
References


References


References


References


