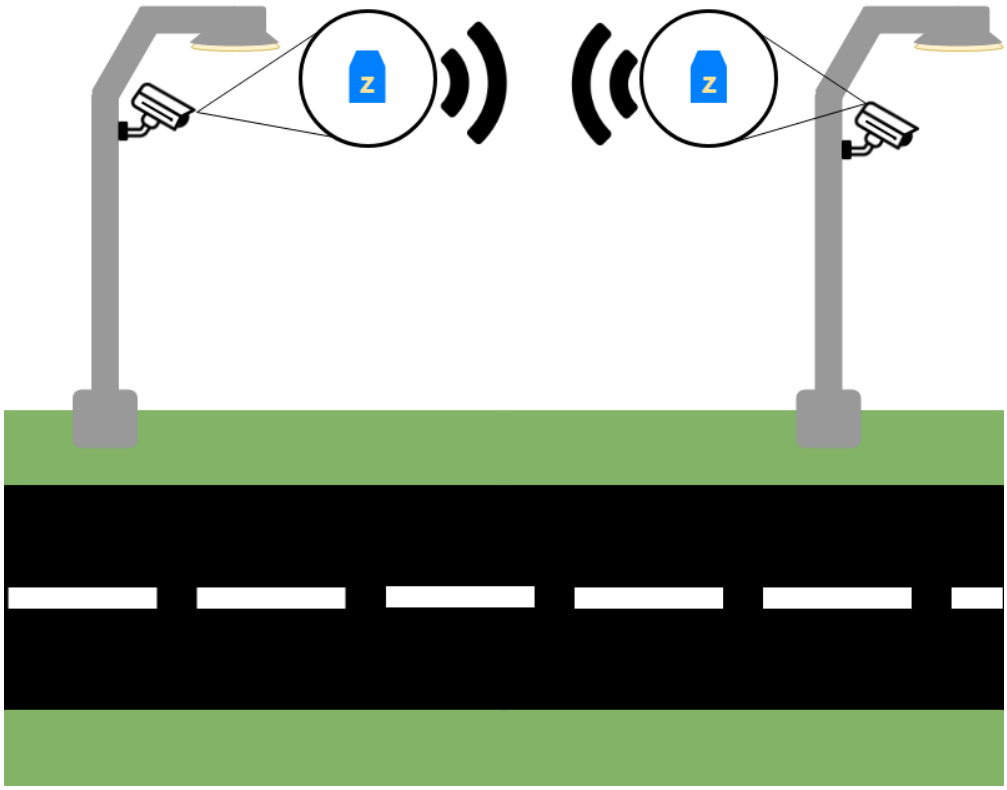

Erlang as an alternative to a non-functional language for communication in a fault-tolerant IoT sensor network

Master's thesis in Computer science and engineering

Jimmy Holdö

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

Master's thesis 2019

Erlang as an alternative to a non-functional language for communication in a fault-tolerant IoT-sensor network

Jimmy Holdö

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

Erlang as an alternative to a non-functional language for communication in a fault-tolerant IoT-sensor network
Jimmy Holdö

Supervisor: Nicholas Smallbone, Computer Science and Engineering
Advisor: Kenneth Jonsson, Cipherstone Technologies AB
Examiner: Carl Seger, Computer Science and Engineering

Master's Thesis 2019
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: An image representing how sensors in a wireless sensor network can communicate between each other with ZigBee.

Typeset in LaTeX
Gothenburg, Sweden 2019

Erlang as an alternative to a non-functional language for communication in a fault-tolerant IoT sensor network

Jimmy Holdö
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This thesis compares a C++ prototype and an Erlang prototype for an Internet of Things application. Internet of Things applications are difficult to program because they consist of a distributed environment of heterogeneous devices, where each device can have limited resources and connectivity technologies. Erlang is a high-level distributed functional language, which can help solve these problems, but an Erlang program may use more resources than an equivalent C++ program.

In this thesis one C++ prototype and one Erlang prototype were developed to handle the communication between sensors in a Wireless Sensor Network using the ZigBee communication technology. These prototypes were evaluated against each other based on power consumption, memory utilization, CPU utilization and lines of code.

The result of the evaluation was unexpected: The Erlang prototype used less memory and CPU in most cases. Therefore, one process in the C++ prototype was further investigated to see why this was the case and it was found that much of the resources required by the C++ prototype came from using dbus for inter-process communication. Without dbus included the C++ prototype would use less resources compared to the Erlang prototype.

The recommendations that can be derived from the investigation in this thesis are that Erlang should be used if the point is to use as little memory as possible and that as long as more than one data packet per second is sent Erlang uses less CPU. Even if the packet rate is less than one per second it can be worth considering the use of Erlang because the code is significantly shorter. Therefore, an Erlang solution should have fewer bugs and fewer security problems.

Keywords: Internet of Things, Wireless Sensor Network, Erlang, IEEE 802.15.4, ZigBee, Network topology.

Acknowledgements

I would like to thank my advisor Kenneth Jonsson and the staff from the affiliated company Cipherstone Technologies AB, who provided tips about how to solve problems and provided the required hardware for my experiments. I would also liked to thank my supervisor Nick Smallbone, who helped me with problems during the time I worked with this thesis.

Jimmy Holdö, Gothenburg, May 2019

Contents

List of Figures

List of Tables

1

Introduction

The number of devices connected to the Internet has been growing impressively for some time and it is predicted that by 2020 there will be around 50 billion devices connected to the Internet (Bello et al., 2017). More and more of these devices are physical objects, such as sensors and actuators. The phenomenon of Internet-connected physical objects is known as the Internet of Things (IoT).

There are many unsolved problems in the field of Wireless Sensor Networks (WSNs). One such problem is the complexity of designing, coding, and testing Wireless Sensor Network (WSN) applications. The problem arises because of the distributed environment of heterogeneous devices in a WSN, where each device may have limited resources and the communication between devices may have many specific requirements, e.g. the full TCP/IP stack may be missing (Sivieri and Cugola, 2012).

One possible approach to solve some of these problems is to use the Erlang language which was designed in the 1980s at Ericsson by Armstrong, Williams, and Virding. At first Erlang was developed for embedded telecommunication systems but it has continued to grow over time and now it is a complete platform with many libraries that offer functionality for a wide range of different applications (Erlang, 2019).

Erlang was developed as a high-level functional language for hiding distribution with facilities such as:

- Support for building software with guaranteed fault tolerance.
- Support for binaries as a data type which in turn allows pattern matching on bit-streams.
- A lightweight concurrency model.
- A virtual machine, which allows the same code to be run on heterogeneous devices.
- Distributed programming, with support for high-level communication primitives.
- Support for transparent resolution of process names over a network.

These features should make it possible to achieve reliable and fault-tolerant communication between nodes in a WSN and because of this Erlang seems to be a good fit for a developing WSN applications (Sivieri and Cugola, 2012; Sivieri, 2012).

1.1 Internet of Things

The concept of IoT is to connect every network-enabled device to the Internet thus creating a "smart world", where our everyday objects can connect to each other to share data with the aim of enhancing our lives. There are many examples of applications that can enhance life in areas such as healthcare, smart buildings, social networks, environment monitoring, transportation and logistics, etc. Every IoT application depends on data collected from a network-enabled device or devices and there exist many different data collection devices and systems, e.g. RFID, sensors and wireless sensor networks (WSNs) (Yang, 2014).

1.2 Aim

The goal of this thesis was to evaluate the performance of using Erlang to manage the communication in a WSN. In this case the WSN application is based on an idea of a product from the affiliated company, where video-based sensors should communicate data to a control system. In order to evaluate the performance two prototypes modelling the communication in this application were developed, one with the low-level language C++ and the other one with the high-level language Erlang. The performance was measured by comparing the number of lines of code, to investigate the effort of implementation for the different prototypes, and the usage of CPU, RAM and power for different rates of sending messages.

In a WSN it is often required that the devices are power-efficient and that devices can communicate directly with each other. This is something that the standard Erlang distribution does not implement and a connection to a suitable networking protocol that can manage device-to-device communication must therefore also be implemented as part of the thesis.

For the resulting thesis to be considered successful the following points should be achieved:

- A C++ prototype that uses a suitable network protocol for energy-efficient device-to-device communication.
- An Erlang prototype that uses the same protocol as the C++ prototype. Because of this some connection to this protocol must be implemented as part of the Erlang prototype.

- An analysis of data collected from the prototypes presented so as to give someone who wants to use Erlang for the communication between nodes in a WSN a guideline for the hardware requirements.

1.3 Limitations

There are many network protocols and multiple variants of some of them. Because of the limited time for the project only one of these protocols and variants is going to be investigated.

The communication requirements for WSN applications vary because the amount of data that needs to be sent can be very different from application to application. The prototypes in this thesis are designed so that each sensor processes its own data and only sends the result, which can be represented with short messages. This matches the expected behaviour of the video-based sensor network. Therefore, WSN applications that send huge amounts of data are not investigated in this thesis.

1.4 Affiliation

The thesis work was carried out at the company Cipherstone Technologies AB. The company is currently in the first stages of developing a new WSN product. This product is planned to be a video-based sensor network where each sensor processes images and then sends results to a control system. The company wants to investigate possible communication solutions for this network.

2

Background

2.1 IoT

As mentioned in Section 1.1 IoT is the concept of connecting all things to create a "smart world". The release of IPv6 has encouraged the development of IoT because of the huge address space that is now available. According to Leibson (2008): "So we could assign an IPV6 address to EVERY ATOM ON THE SURFACE OF THE EARTH, and still have enough addresses left to do another 100+ earths. It isn't remotely likely that we'll run out of IPV6 addresses at any time in the future".

A device can therefore be connected to the Internet without any limitations. The challenge is instead found in the complexity of developing an IoT application and the ensuing security problems because of this complexity (Sivieri and Cugola, 2012).

IoT applications often have concurrent event sources and unreliable communication between devices but they still need to work reliably in the presence of these problems. According to Armstrong (2010) functional programming languages are good for writing highly concurrent application with many processes that at the same time are fault-tolerant in a reliable manner. Functional languages could therefore be a candidate for writing IoT applications (Haenisch, 2016).

Functional languages can also easily describe the functionality of data processing. The data in an application is usually only processed once and this is when it is created. At this time the data is often evaluated in some manner, e.g. for detecting errors. The algorithms for processing data in an application can therefore be seen as a set of mathematical functions operating on a stream of values, where each function creates a new stream of values that can be used in another function to process the data (Haenisch, 2016).

In IoT applications the need for security varies from application to application and applications can be hard to update. Therefore, it is a good idea to use techniques that minimize security risks. The use of a functional language like Erlang could reduce some security risks, as described below.

There does not exist any formal proof for the assumption that a functional language produces more concise code but there is some anecdotal evidence, for example Car-

mack (2013), that suggests that functional languages reduce code size. It is a good idea to minimize the code size of an application as much as possible, because as Ray et al. (2014) show, more concise code tends to have fewer errors. This in turn leads to a reduction of security risks.

Security risks that may arise in an application because of errors in the code include buffer overruns and data races. These risks are eliminated by the use of Erlang. Buffer overruns are impossible because all buffer accesses are bounds-checked and data races are avoided because Erlang processes communicate using message passing instead of shared memory.

One final advantage of using a functional language over an imperative language is that it reduces side effects. The reduction of side effects by using "pure" functions in a functional language in an IoT application results in fewer bugs and therefore the security of the application increases (Haenisch, 2016).

2.2 WSN

WSNs are the next evolution of sensor and actuator networks, which have been around for decades. They solve what is commonly known as the last meter problem in sensor and actuator networks. The problem is that the installation process of such a network is often expensive and complicated and connectors and cables can with time become loose, lost, misconnected or the hardware can even break (Yang, 2014).

When talking about "wireless sensor and actuator networks" people have adopted the shorter name "wireless sensor networks" instead. A WSN is a set of sensor nodes, where a sensor is a low-cost package that integrates wireless communications, sensors and signal processing. A challenge in a network of sensors in a WSN is that a sensor is often required to have low energy consumption and specific coverage requirements and that it is bound by latency (Essameldin and Harras, 2016; Yang, 2014).

In the past the expansion of WSNs has been limited because of the lack of standardization of technologies for communication in the network and at the application level. Communication with higher data throughput has been the main focus in the industry and this has resulted in short-range wireless connectivity techniques being left behind (Gutierrez et al., 2004).

There are many areas where WSNs can be used but an important feature that is required of a WSN is that that it is easy to connect sensors to the network, because a network can consist of a large number of sensors (Gutierrez et al., 2004). Yang (2014) lists some example application areas of WSNs:

- Continuous sensing for environmental and condition monitoring.
- Event detection for disaster response.

- Location sensing for mobile target tracking and localization.
- Local control for home automation, industrial automation etc.

In all these application areas there are some high-level issues that need to be considered when designing and implementing a WSN (Gutierrez et al., 2004): power consumption, range, availability of frequency bands, network topology and self-organization.

Applications sometimes require that the power consumption should be very small. Sometimes, they use batteries as a power source with completely untethered RF transceivers. Since a WSN should be easy to install and low-maintenance it is not practical to require that batteries be replaced. To solve this problem, the usual solution is instead to use power cycling. If the duty cycle is of less than 0.2% then an AAA battery with a capacity of 750mAh can power a normal short-range radio transceiver, with a active current of 10mA, for at least five years.

To transmit data between a transmitter and receiver they need to be inside range of each other. Because of implementation costs and governmental regulation the RF power output in a wireless system operating in unlicensed bands normally ranges from 0 dBm to 20 dBm. This has the consequence of limiting the possible range between a transmitter and receiver. To side-step this issue, WSNs use multihop network protocols with a suitable routing algorithm.

Another issue is the availability of frequency bands. The RF spectrum is a scarce resource so is often regulated by governments with a set of rules that need to be followed. The most commonly used bands in WSNs are the following:

- 868.0 – 868.6 MHz: Available in most European countries.
- 902 – 928 MHz: Available in North America.
- 2.40 – 2.48 GHz: Available in most countries worldwide.
- 5.7 – 5.89 GHz: Available in most countries worldwide.

Another issue that arises because of the limited band space is that incompatible technologies share the same band. This result in different technologies competing to gain and maintain access to the network, which leads to several performance problems.

The network topologies used by WSNs are designed to solve the problems of limited range and that the network needs to be low maintenance. The limited range problem is solved by using multihop network topologies that form a communications mesh. To solve the requirement for a low-maintenance network, the network should be designed so each sensor in the network can be developed with a low duty cycle operation.

The last high-level issue for designing and implementing a WSN is that the network

needs to be self-organizing. Each sensor in the network should be able to start participating in the network without any configuration from a second party and a suitable routing protocol should be used in the network to determine an appropriate message path from a source to a destination.

2.3 Erlang

Erlang was developed at Ericsson from 1987 with the aim of improving the development of telephony applications. The first version of Erlang was implemented in Prolog but this interpreter was far too slow. So, in 1992 the development of the BEAM was started and it now compiles Erlang code to bytecode that can then be executed on the BEAM virtual machine (Armstrong, 1997).

The Erlang language was developed to support distribution, concurrency and fault-tolerance and it is a general-purpose, concurrent, functional programming language. It also has a garbage-collected runtime system (Armstrong, 1996).

To avoid side effects the Erlang language only supports single assignment variables and immutable data. Like other functional languages recursive functions are used instead of loop constructs. Erlang is also a declarative language, where instead of saying how something should be computed, the programmer describes what should be computed. An example of declarativity in the language is the use of pattern matching to distinguish between message types.

One distinctive characteristic of pattern matching in Erlang is that even though it is a high-level language it is possible to pattern match on binary data. This feature was included because the language was designed for embedded systems and it makes it possible to implement high-level descriptive functions for packet manipulation or the design of low-level communication protocols (Sivieri, 2012).

Another characteristic feature of Erlang is that an Erlang process is lightweight. This means that very little computational power is needed to create and destroy a process. In the Erlang language there are primitives that makes it easy to spawn new processes and because processes do not correspond directly with system processes or threads but are handled by the VM, it is possible to run many thousands of processes at the same time without degrading the performance of the system (Armstrong, 2003).

Each process in Erlang has a share-nothing semantics. This means that there is no memory sharing between CPUs, nodes do not share storage and the only way to communicate is by message passing. Each process has a mailbox that receives messages when a process sends a message to the pid of that process. These messages are also the only way to achieve synchronization between processes in an Erlang program (Armstrong, 2003). By not having any shared data it is easy to create a distributed program by assigning a parallel process to another machine and the code

can be highly efficient because no semaphores or mutexes are needed (Armstrong, 2003).

By using a mechanism that is called process linking it is easy to make a system fault-tolerant. Process linking is when a process spawns a new process and a bidirectional relation is established between the processes. Then if one of the processes exits a special kind of message called a exit message is propagated over the link to the other process. If the exit message is an error message the process that observes a linked process can perform error recovery (Armstrong, 2003). The observing process that performs the error recovery procedure is called a supervisor and every observed process can be defined with a specific restart behaviour to handle error recovery (Armstrong, 2003; Sivieri, 2012).

Erlang also supports hot code swapping. This allows Erlang models to be updated on the fly without the need to stop the system while making sure that the old code terminates gracefully (Sivieri, 2012).

2.4 IEEE 802.15.4

The communication standard that is usually used in the Internet is the TCP/IP architecture. The architecture consists of five functional layers: the physical, data-link, network, transport and application layers. When data is passed between these layers extra framing and control data is added to the main data. This added data requires extra processing power and memory capacity and even more memory and processing power are consumed because of buffering of packets between the different layers (Bello et al., 2017).

One additional limitation of the TCP/IP protocol is that it is not designed to manage device-to-device communication. It has no support for the high level of scalability, high amount of traffic and mobility that can be found in WSNs (Bello et al., 2017).

One of the most commonly used communication standards to solve this problem is the IEEE 802.15.4 standard (Yang, 2014). The IEEE 802.15.4 standard was first published in the year 2003 as a low-rate Wireless Personal Area Network and it was developed to provide wireless connectivity in a low-complex, low-cost and low-power manner.

2.4.1 Components

An IEEE 802.15.4 network consists of two types of devices, full-function devices (FFDs) and reduced-function devices (RFDs). The personal area network (PAN) coordinator is a FFD device that has been assigned to serve as the central device in a network and is responsible for starting and managing the network. An FFD device can freely communicate with all devices within range. The RFD device is

a restricted device and can only communicate with its parent FFD device (Yang, 2014).

2.4.2 Network topologies

The IEEE 802.15.4 standard defines two network topologies, a star topology and a peer-to-peer topology. Both topologies are shown in Figure 2.1 and both topologies needs to have a PAN coordinator (Kohvakka et al., 2006).

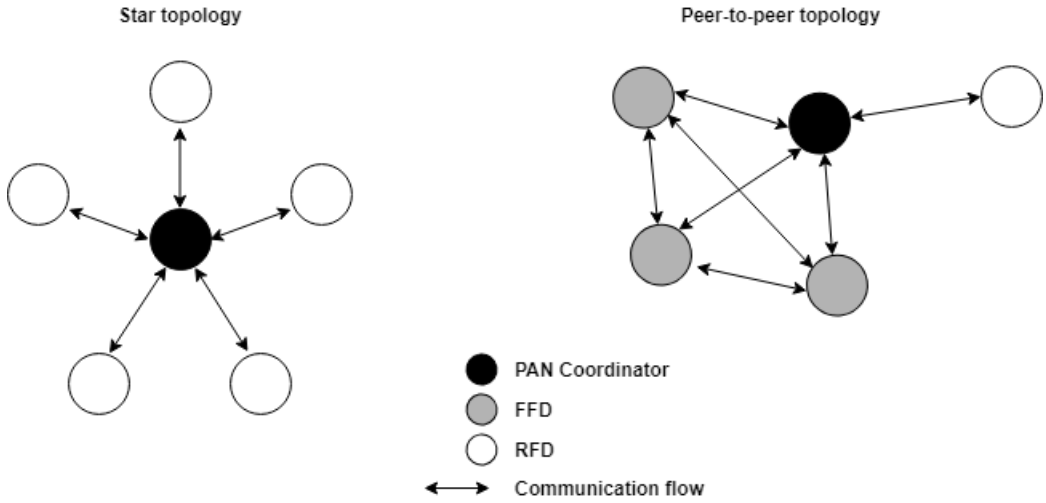

Figure 2.1: IEEE 802.15.4 topologies

The difference between a star network and a peer-to-peer network is that in a star network the PAN coordinator is the master node. All slave nodes of the network can only communicate with this master node (Kohvakka et al., 2006). This is achieved when an FFD device is initiated as a PAN coordinator and a unique PAN identifier in the current radio range is selected. When other devices associate with the network, they get the same PAN identifier and can only communicate with the PAN coordinator of the network with the same PAN identifier (Institute of Electrical and Electronics Engineers, Inc., 2003).

In a peer-to-peer network all devices are allowed to communicate with any other device in the network. This is suitable for networks where self-organizing, self-healing and large coverage by allowing multiple hops to route messages is an advantage. A disadvantage is that the network latency increases due to the message relaying (Kohvakka et al., 2006).

2.4.3 Architecture

The IEEE 802.15.4 standard defines the two first layers in the network stack, the Physical (PHY) and Medium Access Control (MAC) layers.

PHY layer The PHY layer contains the radio frequency transceiver and the low-level mechanisms that are needed to operate the transceiver. It has two different modes. One mode operates in the frequency range 868 – 868.6 MHz in Europe or 902 – 928 MHz in America and the other mode at 2.4 GHz worldwide (Institute of Electrical and Electronics Engineers, Inc., 2003; ZigBee Alliance, 2012). The second mode has the most potential for WSNs, because it has a higher data rate which leads to a reduced frame transmission time and a reduced energy per transmitted and received bit (Kohvakka et al., 2006).

IEEE Computer Society (2016) defines the features of the PHY layer as follows:

- Activation and deactivation of the radio transceiver.
- Energy detection within the current channel.
- Link quality indicator for received packets.
- Clear channel assessment for carrier sense multiple access with collision avoidance (CSMA-CA).
- Channel frequency selection.
- Data transmission and reception.
- Precision ranging for ultra-wide band PHYs.

MAC layer

Two services are provided by the MAC layer. The first is the MAC data service which provides functionality that makes it possible to transmit and receive MAC protocol data units across the PHY data service. The second service is the MAC management service. It provides an interface to the MAC sublayer management entity (MLME) service access point (IEEE Computer Society, 2016).

IEEE Computer Society (2016) defines the features of the MAC layer as follows:

- Beacon management.
- Channel access.
- GTS management.
- Frame validation.

- Acknowledged frame delivery.
- Association, and disassociation.
- Hooks for implementing application-appropriate security mechanisms.

2.5 ZigBee

On top of the IEEE 802.15.4 standard are built the most commonly used protocols in WSN applications, 6LoWPAN and ZigBee. 6LoWPAN gives sensors the possibility of being accessed from the Internet, whereas ZigBee cannot do this because it lacks native IP stack processing. The protocol stack for 6LoWPAN and ZigBee can be seen in Figure 2.2.

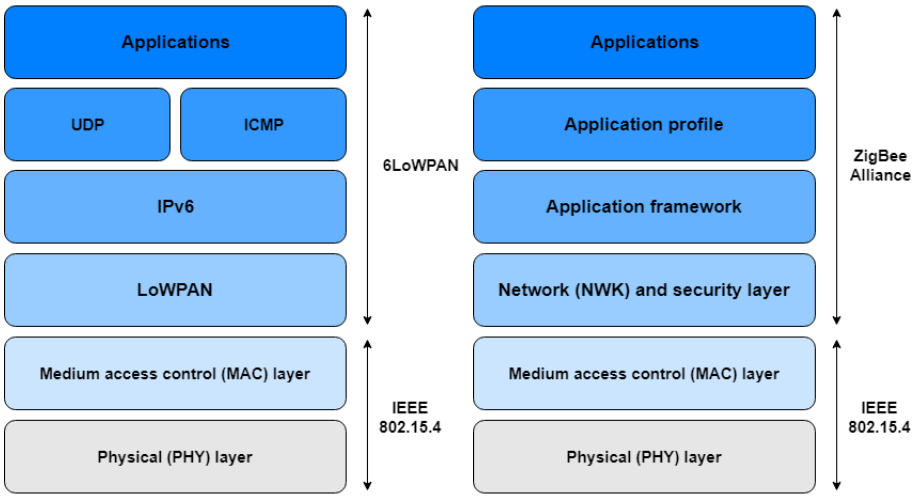

Figure 2.2: 6LoWPAN stack and ZigBee stack on top of the IEEE 802.15.4 stack

One feature of ZigBee is that it is designed to be a low-power wireless technology. To give an overview of this Table 2.1 shows a comparison of the characteristics between ZigBee, Bluetooth and WiFi.

Table 2.1: ZigBee, Bluetooth, and WiFi comparison (DIGI, 2019; Yang, 2014; ZigBee Alliance, 2019)

	ZigBee (IEEE 802.15.4)	Bluetooth (IEEE 802.15.1)	WiFi (IEEE 802.11)
Application	Control and monitoring	Cable replacement	Wireless LAN
Frequency bands	2.4 GHz, 868 and 915 MHz	2.4 GHz	2.4 GHz
Battery life in days	100–700	1–7	0.1–5
Nodes per network	65,000	7	30
Bandwidth	20–250 kbps	1 Mbps	2–100 Mbps
Range in m	1–300	1–10	1–100
Outdoor line-of-sight range in meters	3200		
Topology	Star, tree, cluster tree, mesh	Tree	Tree
Standby current in Amps	$3 * 10^{-6}$	$200 * 10^{-6}$	$20 * 10^{-3}$
Memory in KB	32–60	100	100

2.5.1 Network topology

The network topologies available in ZigBee are based on the star and peer-to-peer topologies specified in IEEE 802.15.4. Based on these topologies the NWK layer of the ZigBee stack supports star, tree and mesh topologies (Farahani, 2011).

The star topology is the simplest topology to form. When an FFD device that is programmed to be a PAN coordinator starts, it establishes a network. Every device that wants to join the network must join the network through the PAN coordinator. Because of this the star topology is not suitable for applications that require a larger area than the radio range of the PAN coordinator (Yang, 2014).

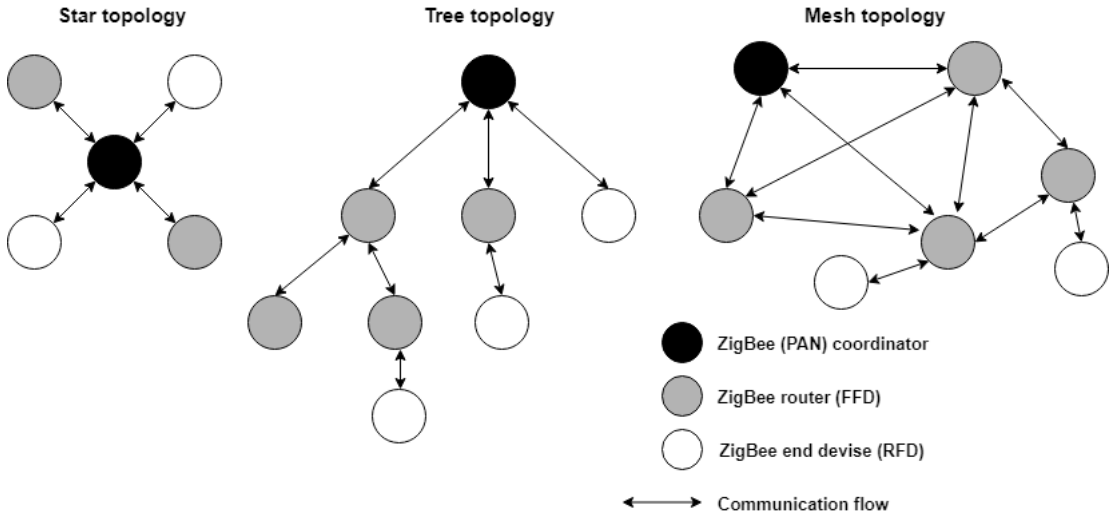

Figure 2.3: ZigBee network topologies

The peer-to-peer topology allows for various network shapes. In a tree topology (see Figure 2.3) a PAN coordinator creates the network, FFDs form the branches of the tree and RFDs are the leaves. The difference between a tree topology and a mesh topology (see Figure 2.3) is that a tree topology restricts the communication between FFDs, but in a mesh topology every FFD can communicate with all other FFDs in radio range. Figure 2.4 shows how a peer-to-peer topology can extend the range of the network and even circumvent barriers (Farahani, 2011).

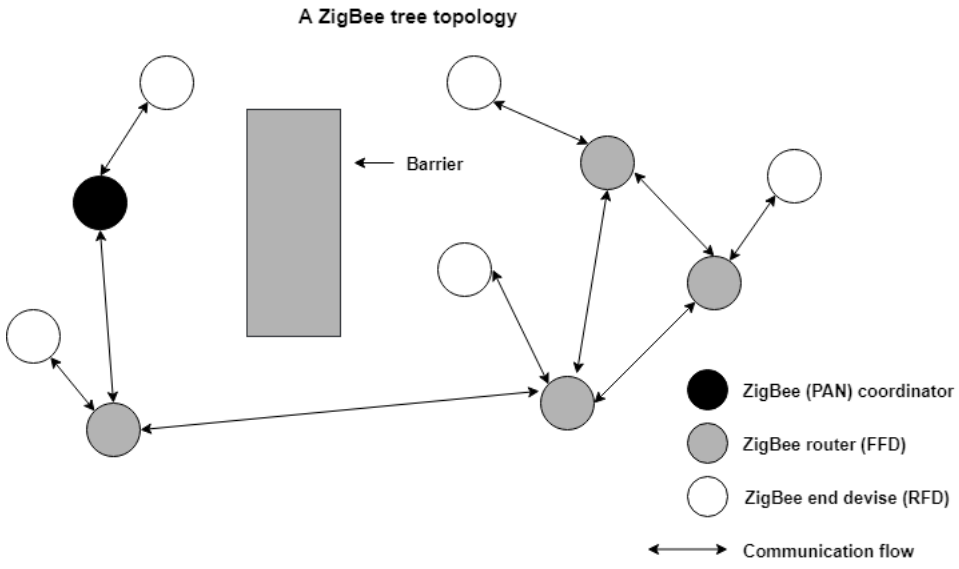

Figure 2.4: A ZigBee network topology with a barrier.

2.5.2 Architecture

The architecture of ZigBee can be seen as a set of blocks, which are usually called layers. Each layer is tasked with performing a specific service for the layer above; lower layers have no knowledge about upper layers. Between each layer there are two Service Access Points (SAPs) that isolate the layer. One of these SAPs provides a data transmission service and the other provides a management entity service that controls all other services in the attached layer by exposing an interface for the layer above (Gislason, 2008; ZigBee Alliance, 2012). An image that represents the architecture of the ZigBee stack can be found in Figure 2.5.

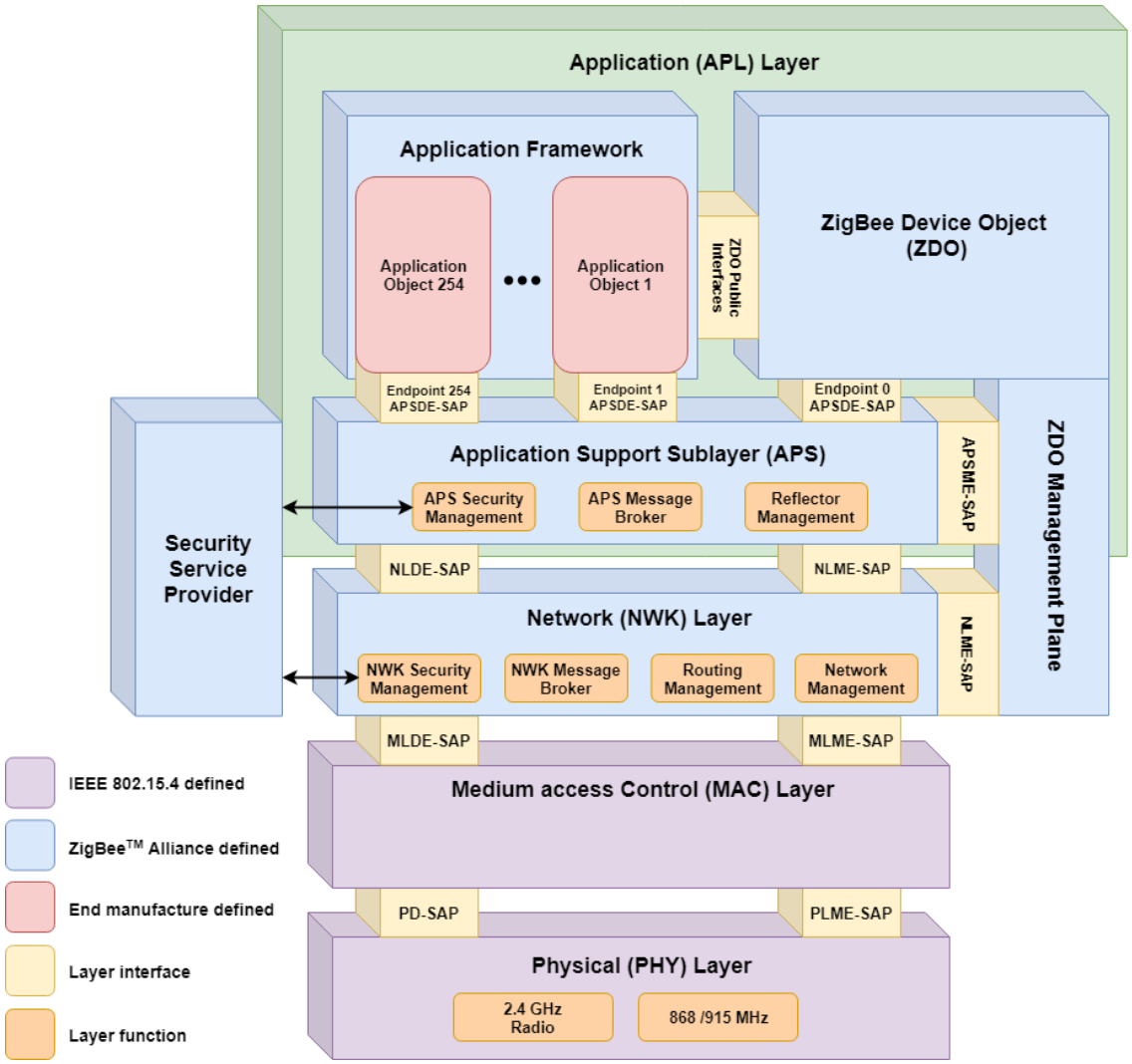

Figure 2.5: Image of the ZigBee stack architecture based on ideas from Gislason (2008); Yang (2014); ZigBee Alliance (2012).

Application layer (APL)

The APL consists of the Application Support Sublayer (APS), ZigBee Device Object (ZDO) and manufacture-defined application objects (ZigBee Alliance, 2012).

The APS is the layer above the Network (NWK) Layer and provides the NWK layer with an interface to the APL. The APS's task is to filter out packets whose endpoints do not exist, manage transmission retries with end-to-end acknowledgment, maintain the local binding table which handles the connection of an endpoint on the current node with one or more endpoints on other nodes, maintain the local groups table which makes it possible to send group-addressed frames to endpoints associated with the same group and maintain the local address map which handles the association between a 64-bit MAC address and a ZigBee 16-bit network address (Gislason, 2008; ZigBee Alliance, 2012).

The ZDO is an application that runs on endpoint 0 on every ZigBee device. It includes the ZigBee Device Profile, which is a specialized application profile that is responsible for discovering, configuring and maintaining ZigBee devices and services on a network. The ZDO application also directly interacts with the NWK layer, by controlling when to create a network or join a network and when to leave a network (Gislason, 2008).

Manufacture-defined application objects reside in the Application Framework (AF). The AF also contains the ZigBee Cluster library and the task of this library is to provide a framework for running applications where each application has a unique endpoint (Gislason, 2008).

Network (NWK) layer

The task of the NWK layer is to connect the above layers with the MAC sub-layer. To connect with the APL, the NWK layer, like all other layers, also provides two SAPs as described in Section 2.5.2. These SAPs are responsible for (Farahani, 2011; ZigBee Alliance, 2012):

- Transporting protocol data units to their intended recipients.
- Providing security that ensures both the authenticity and the confidentiality of a transmission.
- Self-configuration of the stack for either starting a network as a ZigBee coordinator or joining a network.
- Creating a new network.
- Making it possible for devices to join, rejoin and leave a network. This also includes the ability for a ZigBee coordinator or router to request that another device leave the network.
- Address assignment of a device by a ZigBee coordinator or router.
- Discovering, recording and reporting information about devices directly neighbouring to the current device.

- Discovering and recording paths for sending messages.
- Controlling when the recipient device is active and for how long and hence enabling MAC sub-layer synchronisation or direct reception.
- Providing routing mechanisms such as unicast, broadcast, multicast and many to one to exchange data in the network efficiently.

Security service provider

The security service provider is responsible for services that provide encryption for data confidentiality, device and data authentication and replay protection. These security measures are optional, and it is up to the developer to chose if they should be used (Farahani, 2011).

2.6 Further recommended reading

This section presents related work about a variety of other topics: how to modify the Erlang runtime system for WSN applications, comparisons between Erlang-based languages and other languages, how functional languages can reduce security risks, different device-to-device communication techniques, the power consumption of ZigBee, and how a functional language can be used to generate nesC code.

Sivieri and Cugola (2012) and Sivieri et al. (2016) investigate how the Erlang runtime system could be modified for WSN applications. Sivieri and Cugola (2012) write that WSN-Erlang gives a higher level of programming abstraction which makes it easier to produce more reusable, maintainable code and makes it easier to test code that may run on heterogenous networks. They also strip the runtime system of unnecessary libraries and facilities to reduce the memory, storage and processing requirements.

Sivieri et al. (2016) is in some ways a continuation of the work done in Sivieri and Cugola (2012). The development platform ELIoT is presented and a comparison is performed between a C implementation, a Java implementation and an ELIoT implementation of a smart-home application. They find that ELIoT makes it possible to develop more concise and readable code that is easier to test and debug. They also show that CPU and memory consumption are accaptable for their application.

Fedrecheski et al. (2016) compare the performance of a Swarm Broker implementation in Java and Elixir. Elixir is a language with the goal of leveraging all the abstractions of Erlang, whilst at the same time adding new features from other programming languages (Thomas, 2018). They conclude that the Java implementation of the Swarm Broker application uses slightly less CPU. However, Elixir shows better memory usage and the number of lines of code is markedly less.

Haenisch (2016) investigates the use of functional languages for improving the security of IoT applications. The paper shows that the use of functional languages or functional techniques can reduce the code size and complexity of an application, resulting in fewer bugs and fewer security problems.

In this thesis ZigBee was used for device-to-device communication but there are alternatives to this solution. Essameldin and Harras (2016) and Militano et al. (2015) investigate different techniques for device-to-device communications in the field of IoT. Militano et al. (2015) also discuss the main challenges and the coming research directions that need to be investigated to reach what is expected to be the reality for IoT, a device-oriented Anything-as-a-Service ecosystem, in the fifth generation (5G) cellular systems.

The performance of the IEEE 802.15.4 low-rate wireless personal area networks is another field that has been investigated. Kohvakka et al. (2006) analyze the IEEE 802.15.4 standard MAC protocol to investigate the network performance and energy efficiency.

Mainland et al. (2008) show that there are other approaches than the one presented in this thesis to make it possible to use functional programming for devices with a constrained availability of resources in a WSN scenario. The paper presents Flask, a domain specific language embedded in Haskell that generates nesC (a dialect of C) code for the operating system TinyOS.

3

Methods

In order to evaluate the performance of using Erlang to communicate in a WSN two prototypes were developed and data was collected from them. The data was then analysed in order to show when Erlang can be an alternative to a low-level language in a WSN.

3.1 Setup

The prototypes that have been developed are prototypes of a video-based sensor network. A basic idea of how the network has been constructed can be seen in Figure 3.1. This figure shows how the network looks for the experiments performed in this thesis, but the network design of the final product may not necessarily look like this.

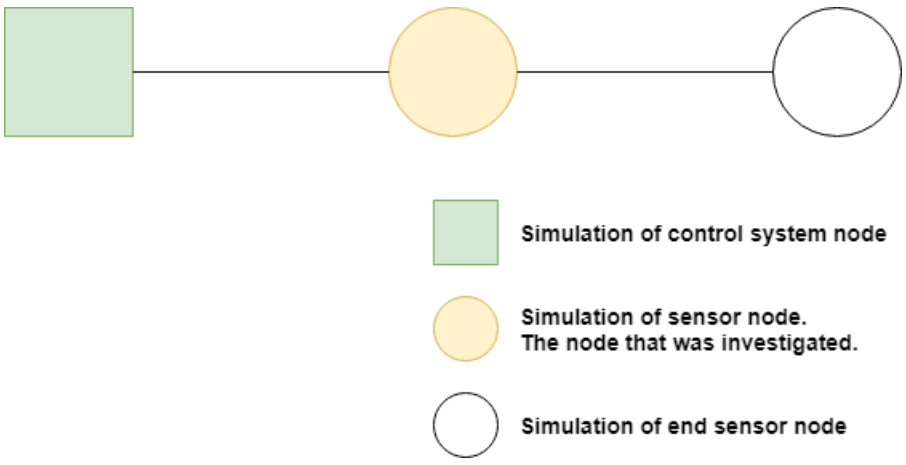

Figure 3.1: The design of the video-based sensor network for this experiment.

There can be arbitrarily many sensor nodes in the network but there is no need for the network to be bigger than the one shown in Figure 3.1 for the investigation in this thesis. This is because the sensor that has been investigated both receives data from another sensor and sends data to another device, both of which are performed

by the middle sensor node in this setup. All measurements presented in this thesis are from the node in the middle of Figure 3.1.

The hardware used for the sensor node is a Freescale i.MX 6Quad SABRE Development Board with a 1GHz ARM Cortex A9 core and 1 GB DDR3 SDRAM with up to 533 MHz memory (NXP, 2012). The investigation does not focus on the other devices in the network and therefore there are no specific requirements on these devices except that they need to be fast enough to keep up with the sensor node. Therefore, the control system node and end sensor node were simulated using two computers.

To achieve communication between the devices in the network ZigBee was used. The i.MX 6 board does not have built in support for the IEEE 802.15.4 standard so therefore a module was needed to add support to the i.MX 6 board. The experiments in this thesis used modules from the XBee ZigBee Mesh Kit (XKB2-Z7T-WZM). This kit consists of three XBee ZigBee modules and three Grove development boards. By mounting the Xbee ZigBee modules in the Grove development boards and connecting the boards by USB to the processing unit, the processing unit can communicate with the connected XBee ZigBee module.

There are many possible choices of operating system for the prototype. Because the company uses a basic Debian distribution in their other products and wants to continue using it, this operating system was chosen.

3.2 Prototypes

In order to compare Erlang against a low-level language for implementing the communication between sensors in a WSN, two prototypes were implemented. The first prototype was developed in C++ and the second was developed in Erlang.

The first step in the prototypes' development process was to decide on a design: how the sensors in the network should communicate and how the prototypes should be structured. For example, should a sensor ask for its child sensor's data or should the child sensor pass data to its parent without being asked for it? For the C++ prototype, the choice was made to follow the company's intended design for the final product: multiple processes communicating over D-Bus. The Erlang prototype followed the same design except that it used message passing instead of D-Bus.

After this first decision was made both prototypes were developed iteratively with features being added until the prototypes were realistic enough to perform the final comparison. This iterative approach was chosen to give the possibility of catching problems early and to only implement as much of the communication code as was needed to carry out the evaluation.

3.2.1 C++ prototype

The C++ prototype is divided into three processes. The task of the first process, the Writer process, is to receive the data produced by the other two processes and to send it to the ZigBee module for transmission to the correct recipient. The second process, the Datagen process, simulates the generation of data and the last process, the Reader process, is responsible for collecting incoming data from another device in the network. The reason why the Writer process exists and the other processes do not directly write the data to the ZigBee module is twofold. Firstly, the Datagen process is going to be vastly different in the final product, perhaps even run on a FPGA unit. Secondly, only one process is allowed to write to the serial port at a time.

The prototype and ZigBee module communicate through a serial port and the settings that the port needs to be configured with can be found in the implementations of the C++ prototype found in Section A.4.

The Datagen and Reader process communicate data to the Writer process using D-Bus. D-Bus is a software bus that provides inter-process communication and a remote procedure call mechanism that allows processes to communicate with each other. D-Bus provides two daemons, a system daemon that handles events sush as when a device is added to the system and a per-user-login-session daemon that handles general inter-process communication between applications. The Writer process implements functionality that listens to incoming messages from the Datagen and Reader processes. The implementation of these processes can be seen in Appendix A.

The final consideration in the development of the C++ prototype was fault-tolerance. If one of the processes crashes it should be restarted automatically. This is achieved using systemd, a system and service manager and initialization system which is commonly used on Linux. By turning the processes into services, systemd can restart them following the provided specifications, which can be seen in Appendix A.5.

3.2.2 Erlang prototype

According to the Erlang documentation it is very hard to implement a new driver for the distribution carrier and therefore for this thesis the choice for the connection between Erlang and a ZigBee module was between implementing a port and a port driver (Ericsson AB, 2019a).

Ports, port drivers, C nodes and Native Implemented Functions (NIFs) are different methods that are provided by Erlang for applications to run C code. The difference between these methods is in how they are loaded. For example, NIFs are dynamically linked into the emulator process and port drivers are loaded as a shared library. Therefore, these methods of loading the C code cause the emulator to crash if the C code terminates. On the other hand, a port is loaded into a separate process and

if it crashes the emulator does not crash. A depiction of the difference between a port and a port driver can be seen in Figure 3.2 (Ericsson AB, 2019b).

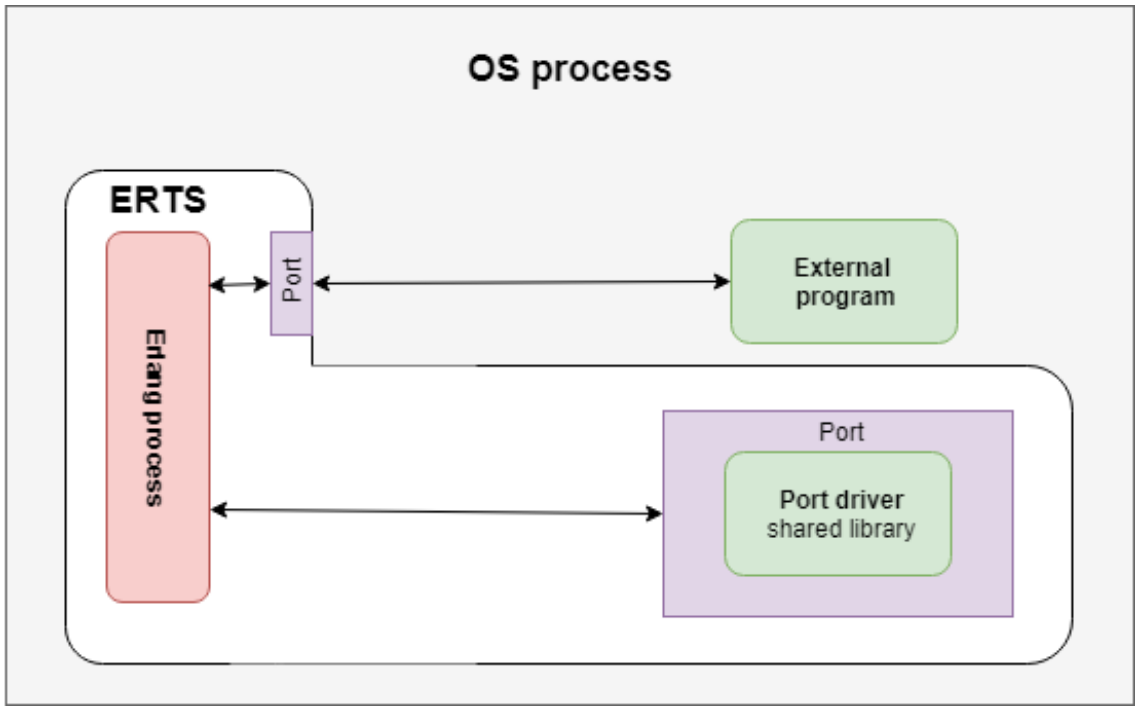

Figure 3.2: The difference between a port and a port driver.

Since a port driver crashing causes the emulator to crash, the choice was made to use a port. For the implementation of the port much of the code from the C++ prototype could be reused and the full implementation can be seen in Appendix B.1.

The Erlang prototype has a similar structure to the C++ prototype, consisting of a Writer, Reader and Datagen process. One difference is that instead of using systemd for restarting crashed processes, the built-in supervisor behaviour of Erlang is used instead. By building a supervisor tree it is possible to specify how to restart the system when if a process crashes. A model of the supervisor tree is shown in Figure 3.3 and the full implementation of the prototype can be seen in Appendix B.

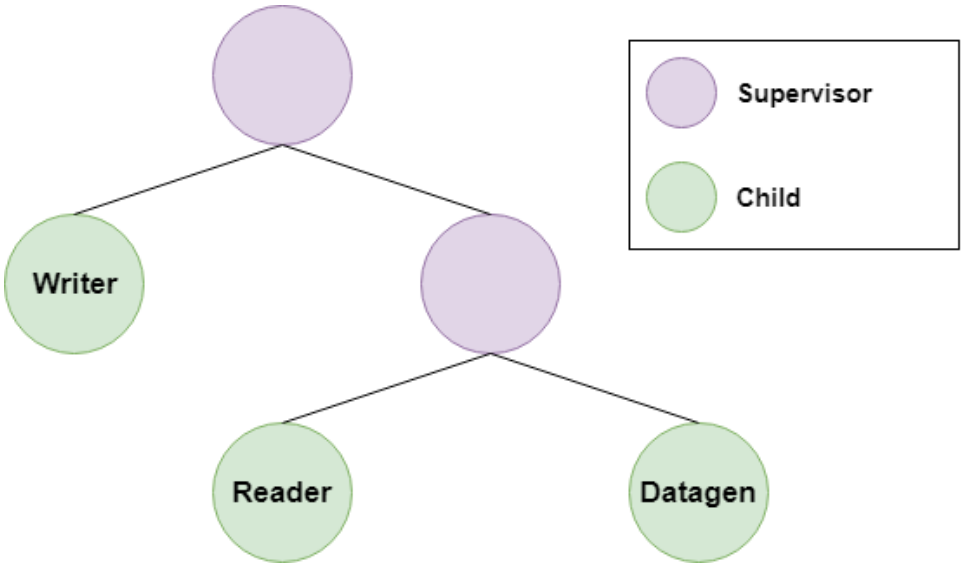

Figure 3.3: The structure of the Erlang prototype.

The behaviour that was wanted of the prototype was to restart all processes if the Writer process crashes and only restart the crashed process if it is not the Writer process. With this supervisor tree and by specifying the restart procedures for the supervisor nodes it is possible to get the desired behaviour.

The specified restart procedure for the root supervisor is that if one of the processes crashes all processes are restarted. This choice was made for two reasons. Firstly, if the Writer process has crashed the work done by the Reader and Datagen process is wasted because there is no recipient for the data they send out. Secondly, to send messages to the Writer process the other processes need to know the process id of the process. To achieve this, the Writer process registers its process id in the global process registry so that it can be addressed by name. If a process tries to send a message to the Writer's registered process id when the Writer has crashed it also crashes. The second supervisor only supervises children that have no dependent processes. Therefore, the restart strategy is to only restart the child that has crashed.

3.3 Data collection

Data was collected from the two prototypes for the purpose of evaluating the performance of using Erlang in a WSN. The collected data was in all instances quantitative data.

For the Erlang prototype some virtual machine flags were used to minimize the memory use. The used flags were:

- **+P 1024**: set the maximum number of processes to 1024.

- **+Q 1024**: set the maximum number of number of simultaneously existing ports to 1024.
- **+L**: turn off the loading of source filenames and line numbers.
- **-noshell**: starts the runtime system without a shell.
- **+Mea min**: use one global memory allocator instead of multiple ones.

To count lines of code, the SLOCCount tool created by David A. Wheeler was used. The tool counts physical Source Lines of Code (SLOC), defined as a line of code that contains at least one non-whitespace non-comment character (Wheeler, 2019). This means that comments and empty lines are not included in the count. By counting the lines of code data was collected that was used to do a quantitative evaluation of the implementation effort of each prototype.

Static code size of the prototypes was measured by using the *ls -l "filename"* command and the static code size of the Erlang runtime system was measured by using the Erlang function *erlang:memory(code)*.

To measure the memory and CPU usage of the prototype a C program was implemented. For the full implementation of the program see Appendix C. This program executes each prototype ten times for all data generation rates, where each generated data packet has a size of 10 bytes. As described in Section 1.3, each sensor is not intended to send huge amounts of data. Therefore, data packets of 10 bytes are realistic in this case. Each time a prototype is executed it runs for two minutes and a measurement is performed each second.

The measurement code performs the measurements by using the Linux command *ps* and *free* and the file */proc/[pid]/stat*. By using these commands data was collected about memory and CPU usage of the prototypes.

The *ps* command was used to collect resident set size (RSS) and virtual memory size (VSZ) usage of each prototype for all data generation rates. The measurements was performed multiple times and the average value of these results was used to show memory usage. RSS memory is the most interesting value of these two because it represent the amount of used RAM memory. This value can be somewhat misleading because shared libraries are only loaded once but counted in the RSS of all processes that use them. VSZ is not as important as the RSS value because it is the total accessible address space of a process and that includes memory that is swapped out, allocated memory that is not used, and memory from shared libraries.

The *ps* command was also used to collect information about the CPU usage. The CPU value that is reported by the *ps* command is the average CPU usage from the time the program started to the point of the measurement. Therefore, only the last collected CPU value is used.

The *free* command also reports on memory use but in a different manner. The reported values that are used in this thesis are:

- Free, a value that reports the free memory of the system.
- Used, a value that reports the used physical memory of the system excluding kernel buffers, page cache and slab memory.
- Available, a value that represents an estimation about how much memory is available for starting a new process in the system.

Finally, CPU usage information was read from the file */proc/[pid]/stat*. This information was read multiple times with one second between two measurements during the whole execution of a prototype. The fields that record CPU usage are called *utime* and *stime*. Utime shows how much scheduled time a process has had in user mode and stime shows how much scheduled time the process has had in kernel mode. By using the following formula with two subsequent measurements it is possible to calculate how much CPU time a process uses between two measurements:

$$((utime - old\_utime) + (stime - old\_stime))/(time - old\_time)$$

The average of these values was then used to calculate a CPU usage value for the whole running time of the prototype. The CPU usage is a relevant measure of performance in this case because the data generation rate is constant. As long as the data generation rate is constant and the CPU usage is below 100%, the CPU usage indicates how much time is needed to process the packets generated during one second.

Lastly, data was collected about the power consumption of the prototype. This was done by placing a multimeter between the board and the power supply. Then for each execution of a prototype, for each data generation rate, a measurement was collected every five seconds for two minutes.

3.4 Evaluation

The collected data was analysed as follows to show when Erlang can be used for communication in a WSN.

- Calculate the percentage of how much capacity is lost or gained by using Erlang. How many messages can be sent with an Erlang solution compared to a C++ solution if the power budget, RAM usage and CPU usage is fixed respectively or if more than one hardware resource is fixed at the same time?
- Compare the power consumption, RAM usage and CPU usage of the C++ prototype from the performed measurements against the Erlang prototype, in order to find the minimum hardware requirements for running the application part of the sensor and at which data generation rates each prototype performs better than the other.

These analyses give someone who wants to develop the communication between nodes in a WSN with Erlang a guideline for hardware requirements. For example, if someone wants to develop an WSN application and they have a fixed amount of RAM and some other process already running on the system, they can see if there is enough RAM for Erlang to be used for communication.

4

Results

This chapter contains the results of the comparison of the C++ and Erlang prototypes. The first five sections contain results about the number of lines of code, static code size, memory use, CPU usage and power consumption of the two prototypes. In the last section of this chapter some experiments are presented that explain a surprising result.

4.1 Lines of code

The C++ prototype has 352 lines of code and the Erlang prototype has 317 lines of code. This means that the C++ prototype has 10% more lines of code in total. This total percentage value is somewhat misleading because each prototype has parts that do not exist in the other prototype. Therefore, it is more interesting to look at each corresponding part of the prototypes separately.

The prototypes mainly consist of the Writer, Reader and Datagen processes. This part of the C++ prototype has 167 lines of code and in the Erlang prototype it has 45 lines of code. This part of the C++ prototype therefore has nearly three times more lines of code than the Erlang prototype.

To restart a process if it crashes, the C++ prototype uses scripts to turn the processes into systemd services and these scripts consist of 34 lines of code. The Erlang prototype instead uses the built-in supervisor behaviour. The code needed for setting up this functionality consists of 45 lines of code. The C++ prototype therefore uses 25% less code.

The communication with the ZigBee module is done over a serial port. The C++ prototype implements this functionality using 150 lines of normal C++ code but the Erlang prototype uses a port, implemented with 105 lines of code. This means that the C++ prototype has 50% more lines of code for this part. It should be noted that most of the extra lines of code in the C++ prototypes come from the header file.

The final part of the program is the code that implements the port functionality of

the Erlang prototype. This part does not have any corresponding part in the C++ prototype. In total this part consists of 106 lines of code where 32 lines are Erlang and 74 are C.

4.2 Static code size

Measuring the static code size as described in Section 3.3, the C++ prototype has a static code size of 50kB and the Erlang prototype has a static code size of 2138kB. Much of the static code size of the Erlang prototype comes from the code that is needed to load the runtime system, which in this case amounts to 2121kB. One thing to mention is that the runtime system can be modified by stripping unnecessary parts, for example unused modules from the standard library, to decrease the static code size. This was not investigated in this thesis.

In this case it is hard to do a fair comparison between the C++ prototype and the Erlang prototype because much of the static code size of the Erlang prototype comes from the implementation of the port that provides functionality to communicate with a ZigBee module. If the port is not included in the calculation the static code size for the BEAM files is only 7kB. This shows that if there are many more processes or the processes are bigger, there could be a point where the static code size becomes larger if C++ is used instead of Erlang.

4.3 Memory usage

Figure 4.1 shows the resident set size (RSS), which measures physical memory use, of both the C++ prototype and the Erlang prototype. The data shows that the Erlang prototype uses less physical memory for all data generation rates.

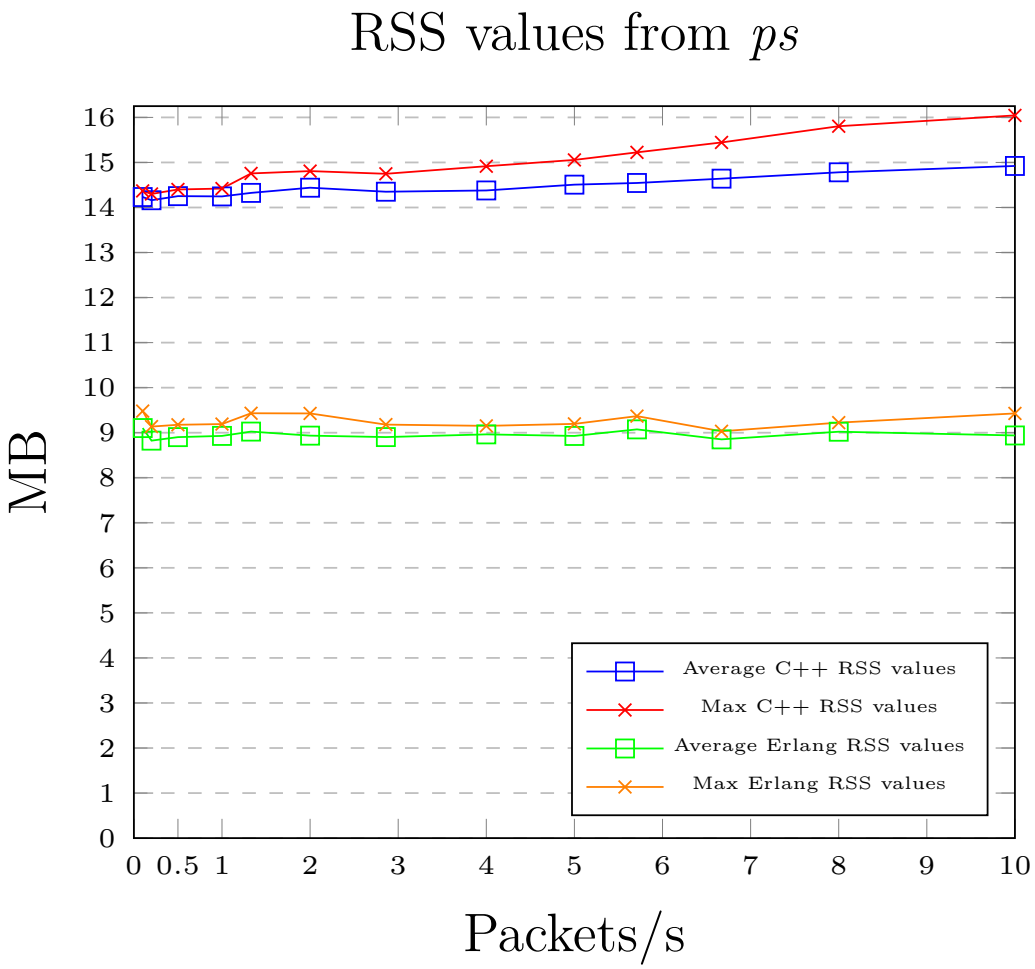

Figure 4.1: Data about physical memory usage collected with the *ps* command.

Figure 4.2 shows the average virtual memory size (VSZ) for the processes C++ and Erlang prototypes.

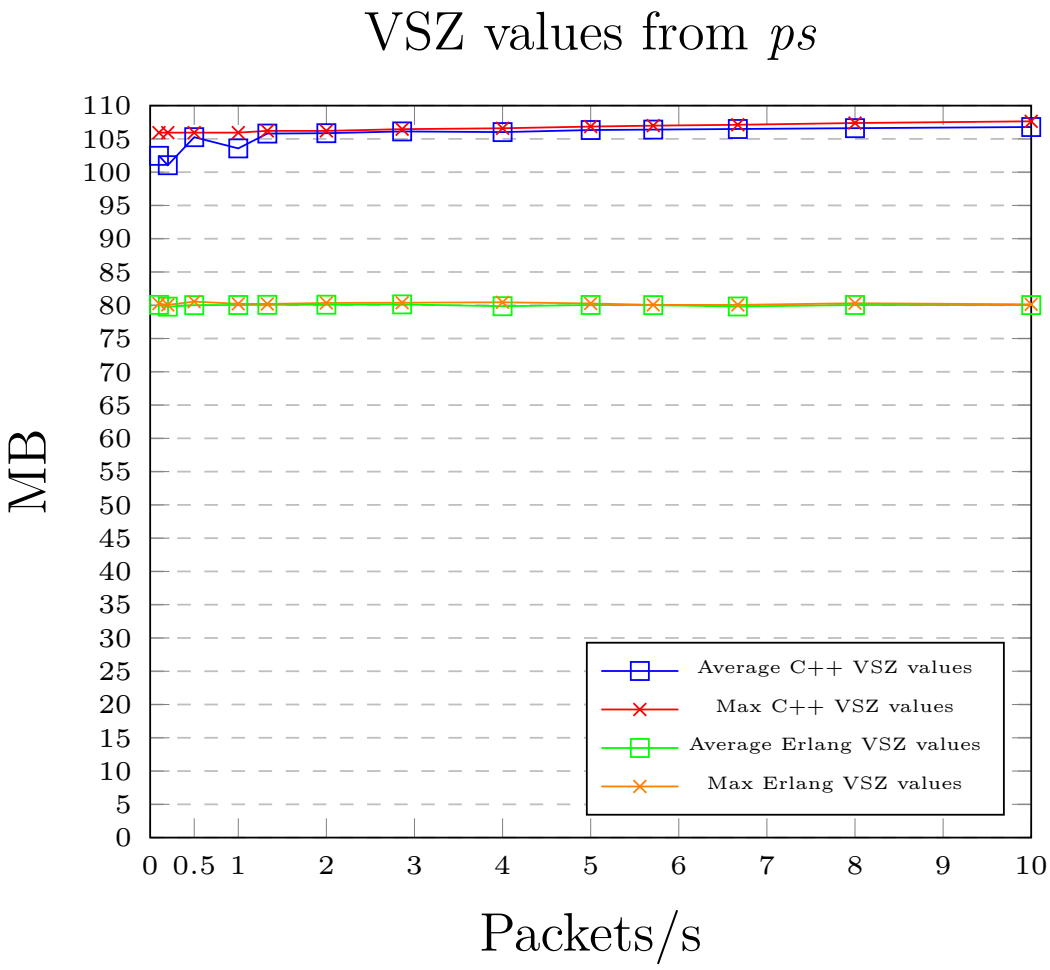

Figure 4.2: Data about virtual memory usage of the C++ and Erlang prototypes collected with the *ps* command.

Figures 4.1 and 4.2 show that the Erlang prototype uses less physical and virtual memory for all data generation rates. Figure 4.3 shows how many times greater the memory usage is for the Erlang prototype. If the value is below 1 it means that the Erlang prototype uses less of that type of memory than the C++ prototype.

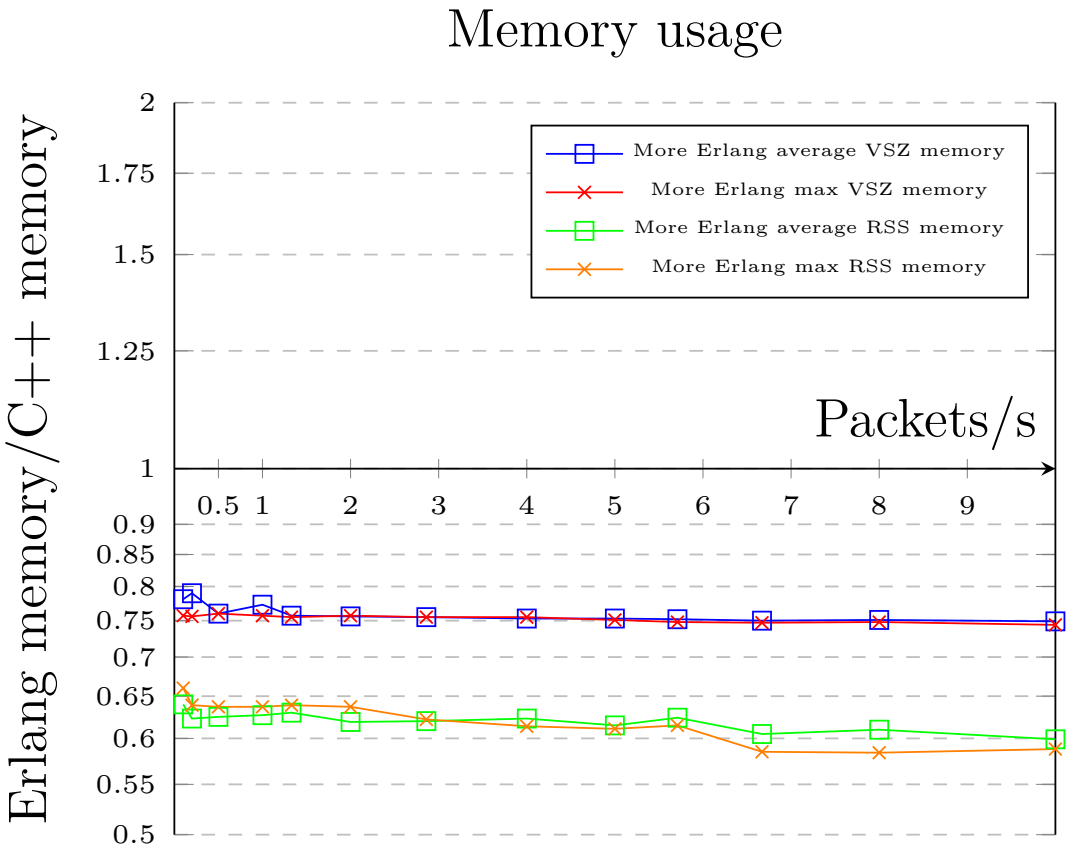

Figure 4.3: Ratio of Erlang to C++ memory use.

Figure 4.4 compares the values reported by the *free* command (used, free, and available) for the two prototypes. For each measurement, the value reported was subtracted by the value when the system was idle. As in Figure 4.3, a value below 1 means that the Erlang prototype has a lower figure for that type of memory.

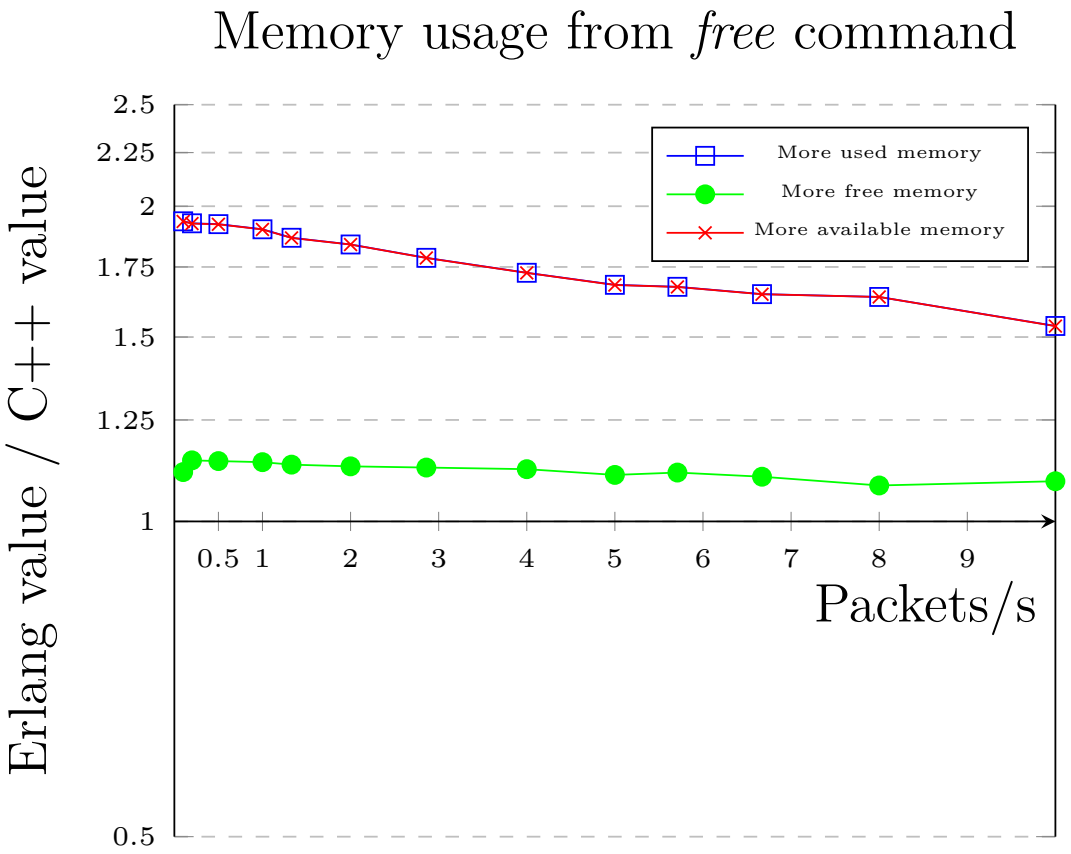

Figure 4.4: Ratio of Erlang to C++ memory use as reported by the *free* command.

4.4 CPU usage

The CPU usage was measured in two different ways. The first was with the *ps* command which gives the average CPU usage over the total execution of the program. This data can be seen in Figure 4.5.

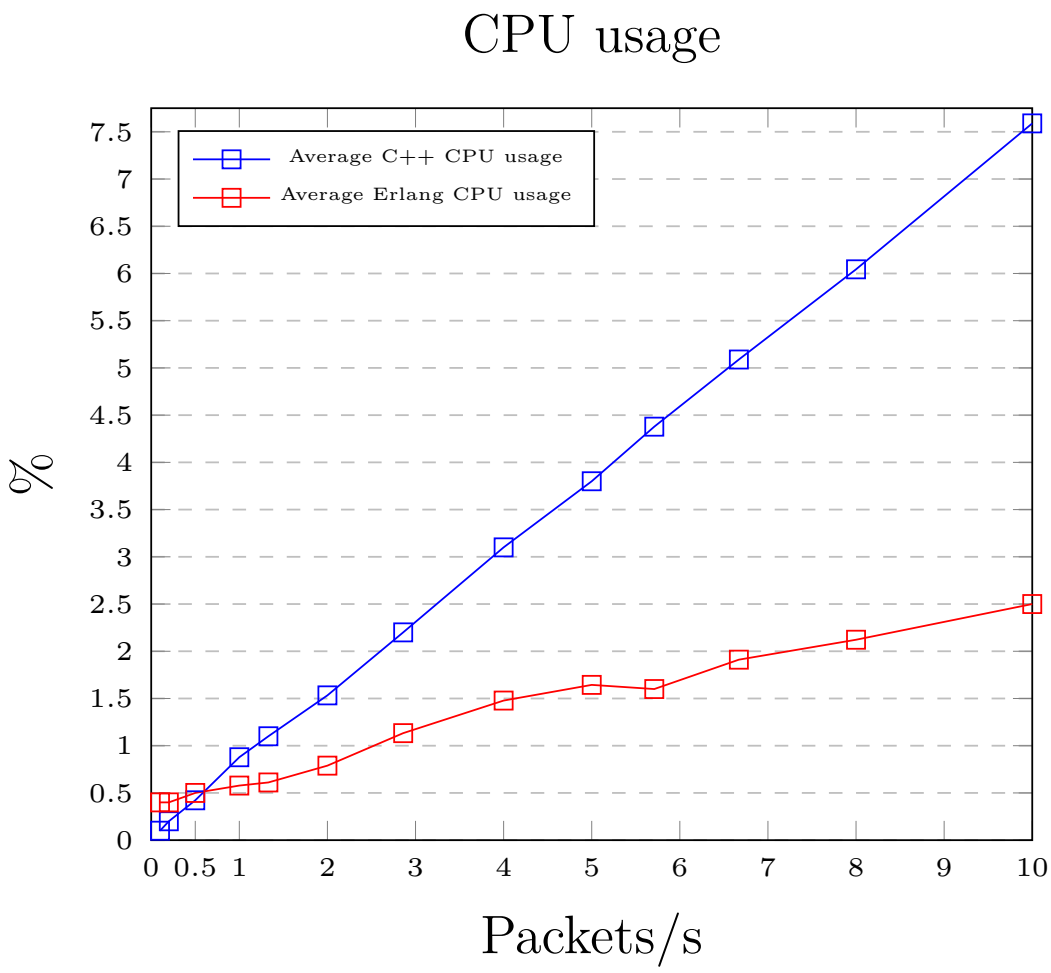

Figure 4.5: Collected data about CPU usage from the *ps* command.

The second way the CPU usage was measured was to take multiple samples with one second between samples and then calculate how much CPU time had been used between the samples. The result of this calculation can be seen in Figure 4.6. One interesting thing to notice is that the CPU usage for the C++ prototype results in a much straighter line than the Erlang prototype in both Figures 4.5 and 4.6. A possible explanation for this is that the Erlang runtime system uses garbage collection and this introduces unpredictability in how much work needs to be done.

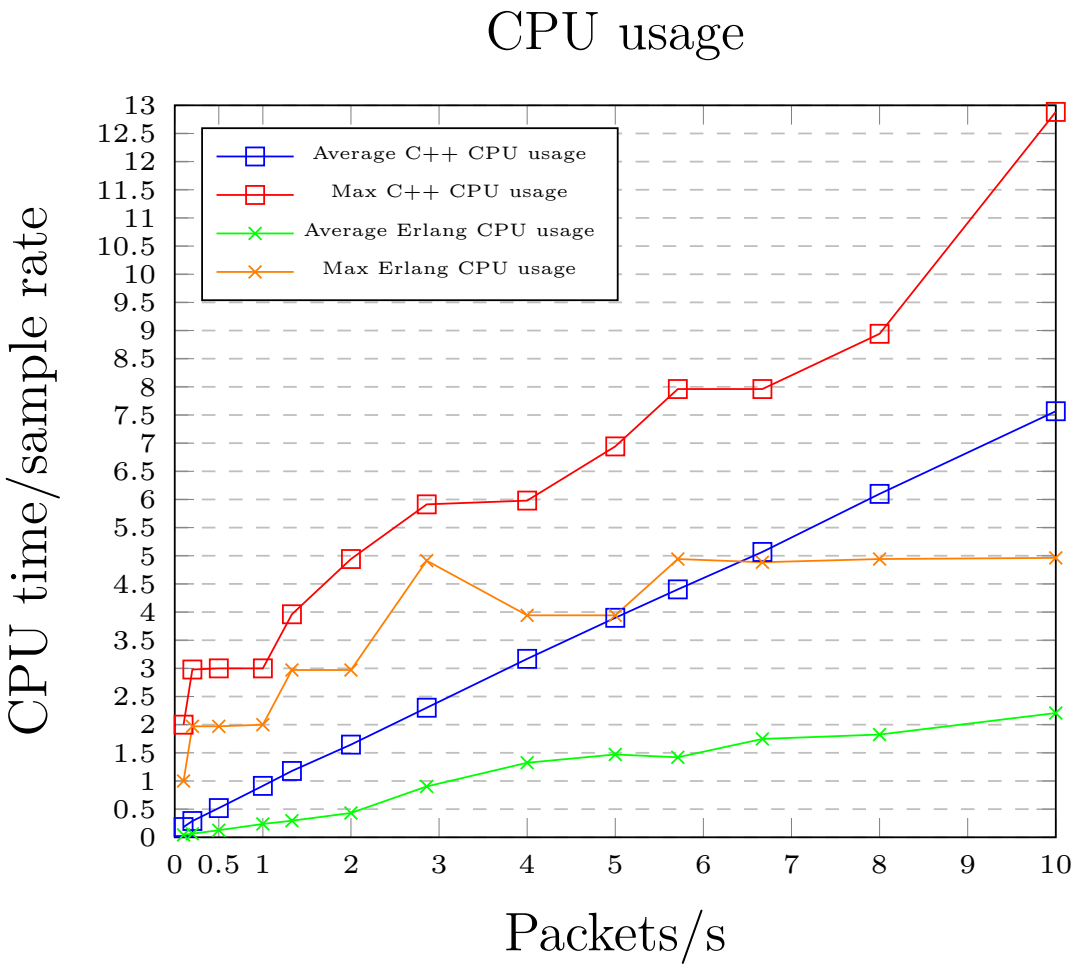

Figure 4.6: Data collected about CPU usage as described in Section 3.3.

Figure 4.7 compiles the data to show how much more the Erlang prototype uses the CPU. A value below 1 means that the Erlang prototype uses the CPU less than the C++ prototype.

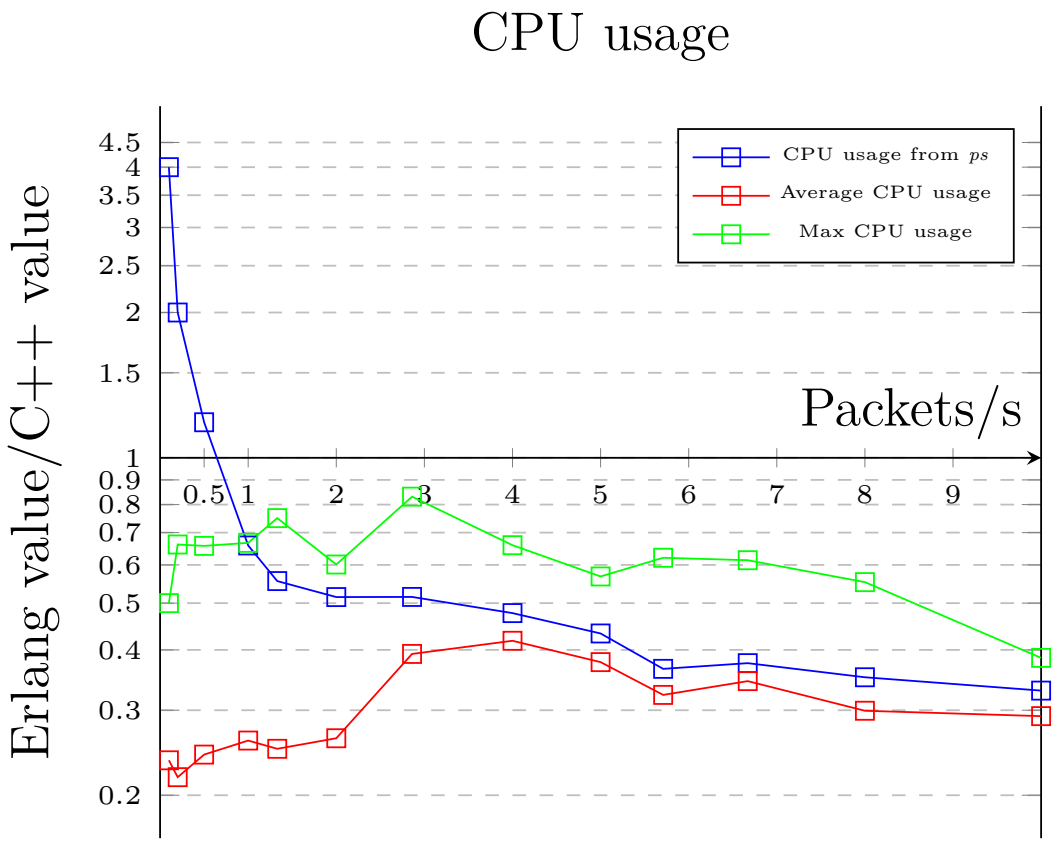

Figure 4.7: Data collected about CPU usage as described in Section 3.3.

4.5 Power consumption

By using a multimeter to measure amperes and volts for each data generation rate of a prototype every five seconds an average power consumption value was attained. Figure 4.8 presents these values and Figure 4.9 shows how many times greater the power consumption is for the Erlang prototype.

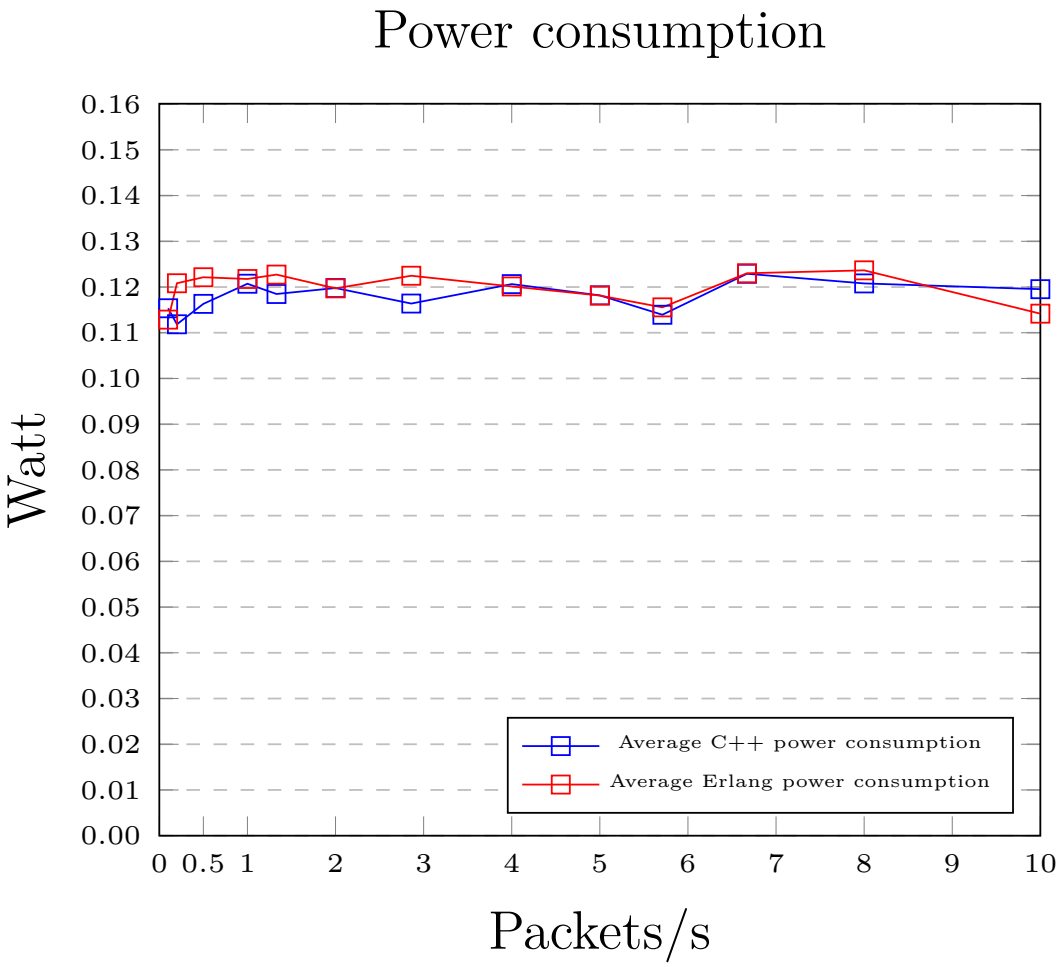

Figure 4.8: Data collected about the power consumption of the prototypes.

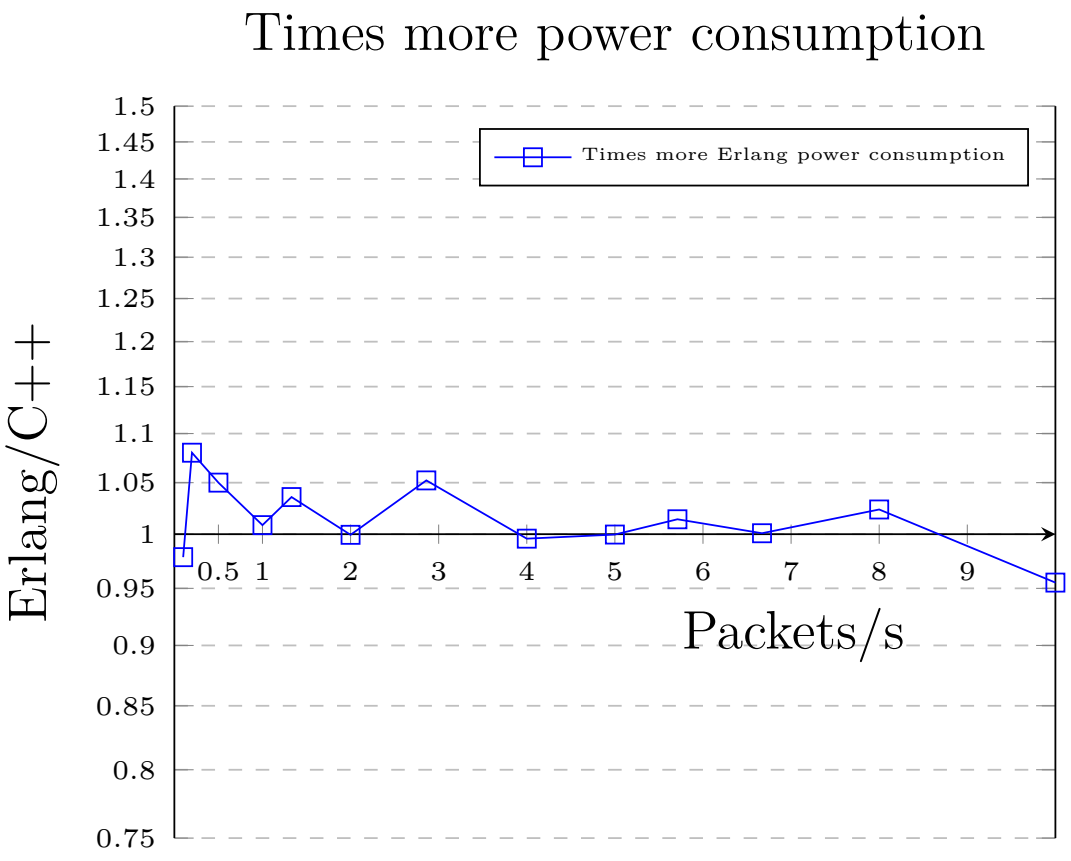

Figure 4.9: Diagram showing how much more power Erlang consumes.

Figure 4.9 shows that there is no clear difference between the power consumption of the prototypes. Therefore, the next step was to investigate the confidence of that the data is a representation of real data.

The data collection process revealed that the board seemed to have two states of power consumption, one high and one low. Therefore, it is possible to ask the question: what is the probability that the board is in a high-power state at any given time?

The Wilson procedure answers this question by calculating a confidence interval (Wilson, 1927). If this procedure is used on the values collected from the Erlang prototype with the data generation rate of ten packets per second it results in a 95% confidence interval of $52.02 \pm 19.65\%$ that the board is in a high-power state during the execution of the prototype. The meaning of the 95% confidence interval is that the true mean of the power consumption is 95% likely to be inside the interval. In this case, the number of observations is too few to give statistically significant results for any separate data generation rate.

An alternative is to collect all the measurements from each prototype into one data set. This models the situation where the data generation rate varies at random. This gives the confidence interval $65.92 \pm 5.72\%$ for the Erlang prototype and the

confidence interval $61.75 \pm 5.87\%$ for the C++ prototype that the board is in a high-power state. Because the intervals overlap, the test still does not give any significant information about the power consumption data.

4.6 Analysis of the results

The results show that the Erlang prototype uses less memory and CPU than the C++ prototype. This is perhaps surprising since C++ is a language known for its efficiency while Erlang is not. Therefore, the prototypes were investigated to see where the performance difference came from.

Two main differences were found: The Erlang prototype uses a port to communicate with a ZigBee module while the C++ prototype communicates directly with the ZigBee module, and the C++ prototype uses D-Bus to communicate data between processes while the Erlang prototype uses message passing inside the virtual machine. The first difference could immediately be discarded as an explanation of the performance results, because adding something extra compared to the C++ prototype could not possibly result in less memory and CPU usage. Therefore, the one remaining reason that the C++ prototype uses more memory and CPU is that the C++ prototype uses D-Bus to communicate between processes.

To show the impact that D-Bus has on performance the Reader process was selected to perform some more experiments on. The same data collection program that had been used to measure the complete prototypes was modified to run only the Reader process, and data was collected about the impact D-Bus had on the memory and CPU usage.

4.6.1 Memory use of the Reader process

These experiments were performed on a version of the Reader process that had all D-Bus functionality included and on a version that had all D-Bus functionality removed. The results for memory usage from these experiments is shown in Figures 4.10 and 4.11.

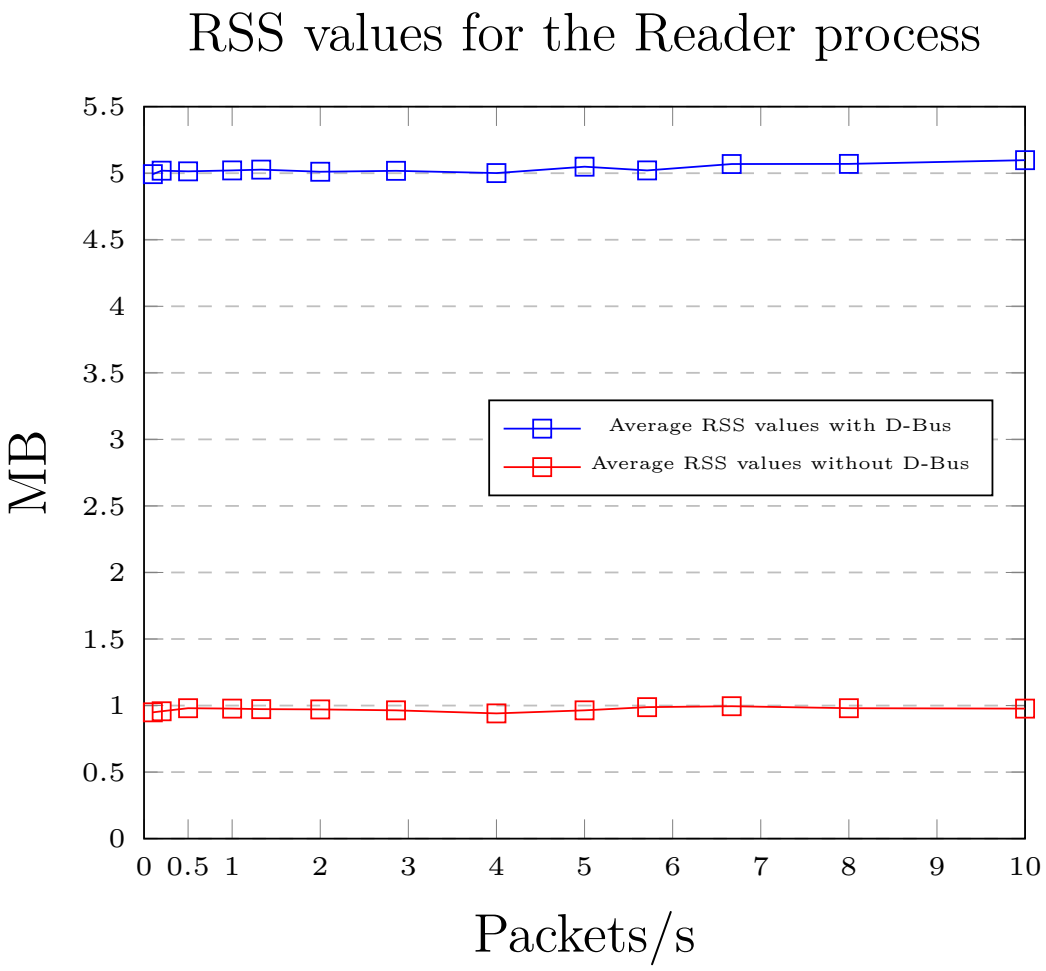

Figure 4.10: Data about physical memory usage for the Reader process.

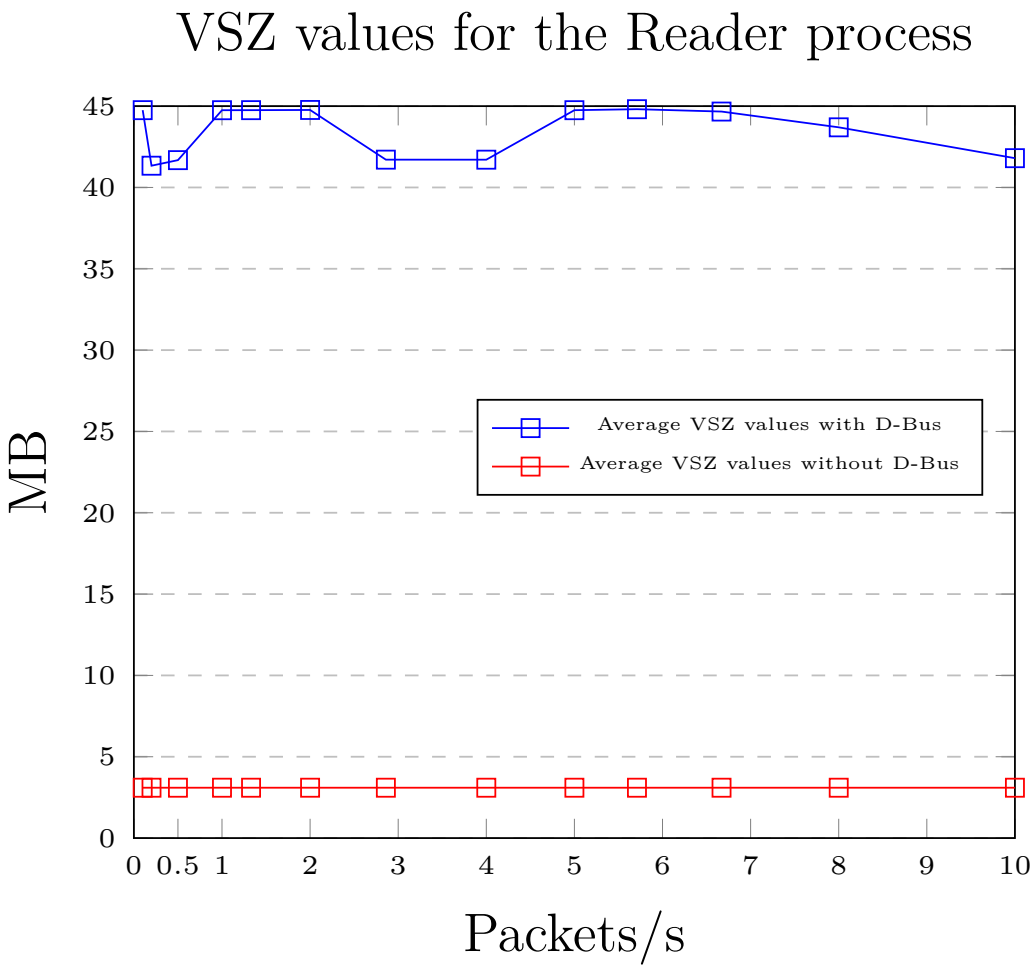

Figure 4.11: Data about virtual memory usage for the Reader process with D-Bus included and without D-Bus included.

The diagrams in Figures 4.10 and 4.11 show the amount of each type of memory the Reader process uses. Figure 4.12 uses this information to show how many times more memory the process uses with D-Bus included.

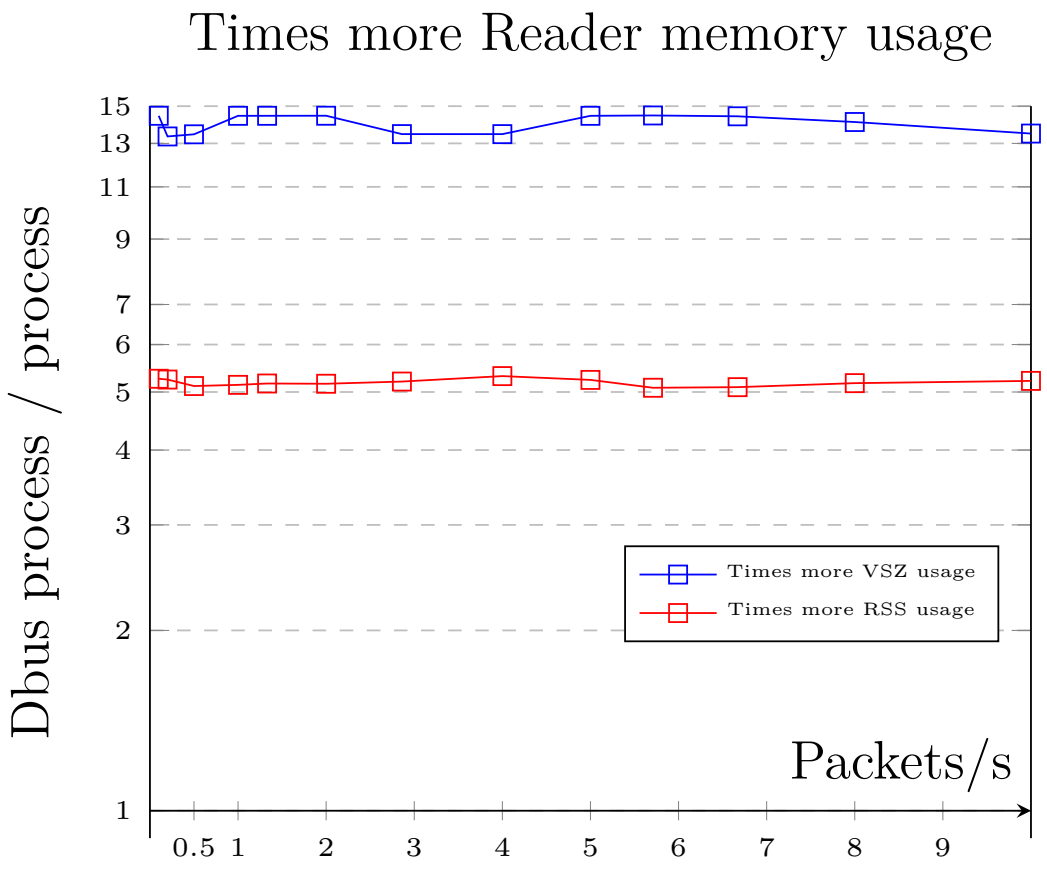

Figure 4.12: Diagram showing how many times more memory the Reader process uses when D-Bus is included.

4.6.2 CPU usage of the Reader process

Figures 4.13 and 4.14 show that the Reader process uses significantly more CPU when D-Bus is included. The two figures correspond to the two methods of measuring CPU usage described in Section 3.3.

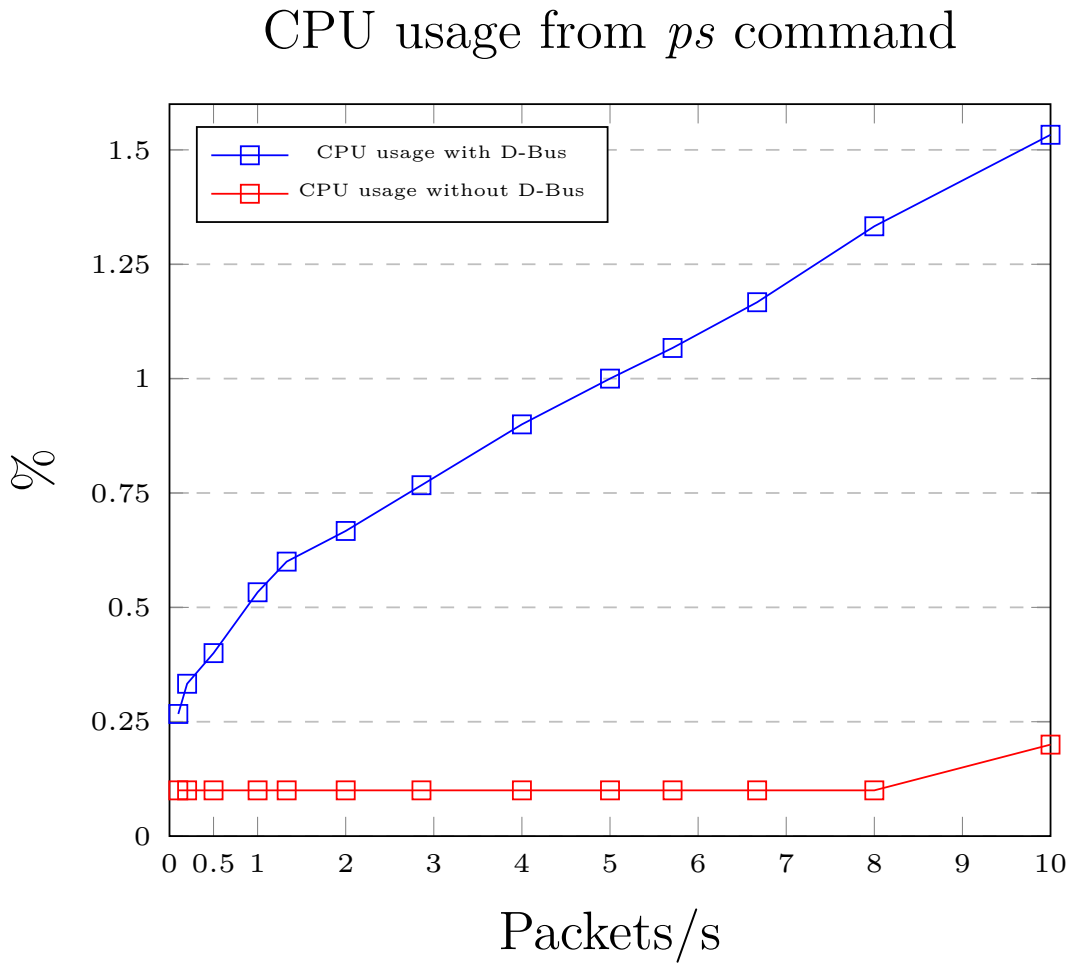

Figure 4.13: Data about the CPU usage of the Reader process collected from the *ps* command.

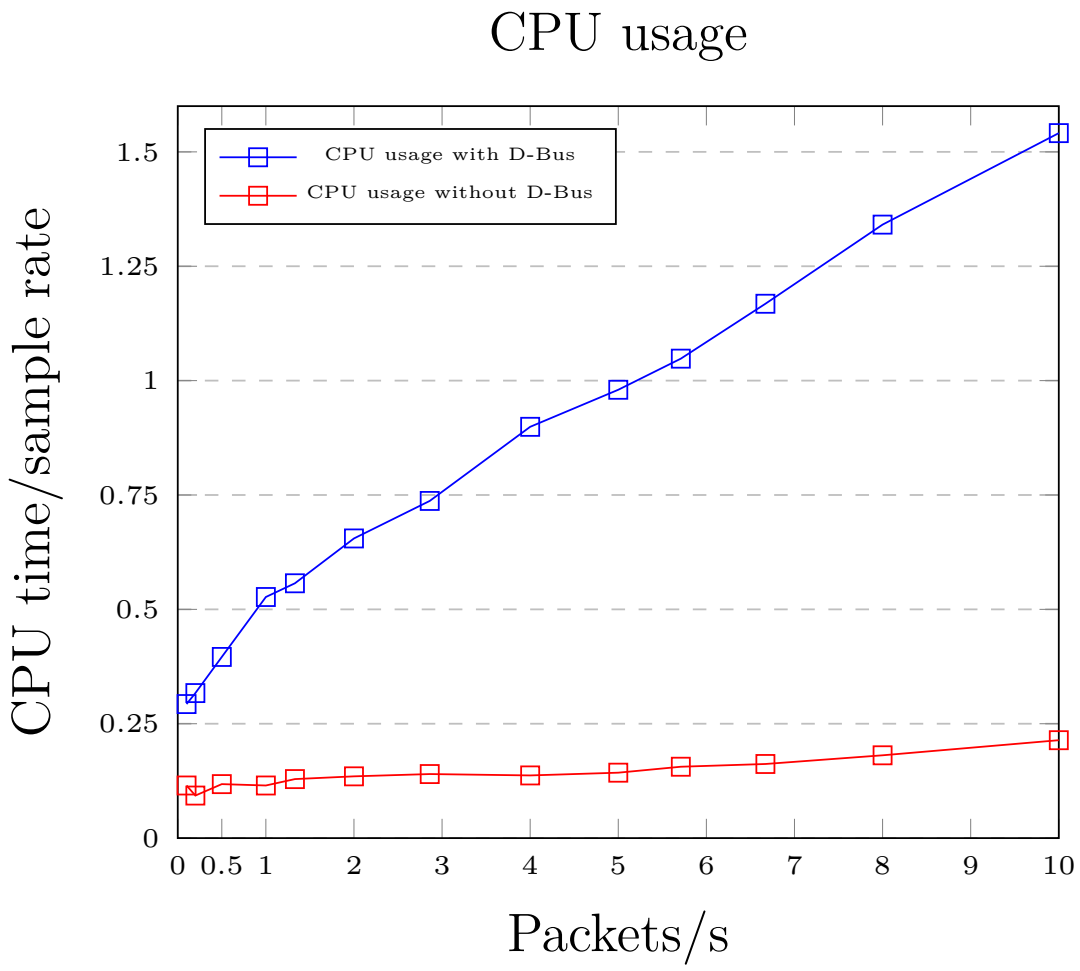

Figure 4.14: Data about the CPU usage of the Reader process collected as described in Section 3.3.

Figures 4.13 and 4.14 show that the use of D-Bus to communicate messages between processes significantly increases the CPU usage. Figure 4.15 shows how much more the D-Bus version of the Reader process uses the CPU.

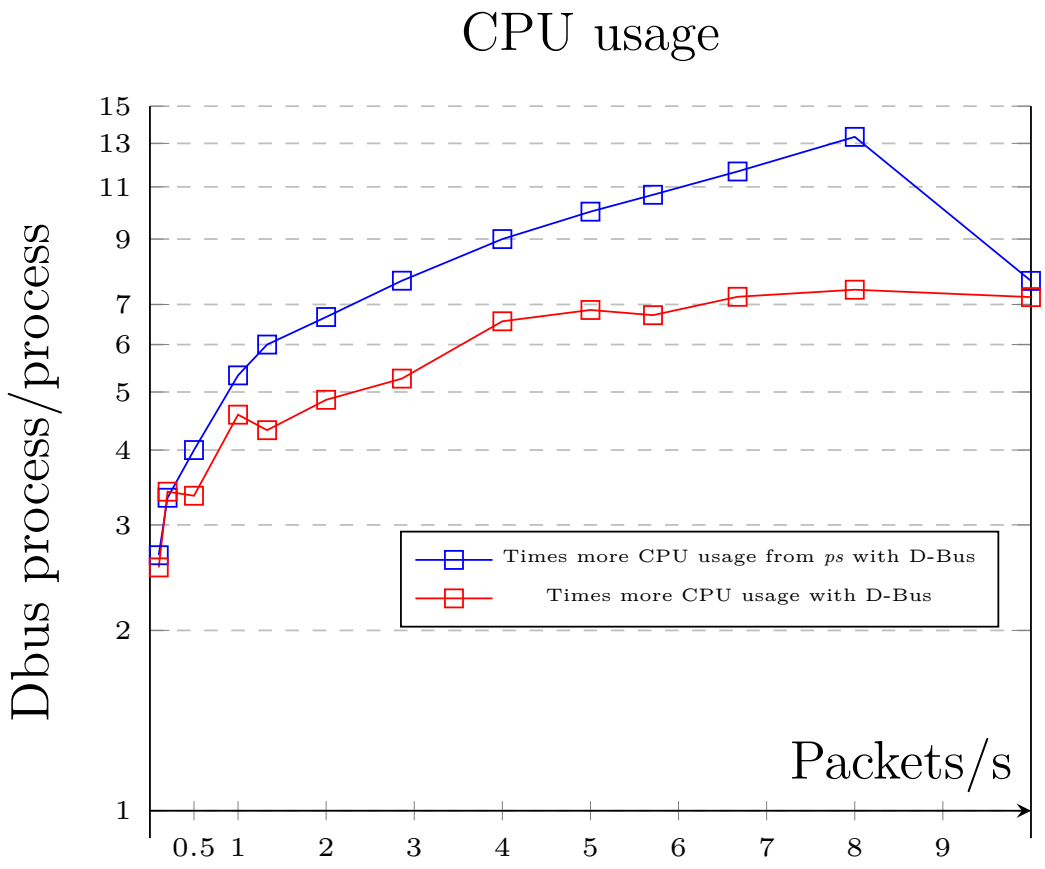

Figure 4.15: Diagram showing how much more CPU the Reader process uses when D-Bus is included.

4.6.3 Comparison with Erlang prototype

The figures above quantify the effect of using D-Bus on the Reader process's performance. Since the Reader, Writer and Datagen processes are quite similar the assumption was made that D-Bus has a similar performance effect on all of them, and it was used to estimate the performance of a hypothetical prototype that does not use D-Bus.

This estimate is done by multiplying the C++ prototype's performance values with the performance factor gained by removing D-Bus from the Reader process. Figure 4.16 shows how many times more memory and CPU the Erlang prototype uses compared to the hypothetical C++ prototype.

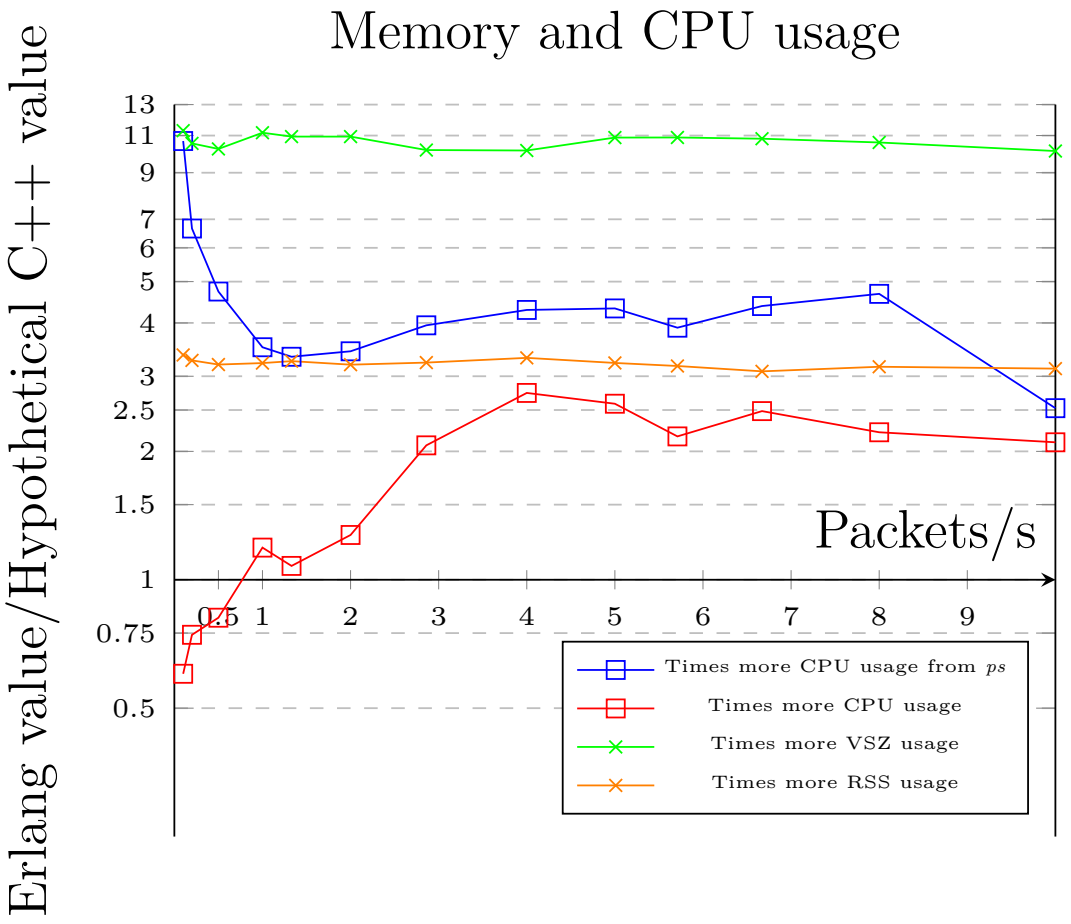

Figure 4.16: Diagram showing how many times more memory and CPU the Erlang prototype require compared to a hypothetical C++ prototype.

4.7 Summary of the results

Table 4.1 summarises the results of this chapter. The values for the memory usage are an average value over all data generation rates and the CPU values are computed from the gradient of the trend line equation.

The values in the hypothetical C++ prototype are derived by assuming that D-Bus would have the same performance impact on every process. This does not give a working prototype because there is now no method to communicate between processes but it gives an estimate of what the values could be if all process were bundled together using shared memory and threads instead. However, using shared memory is not a good solution because the complexity of the program becomes much higher and the robustness of the program is worse. The chance of mistakes increases and it can lead to for example data races when accessing the shared memory.

Table 4.1: Summary of the results from the investigation.

	Erlang prototype	C++ prototype	Hypothetical C++ prototype
Physical memory usage in MB	8.95	14.44	2.79
Virtual memory usage in MB	80	105.29	7.5
CPU ms per packet from *ps* command	2.2	7.5	0.7
CPU ms per packet	2.3	7.4	0.9
Lines of code in total	317	352	-
Lines of code, only the processes	45	167	-
Static code size in kB with the Erlang runtime system included	2138	50	-
Static code size in kB without the Erlang runtime system included	17	50	-

5

Conclusion

This thesis compares C++ and Erlang for the task of communication in a WSN, by implementing and evaluating one prototype in each language. The design of the prototypes corresponds with the product that the affiliated company Cipherstone Technologies is considering to develop. The evaluation has been done on one node in the network and the purpose of the evaluation was to investigate the performance impact of using Erlang instead of C++ to handle the communication with ZigBee in the network.

5.1 Discussion of results

The evaluation showed that the Erlang prototype uses less memory and less CPU if more than one data packet is sent per second. This result was at first glance a bit surprising: The expected result was that the Erlang prototype should use more memory and CPU than the C++ prototype.

Further experiments revealed that the reason for this was the use of D-Bus. A C++ version of the Reader process in which D-Bus has been removed used much less memory and much less CPU than the Erlang prototype, in all cases. The D-Bus functionality was made available in the prototype by including the glib library. Therefore, if some alternative method that require less resources to achieve inter-process communication can be found it is a possibility that a C++ prototype can be developed that use less resources compared to the Erlang prototype.

When the implementation effort was measured by counting the lines of code, the difference was not so great if the whole prototypes was compared. If instead only the corresponding parts of the C++ prototype are compared with the Erlang prototype with all C code removed the difference was significantly greater. In that case, the C++ prototype uses around three times more lines of code compared to the Erlang prototype.

The measurements of power consumption were inconclusive because the method used was not appropriate to collect data with any significance. Much more data was needed before it would show anything of significance. An automatic collection

procedure is recommended for this, because it was already tiring to collect the data and if more data was collected in the same manner the risk for human errors would increase.

Lastly, the data presented in this thesis comes from running the prototypes for each data generation rate 10 times. Each time the prototype is run for two minutes and a sample of CPU and memory usage is taken each second. This is quite a small data set for each data generation rate and prototype. Therefore, if some unforeseen noise impacted any of the executed data collection processes it is possible that it would have had a significant impact on the results. The optimal approach would be to run many more iterations of data collection to counter possible noise in the data and possibly give a more accurate data about power consumption but because of time limitations this has not been possible.

5.2 Final statements

The work done in this thesis has shown that Erlang can be an alternative to C/C++ in developing WSN applications. Comparing results from the two prototypes show that the Erlang prototype requires less code, uses less memory and CPU in most cases. Data about CPU usage collected with the *ps* command shows that the C++ prototype use less CPU only if no more than 0.5 packets are sent per second. The CPU usage also increases more rapidly as the data generation rate increases for the C++ prototype compared to the Erlang prototype. Therefore, more is gained by using Erlang if the system needs to send data at a higher rate.

The investigation into the requirements of using D-Bus showed that D-Bus accounted for a large part of the resource usage. Therefore, it would be easy to conclude that if D-Bus is not used the C++ prototype would use less resources but this is something that can not be concluded from in this thesis because, by removing the D-Bus functionality and not replacing it with something else the hypothetical C++ prototype is not a working prototype. It only gives approximate values for an alternative solution where the resource cost of moving data between processes is zero. Furthermore, as the paper by Sivieri (2012) it is also possible to reduce the memory usage of the Erlang runtime system by removing unnecessary functionality and features.

In application similar to the one in this thesis, where the performance is the most important aspect then Erlang should be used as long as the data generation rate is higher than two packets per second. The Erlang prototype also uses less memory and it would therefore be recommended as the solution in a system where it is required that the memory usage is as small as possible.

The final consideration is how important the security and implementation effort are. The Erlang prototype uses less lines of code and the chance for security risks in the code is therefore less. If an application generates data at a rate less than one packets

per second a position must be taken about what is more important, the security of the application or the performance.

5.3 Future work

The high memory and CPU usage of the C++ prototype in this thesis was caused by the use of the D-Bus inter-process communication functionality from the glib library. Therefore, one obvious question is if there are any other alternatives that can achieve better performance and what the reason for the higher resource requirements are.

In the Erlang prototype a port was implemented to connect the Erlang part of the prototype to C code that managed the communication with other nodes in the network over ZigBee. This connection could for example be done with a port driver instead or something else. It would therefore be interesting to to implement these alternatives and investigate if any of them would reduce the memory usage, CPU usage and the power consumption of the prototype.

Another direction that could be interesting to investigate is if there are other techniques that can achieve the same communication requirements with the same or lower power consumption. If there exists such a technique, the next step would be to investigate how to communicate between it and Erlang and to measure if it would require more or less memory and CPU usage.

Bibliography

Armstrong, J. (1996). Erlang—a Survey of the Language and its Industrial Applications. In *Proc. INAP*, Volume 96.

Armstrong, J. (1997). The development of Erlang. In *ACM SIGPLAN Notices*, Volume 32, No. 8, pp. 196–203. ACM.

Armstrong, J. (2003). Concurrency oriented programming in Erlang. `https://guug.de/veranstaltungen/ffg2003/papers/ffg2003-armstrong.pdf` [Accessed: 2019-04-25].

Armstrong, J. (2010). Erlang. *Communications of the ACM 53*(9), 68–75.

Bello, O., S. Zeadally, and M. Badra (2017). Network layer inter-operation of Device-to-Device communication technologies in Internet of Things (IoT). *Ad Hoc Networks 57*, 52 – 62. Special Issue on Internet of Things and Smart Cities security, privacy and new technologies.

Carmack, J. (2013). John Carmack Keynote - Quakecon 2013. Youtube. `https://www.youtube.com/watch?v=Uooh0Y9fC_M` [Accessed: 2018-11-30].

DIGI (2019). *DIGI XBEE® AND DIGI XBEE-PRO® ZIGBEE*. Digi International Inc. `https://www.mouser.se/datasheet/2/111/ds_xbee_zigbee-1019686.pdf` [Accessed: 2019-02-09].

Ericsson AB (2019a). Erlang Run-Time System Application (ERTS) - 6 How to Implement an Alternative Carrier for the Erlang Distribution. Erlang.org. `http://erlang.org/doc/apps/erts/alt_dist.html` [Accessed: 2019-05-01].

Ericsson AB (2019b). Erlang/OTP 21.2. Erlang.org. `http://erlang.org/doc/index.html` [Accessed: 2019-01-18].

Erlang (2019). Erlang programming language. `http://www.erlang.org/` [Accessed: 2019-05-13].

Essameldin, A. and K. Harras (2016). Device-to-Device Communication in the Internet of Things QSIURP Report. `http://www.contrib.andrew.cmu.edu/~aeahmed/device-device-communication%20(1).pdf`.

Farahani, S. (2011). *ZigBee wireless networks and transceivers*. Newnes. ISBN: 9780080558479.

Fedrecheski, G., L. C. Costa, and M. K. Zuffo (2016). Elixir programming language evaluation for IoT. In *Consumer Electronics (ISCE), 2016 IEEE International Symposium on*, pp. 105–106. IEEE.

Gislason, D. (2008). *ZigBee wireless networking*. Newnes. ISBN: 9780080558622.

Gutierrez, J. A., E. H. Callaway, and R. L. Barrett (2004). *Low-rate wireless personal area networks: enabling wireless sensors with IEEE 802.15.4*. IEEE Standards Association.

Haenisch, T. (2016). A case study on using functional programming for internet of things applications. *Athens Journal of Technology & Engineering 3*(1), 29–38.

IEEE Computer Society (2016). *IEEE Standard for Low-Rate Wireless Networks*. IEEE Std 802.15.4-2015 (Revision of IEEE Std 802.15.4-2011).

Institute of Electrical and Electronics Engineers, Inc. (2003). *IEEE Standard for Information Technology — Telecommunications and Information Exchange between Systems — Local and Metropolitan Area Networks — Specific Requirements — Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low Rate Wireless Personal Area Networks (LR-WPANs)*. IEEE Std 802.15.4-2003. New York: IEEE Press. `https://www.iith.ac.in/~tbr/teaching/docs/802.15.4-2003.pdf` [Accessed: 2019-01-17].

Kohvakka, M., M. Kuorilehto, M. Hännikäinen, and T. D. Hämäläinen (2006). Performance analysis of IEEE 802.15. 4 and ZigBee for large-scale wireless sensor network applications. In *Proceedings of the 3rd ACM international workshop on Performance evaluation of wireless ad hoc, sensor and ubiquitous networks*, pp. 48–57. ACM.

Leibson, S. (2008). IPV6: How Many IP Addresses Can Dance on the Head of a Pin? EDN Network. `https://www.edn.com/electronics-blogs/other/4306822/IPV6-How-Many-IP-Addresses-Can-Dance-on-the-Head-of-a-Pin-` [Accessed: 2018-11-30].

Mainland, G., G. Morrisett, and M. Welsh (2008). Flask: Staged functional programming for sensor networks. In *ACM Sigplan Notices*, Volume 43, No. 9, pp. 335–346. ACM.

Militano, L., G. Araniti, M. Condoluci, I. Farris, and A. Iera (2015). Device-to-Device Communications for 5G Internet of Things. *EAI Endorsed Transactions on Internet of Things 1*(1), e4.

NXP (2012). *SABRE Board for Smart Devices Based on the i.MX 6 Series*. Document number: IMX6SABRESDBFS REV 3. `https://www.nxp.com/files-static/32bit/doc/fact_sheet/RDIMX6SABREBRDFS.pdf` [Accessed: 2019-01-24].

Ray, B., D. Posnett, V. Filkov, and P. Devanbu (2014). A large scale study of programming languages and code quality in Github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 155–165. ACM.

Sivieri, A. (2012). Erlang meets WSNs: a functional approach to WSN programming. In *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2012 IEEE International Conference on*, pp. 562–563. IEEE.

Sivieri, A. and G. Cugola (2012). WSN-Erlang: a Functional, High Level Approach to WSN Development. In *9th European Conference on Wireless Sensor Networks (EWSN 2012)*, pp. 27–28.

Sivieri, A., L. Mottola, and G. Cugola (2016). Building Internet of Things software with ELIoT. *Computer Communications 89*, 141–153.

Thomas, D. (2018). *Programming Elixir≥ 1.6: Functional/> Concurrent/> Pragmatic/> Fun*. Pragmatic Bookshelf. ISBN: 978-1-68050-299-2.

Wheeler, D. A. (2019). SLOCCount. `https://dwheeler.com/sloccount/` [Accessed: 2019-05-01].

Wilson, E. B. (1927). Probable inference, the law of succession, and statistical inference. *Journal of the American Statistical Association 22*(158), 209–212.

Yang, S.-H. (2014). Internet of Things. In *Wireless Sensor Networks: Principles, Design and Applications*. Springer London. ISBN: 978-1-4471-5505-8.

ZigBee Alliance (2012). *Document 053474r20; ZigBee Specification*. ZigBee Standards Organization.

ZigBee Alliance (2019). ZigBee is the only complete loT solution, from the mesh network to the universal language that allows smart objects to work together. ZigBee Alliance. `https://www.zigbee.org/zigbee-for-developers/zigbee-3-0/` [Accessed: 2019-01-13].

Appendix A

C++ prototype

A.1 Writer

```
1  #include <iostream>
2  #include "../cpp_serial/SerialPort.h"
3  #include <unistd.h>
4  #include <glib.h>
5  #include <gio/gio.h>
6  #include <string.h>
7
8  using namespace std;
9  SerialPort port;
10
11 /* this function will get called everytime a client attempts to
       connect */
12
13 gboolean incoming_callback  (GSocketService *service,
14                              GSocketConnection *connection,
15                              GObject *source_object,
16                              gpointer user_data)
17 {
18     //g_print("Received Connection from client!\n");
19     GInputStream * istream = g_io_stream_get_input_stream (
           G_IO_STREAM (connection));
20     gchar message[1024];
21     g_input_stream_read  (istream,
22                           message,
23                           1024,
24                           NULL,
25                           NULL);
26     char dataStr[11];
27     sprintf(dataStr, "%s", message);
28     port.setData(dataStr);
29     std::cout << "Writer: " << dataStr << std::endl;
30
31     int n = port.write_to_zigbee();
32     if(n == -1){
33       std::cout << "Could not write!" << std::endl;
34     }
35     return FALSE;
36 }
```

```
37
38 int main()
39 {
40     while(port.getPort() == -1){
41
42         port.open_port_serial("/dev/ttyUSB0");
43
44         if (port.getPort() == -1){
45             printf("Error opening serial port /dev/ttyUSB1 \n");
46         }
47         else
48         {
49             printf("Serial Port /dev/ttyUSB1 is Open\n");
50             if (port.initport() == -1)
51             {
52                 printf("Error Initializing port");
53                 port.uninitialize();
54                 return 0;
55             }
56         }
57     }
58
59     GError * error = NULL;
60
61     /* create the new socketservice */
62     GSocketService * service = g_socket_service_new ();
63
64     /* connect to the port */
65     g_socket_listener_add_inet_port ((GSocketListener*)service,
66                                      1500, /* your port goes here */
67                                      NULL,
68                                      &error);
69
70     /* don't forget to check for errors */
71     if (error != NULL)
72     {
73         g_error (error->message);
74     }
75
76     /* listen to the 'incoming' signal */
77     g_signal_connect (service,
78                       "incoming",
79                       G_CALLBACK (incoming_callback),
80                       NULL);
81
82     /* start the socket service */
83     g_socket_service_start (service);
84
85     /* enter mainloop */
86     g_print ("Waiting for client!\n");
87     GMainLoop *loop = g_main_loop_new(NULL, FALSE);
88     g_main_loop_run(loop);
89
90     return 0;
91 }
```

A.2 Reader

```
1  #include <iostream>
2  #include "../cpp_serial/SerialPort.h"
3
4  #include <string.h>
5  #include <unistd.h>
6  #include <cstddef>
7  #include <cstdio>
8  #include <glib.h>
9  #include <gio/gio.h>
10
11 int main()
12 {
13     SerialPort port;
14
15    while(port.getPort() == -1){
16
17        port.open_port_serial("/dev/ttyUSB0");
18
19        if (port.getPort() == -1){
20            printf("Error opening serial port /dev/ttyUSB2 \n");
21        }
22        else
23        {
24            printf("Serial Port /dev/ttyUSB2 is Open\n");
25            if (port.initport() == -1)
26            {
27                printf("Error Initializing port");
28                port.uninitialize();
29                return 0;
30            }
31        }
32    }
33
34     while(1){
35
36         if(port.read_from_zigbee() > 0)
37         {
38             char dataStr[11];
39             port.getData(dataStr);
40             const char *str = (const char*)dataStr;
41             std::cout  << str << std::endl;
42
43             GError * error = NULL; /* initialize glib */
44
45             /* create a new connection */
46             GSocketConnection * connection = NULL;
47             GSocketClient * client = g_socket_client_new();
48
49             /* connect to the host */
50             connection = g_socket_client_connect_to_host
51                 (client,
52                 (gchar*)"localhost",
53                  1500, /* your port goes here */
```

```
54                      NULL,
55                      &error);
56                  /* use the connection */
57                  //GInputStream * istream =
                        g_io_stream_get_input_stream (G_IO_STREAM (
                        connection));
58                  GOutputStream * ostream =
                        g_io_stream_get_output_stream (G_IO_STREAM (
                        connection));
59                  g_output_stream_write  (ostream,
60                                          str, /* your message goes here
                                                */
61                                          strlen(str), /* length of your
                                                message */
62                                          NULL,
63                                          &error);
64
65                  g_io_stream_close((GIOStream*)connection, NULL, NULL);
66                  /* don't forget to check for errors */
67                  if (error != NULL)
68                  {
69                      g_error (error->message);
70                  }
71              }
72          }
73          return 0;
74      }
```

A.3 Datagen

```
1  #include <iostream>
2  #include <glib.h>
3  #include <gio/gio.h>
4  #include <unistd.h>
5  #include <time.h>
6  #include <string.h>
7
8  using namespace std;
9
10 int main(int argc, char *argv[])
11 {
12     std::cout << "DataGen started" << std::endl;
13     struct timespec ts;
14     ts.tv_sec  = atoi(argv[1]);
15     ts.tv_nsec = atoi(argv[2])*1000*1000; //250000000L;
16
17     int counter = 0;
18     while(1){
19
20         std::string text = "1 Msg:";
21         text += std::to_string((counter % 899)+100) + "\n";
22         counter++;
```

```
23          const char *str = text.c_str();
24          std::cout << str << std::endl;
25
26          GError * error = NULL; /* initialize glib */
27
28          /* create a new connection */
29          GSocketConnection * connection = NULL;
30          GSocketClient * client = g_socket_client_new();
31
32          /* connect to the host */
33          connection = g_socket_client_connect_to_host
34               (client,
35               (gchar*)"localhost",
36                1500, /* your port goes here */
37                NULL,
38                &error);
39          /* use the connection */
40          //GInputStream * istream = g_io_stream_get_input_stream (
                G_IO_STREAM (connection));
41          GOutputStream * ostream = g_io_stream_get_output_stream (
                G_IO_STREAM (connection));
42          g_output_stream_write  (ostream,
43                                  str, /* your message goes here */
44                                  strlen(str), /* length of your
                                        message */
45                                  NULL,
46                                  &error);
47          /* don't forget to check for errors */
48          if (error != NULL)
49          {
50              g_error (error->message);
51          }
52          g_io_stream_close((GIOStream*)connection, NULL, NULL);
53          nanosleep(&ts, NULL);
54      }
55      return 0;
56  }
```

A.4 Serial port

```
 1  #ifndef SERIALPORT_H_INCLUDED
 2  #define SERIALPORT_H_INCLUDED
 3
 4  #include <termios.h>
 5  #include <inttypes.h>
 6
 7  #include <string>
 8
 9  #include <sstream>
10
```

```
11 #include <fcntl.h>
12 #include <errno.h>
13 #include <unistd.h>
14
15 #include <cstddef>
16 #include <cstdio>
17 #include <stdio.h>
18
19 class SerialPort
20 {
21     public:
22
23         void open_port_serial(std::string  str);
24         int initport();
25         int setBlocking(int should_block);
26         int getPort();
27         void setPort(int i);
28         void getData(char* outStr);
29         void setData(char* inStr);
30         int read_from_zigbee();
31         int write_to_zigbee();
32
33         void uninitialize();
34
35         ~SerialPort();
36
37     private:
38         void append(int i, int n, char* indata);
39         int serial_fd = -1;
40         char data[11];
41
42 };
43
44 #endif // SERIALPORT_H_INCLUDED
```

```
1  #include "SerialPort.h"
2  #include <iostream>
3  #include <sstream>
4  #include <stdexcept>
5
6  #include <fcntl.h>
7  #include <errno.h>
8  #include <unistd.h>
9  #include <string.h>
10 #include <time.h>
11
12 using namespace std;
13
14
15 void SerialPort::open_port_serial(std::string  str)
16 {
17     int fd; /* File descriptor for the port */
18     fd = open(str.c_str(), O_RDWR | O_NOCTTY | O_NONBLOCK);
19     serial_fd = fd;
20 }
21
```

```
22 //Initialize serial port
23 int SerialPort::initport()
24 {
25     int portstatus = 0;
26
27     struct termios options;
28
29     // Get the current options for the port...
30     tcgetattr(serial_fd, &options);
31
32     // Set the baud rates to 9600...
33     cfsetispeed(&options, B9600);
34     cfsetospeed(&options, B9600);
35
36     // Enable the receiver and set local mode...
37     options.c_cflag |= (CLOCAL | CREAD);
38
39     options.c_cflag &= ~PARENB;
40     options.c_cflag &= ~CSTOPB;
41     options.c_cflag &= ~CSIZE;
42     options.c_cflag |= CS8;
43     //options.c_cflag |= SerialDataBitsInterp(8);      /* CS8 -
           Selects 8 data bits */
44     options.c_cflag &= ~CRTSCTS;                       // Disable
           hardware flow control
45     options.c_iflag &= ~(IXON | IXOFF | IXANY);        // Disable
          XON XOFF (for transmit and receive)
46     //options.c_cflag |= CRTSCTS;                      /* Enable
           hardware flow control */
47
48     options.c_cc[VMIN] = 1;    //Minimum characters to be read
49     options.c_cc[VTIME] = 2;     //Time to wait for data (tenths of
           seconds)
50     options.c_oflag &=~OPOST;
51     options.c_iflag &=~(ICANON | ECHO | ECHOE | ISIG);
52     // Set the new options for the port...
53     tcsetattr(serial_fd, TCSANOW, &options);
54
55     //Set the new options for the port...
56     tcflush(serial_fd, TCIFLUSH);
57     if (tcsetattr(serial_fd, TCSANOW, &options) == -1)
58     {
59         perror("On tcsetattr:");
60         portstatus = -1;
61     }
62     else
63         portstatus = 1;
64
65     return (portstatus);
66 }
67
68 int SerialPort::setBlocking(int should_block)
69 {
70     struct termios tty;
71         memset (&tty, 0, sizeof tty);
72         if (tcgetattr (serial_fd, &tty) != 0)
```

```
73          {
74              printf("error %d from tggetattr\n", errno);
75          }
76
77          tty.c_cc[VMIN]  = should_block ? 1 : 0;
78          tty.c_cc[VTIME] = 10;            // 1 seconds read timeout
79
80          if (tcsetattr (serial_fd, TCSANOW, &tty) != 0)
81                  printf("error %d setting term attributes\n", errno)
                        ;
82  }
83
84  int SerialPort::getPort()
85  {
86      return serial_fd;
87  }
88
89  void SerialPort::getData(char* outStr)
90  {
91    for(int i=0; i < 11; ++i){
92      outStr[i] = data[i];
93    }
94  }
95
96  void SerialPort::setData(char* inStr)
97  {
98      strncpy(data,inStr,10);
99  }
100
101 SerialPort::~SerialPort()
102 {
103     uninitialize();
104 }
105
106 void SerialPort::uninitialize()
107 {
108     if (serial_fd >= 0)
109     {
110         close(serial_fd);
111         serial_fd = -1;
112     }
113 }
114
115 int SerialPort::read_from_zigbee()
116 {
117     struct timespec ts;
118     ts.tv_sec  = 0;
119     ts.tv_nsec = 95*1000*1000;
120
121     int n1 = 0;
122     char indata[11];
123     while(n1 < 1 || (n1 != 10)){
124         setBlocking(1);
125         int n = read(serial_fd, indata, 10-n1);
126         //std::cout << n << std::endl;
127         setBlocking(0);
```

```
128             if(n == 10){
129                 strncpy(data, indata, 10);
130                 return n;
131             } else if(n != -1){
132                 append(n1, n, indata);
133                 n1 = n1+n;
134             }
135             if(n1 < 1 || (n1 != 10)) {
136                 nanosleep(&ts, NULL);
137             }
138         }
139         return n1;
140     }
141
142     void SerialPort::append(int i, int n, char* indata)
143     {
144             for(int j=0; j<n; j++){
145                 data[i+j] = indata[j];
146             }
147     }
148
149     int SerialPort::write_to_zigbee()
150     {
151         return write(serial_fd, &data, 10);
152     }
```

A.5 Systemd

```
1  [Unit]
2  Description=writer service
3  StartLimitIntervalSec=0
4
5  [Service]
6  Type=simple
7  Restart=always
8  RestartSec=1
9  User=user
10 ExecStart=/usr/bin/env bash /writer
11
12 [Install]
13 WantedBy=multi-user.target
```

Appendix B

Erlang prototype

B.1 Port

```
1  #define _GNU_SOURCE
2  #include <stdio.h>
3  #include <string.h>
4  #include <unistd.h> /* UNIX standard function definitions */
5  #include <fcntl.h> /* File control definitions */
6  #include <errno.h>
7  #include <termios.h> /* POSIX terminal control definitions */
8  #include <signal.h>
9  #include <stdlib.h>
10 #include <sys/mman.h>
11 #include <sys/types.h>
12 #include <sys/wait.h>
13 #include <sys/file.h>
14 #include <sys/stat.h>        /* For mode constants */
15 #include <sys/ipc.h>
16 #include <sys/shm.h>
17 #include <time.h>
18
19 //Initialize serial port
20 int initport(int fd)
21 {
22     int portstatus = 0;
23
24     struct termios options;
25
26     // Get the current options for the port...
27     tcgetattr(fd, &options);
28
29     // Set the baud rates to 9600...
30     cfsetispeed(&options, B9600);
31     cfsetospeed(&options, B9600);
32
33     // Enable the receiver and set local mode...
34     options.c_cflag |= (CLOCAL | CREAD);
35
36     options.c_cflag &= ~PARENB;
37     options.c_cflag &= ~CSTOPB;
38     options.c_cflag &= ~CSIZE;
```

```
39      options.c_cflag |= CS8;
40      //options.c_cflag |= SerialDataBitsInterp(8);       /* CS8 -
            Selects 8 data bits */
41      options.c_cflag &= ~CRTSCTS;                        // Disable
            hardware flow control
42      options.c_iflag &= ~(IXON | IXOFF | IXANY);         // Disable
            XON XOFF (for transmit and receive)
43      //options.c_cflag |= CRTSCTS;                       /* Enable
            hardware flow control */
44
45      options.c_cc[VMIN] = 1;    //Minimum characters to be read
46      options.c_cc[VTIME] = 2;     //Time to wait for data (tenths of
            seconds)
47      options.c_oflag &=~OPOST;
48      options.c_iflag &=~(ICANON | ECHO | ECHOE | ISIG);
49      // Set the new options for the port...
50      tcsetattr(fd, TCSANOW, &options);
51
52      //Set the new options for the port...
53      tcflush(fd, TCIFLUSH);
54      if (tcsetattr(fd, TCSANOW, &options) == -1)
55      {
56          perror("On tcsetattr:");
57          portstatus = -1;
58      }
59      else
60          portstatus = 1;
61
62      return (portstatus);
63  }
64
65  int open_port_serial(char *str)
66  {
67      int fd; /* File descriptor for the port */
68      fd = open(str, O_RDWR | O_NOCTTY | O_NONBLOCK);
69
70      return fd;
71  }
72
73  void append(int i, int n, char* content, char* indata)
74  {
75          for(int j=0; j<n; j++){
76              content[i+j] = indata[j];
77          }
78  }
79
80  void set_blocking (int fd, int should_block)
81  {
82          struct termios tty;
83          memset (&tty, 0, sizeof tty);
84          if (tcgetattr (fd, &tty) != 0)
85          {
86                  printf("error %d from tggetattr\n", errno);
87                  return;
88          }
89
```

```
90          tty.c_cc[VMIN]  = should_block ? 1 : 0;
91          tty.c_cc[VTIME] = 10;            // 1 seconds read timeout
92
93          if (tcsetattr (fd, TCSANOW, &tty) != 0)
94                  printf("error %d setting term attributes\n", errno)
                        ;
95  }
96
97  int read_from_zigbee(int serial_fd, char *content)
98  {
99      struct timespec ts;
100     ts.tv_sec  = 0;
101     ts.tv_nsec = 95*1000*1000;
102
103     int n1 = 0;
104     char indata[11];
105     while(n1 < 1 || (n1 != 10)){
106         //set_blocking(serial_fd, 1);
107         int n = read(serial_fd, indata, 10-n1);
108         //set_blocking(serial_fd, 0);
109         if(n == 10){
110             strncpy(content, indata, 10);
111             return n;
112         } else if(n != -1){
113             append(n1, n, content, indata);
114             n1 = n1+n;
115         }
116         if(n1 < 1 || (n1 != 10)) {
117             nanosleep(&ts, NULL);
118         }
119     }
120     return n1;
121 }
122
123 int write_to_zigbee(int serial_fd, char *content)
124 {
125     int n = write(serial_fd, content, 10);
126     if (n < 0)
127     {
128       printf("write() of %ld bytes failed!\n",sizeof(*content));
129       return 0;
130     }
131     return 1;
132 }
```

```
1  /* erl_comm.c */
2
3  typedef unsigned char byte;
4
5  int read_cmd(byte *buf)
6  {
7    int len;
8
9    if (read_exact(buf, 2) != 2)
10     return(-1);
11   len = (buf[0] << 8) | buf[1];
```

```
12    return read_exact(buf, len);
13  }
14
15  int write_cmd(byte *buf, int len)
16  {
17    byte li;
18
19    li = (len >> 8) & 0xff;
20    write_exact(&li, 1);
21
22    li = len & 0xff;
23    write_exact(&li, 1);
24
25    return write_exact(buf, len);
26  }
27
28  int read_exact(byte *buf, int len)
29  {
30    int i, got=0;
31
32    do {
33      if ((i = read(0, buf+got, len-got)) <= 0)
34        return(i);
35      got += i;
36    } while (got<len);
37
38    return(len);
39  }
40
41  int write_exact(byte *buf, int len)
42  {
43    int i, wrote = 0;
44
45    do {
46      if ((i = write(1, buf+wrote, len-wrote)) <= 0)
47        return (i);
48      wrote += i;
49    } while (wrote<len);
50
51    return (len);
52  }
```

```
1  /* port.c */
2  #include <string.h>
3  #include <stdio.h>
4
5
6  typedef unsigned char byte;
7
8  int main() {
9    int fn, arg, res;
10   byte buf[100];
11
12   while (read_cmd(buf) > 0) {
13     fn = buf[0];
14     arg = buf[1];
```

```
15
16      if (fn == 1) {
17        res = initport(arg);
18        buf[0] = res;
19        write_cmd(buf, 1);
20      }else if (fn == 2) {
21        char *ps = buf;
22        ps++;
23        res = open_port_serial(ps);
24        buf[0] = res;
25        write_cmd(buf, 1);
26      }else if (fn == 3) {
27        char content[11];
28        res = read_from_zigbee(arg, content);
29        for (int i = 0; i < 10; i++){
30          buf[i] = (byte)(content)[i];
31        }
32        write_cmd(buf, 10);
33      }else if (fn == 4) {
34        char *ps = buf;
35        char sendStr[11];
36        char tempdata[9];
37        for(int i=0; i<10; i++){
38            tempdata[i] = ps[i+2];
39        }
40        sprintf(sendStr, "%s%s", tempdata, "\n");
41        res = write_to_zigbee(arg, sendStr);
42        buf[0] = res;
43        write_cmd(buf, 1);
44      }
45    }
46  }
```

```
1  -module(serialport).
2
3  -export([open_port_serial/2, initport/2,
4          read_from_zigbee/2, write_to_zigbee/2, stop/1]).
5
6  open_port_serial(Pid, Portname) ->
7      decode(call_port(Pid, {open_port_serial, Portname})).
8
9  initport(Pid, Portinit) ->
10     call_port(Pid, {initport, Portinit}).
11
12 read_from_zigbee(Pid, Msg) ->
13     call_port(Pid, {read_from_zigbee, Msg}).
14
15 write_to_zigbee(Pid, Msg) ->
16     call_port(Pid, {write_to_zigbee, Msg}).
17
18 call_port(Pid, Msg) ->
19     process_flag(trap_exit, true),
20     Pid ! {self(), {command, encode(Msg)}},
21     receive
22         {Pid, {data, Data}} ->
23             Data;
```

```
24          {'EXIT', Port, _} ->
25              stop(Port),
26              exit(port_terminated)
27      end.
28
29  stop(Port) ->
30      Port ! {self(), close},
31      receive
32          {Port, closed} ->
33              exit(normal)
34      end.
35
36  encode({initport, X}) -> [1, X];
37  encode({open_port_serial, Y}) -> [2, Y, 0];
38  encode({read_from_zigbee, Z}) -> [3, Z];
39  encode({write_to_zigbee, Msg}) -> [4, Msg].
40
41  decode([Int]) -> Int.
```

B.2 Writer

```
1   -module(writer).
2
3   -export([start/2]).
4
5   start(Arg, ExtPrg) ->
6     %erlang:garbage_collect(self()),
7     %io:format("supervisor started writer! Arg=~s ~p~n",
8       %        [Arg, erlang:process_info(self(),memory)]),
9     Pid = spawn_link(fun() -> init(Arg, ExtPrg) end),
10    register(writerloop, Pid),
11    {ok, Pid}.
12
13  init(Arg, ExtPrg) ->
14      PortWriter = open_port({spawn_executable, ExtPrg}, [{packet, 2}
            ]),
15      Serial_fd = serialport:open_port_serial(PortWriter, Arg),
16      case Serial_fd of
17          255 -> exit(normal);
18          _   -> serialport:initport(PortWriter, Serial_fd),
19                 loop(PortWriter, Serial_fd)
20      end.
21
22  loop(PortWriter, Serial_fd) ->
23      receive
24          {msgToWriter, Msg} ->
25              io:format("~s~n", [Msg]),
26              serialport:write_to_zigbee(PortWriter, [Serial_fd, Msg,
                    0])
27      end,
28      loop(PortWriter, Serial_fd).
```

B.3 Reader

```
1  -module(reader).
2
3  -export([start/2]).
4
5  start(Arg, ExtPrg) ->
6      %erlang:garbage_collect(self()),
7      %io:format("supervisor started reader! ~p~n", [erlang:
           process_info(self(),memory)]),
8      Pid = spawn_link(fun() -> init(Arg, ExtPrg) end),
9      {ok, Pid}.
10
11 init(Arg, ExtPrg) ->
12     PortReader = open_port({spawn_executable, ExtPrg}, [ {packet, 2
           } ]),
13     Serial_fd = serialport:open_port_serial(PortReader, Arg),
14     case Serial_fd of
15         255 -> exit(normal);
16         _   -> serialport:initport(PortReader, Serial_fd),
17                loop(PortReader, Serial_fd)
18     end.
19
20 loop(PortReader, Serial_fd) ->
21     Msg = serialport:read_from_zigbee(PortReader, Serial_fd),
22     writerloop ! {msgToWriter, Msg},
23     loop(PortReader, Serial_fd).
```

B.4 Datagen

```
1  -module(datagen).
2
3  -export([start/1]).
4
5  start([Time]) ->
6    %erlang:garbage_collect(self()),
7    %io:format("supervisor started datagen! ~p~n", [erlang:
         process_info(self(),memory)]),
8    Pid = spawn_link(fun() -> loop(0, Time) end),
9    {ok, Pid}.
10
11 loop(N, Time) ->
12   writerloop ! {msgToWriter, ("1 Msg:" ++ integer_to_list(N+100))},
13   timer:sleep(Time),
14   loop(((N+1) rem 899), Time).
```

B.5 Supervisors

```
1  -module(worker_sup).
2  -behaviour(supervisor).
3
4  -export([start_link/3, start/1]).
5  -export([init/1]).
6
7  start(Time) ->
8      start_link("/dev/ttyUSB0", "readerwriterprg", Time).
9
10 start_link(Arg, ExtPrg, Time) ->
11   supervisor:start_link({local,?MODULE}, ?MODULE, [Arg, ExtPrg,
       Time]).
12
13 init([Arg, ExtPrg, Time]) ->
14     SupFlags = {one_for_all, 1, 5},
15     ChildSpecs = [{writer,
16                    {writer, start, [Arg, ExtPrg]},
17                    permanent,
18                    1000,
19                    worker,
20                    [writer]},
21                   {subworker_sup,
22                    {subworker_sup, start_link, [Arg, ExtPrg, Time]}
                         ,
23                    permanent,
24                    1000,
25                    supervisor,
26                    [subworker_sup]}],
27     {ok, {SupFlags, ChildSpecs}}.
```

```
1  -module(subworker_sup).
2  -behaviour(supervisor).
3
4  -export([start_link/3]).
5  -export([init/1]).
6
7  start_link(Arg, ExtPrg, Time) ->
8    supervisor:start_link({local,?MODULE}, ?MODULE, [Arg, ExtPrg,
       Time]).
9
10 init([Arg, ExtPrg, Time]) ->
11     SupFlags = {one_for_one, 1, 5},
12     ChildSpecs = [{reader,
13                    {reader, start, [Arg, ExtPrg]},
14                    permanent,
15                    1000,
16                    worker,
17                    [reader]},
18                   {datagen,
19                    {datagen, start, [Time]},
20                    permanent,
21                    1000,
22                    worker,
23                    [datagen]}],
24     {ok, {SupFlags, ChildSpecs}}.
```

Appendix C

Data collection program

```
1
2  #include <stdlib.h>
3  #include <string.h>
4  #include <time.h>
5  #include <unistd.h>
6  #include <stdio.h>
7
8  int runps(char *arg, char *drs, char *rss)
9  {
10     FILE *fp;
11     char path[1035];
12
13     char command[100];
14     sprintf(command, "ps v -p %s", arg);
15
16     /* Open the command for reading. */
17     fp = popen(command, "r");
18
19     if (fp == NULL)
20     {
21         printf("Failed to run command\n" );
22         exit(1);
23     }
24
25     char seps[]   = " ";
26     char *token;
27     int test =  0;
28     /* Read the output a line at a time - output it. */
29
30     while (fgets(path, sizeof(path)-1, fp) != NULL)
31     {
32        // printf("%s", path);
33
34         if(test == 1)
35         {
36             int i = 0;
37             token = strtok( path, seps );
38             while( token != NULL )
39             {
40                 /* While there are tokens in "string" */
41                 //printf( " %s\n", token );
42                 if(i == 6) {
```

```
43                      const char *ptr = token;
44                      strcpy(drs, ptr);
45                  } else if (i == 7){
46                      const char *ptr = token;
47                      strcpy(rss, ptr);
48                  }
49                  /* Get next token: */
50                  token = strtok( NULL, seps );
51                  i++;
52              }
53
54          } else {
55              test = 1;
56          }
57      }
58      /* close */
59      pclose(fp);
60      return 0;
61  }
62
63  int runps2(char *arg, char *cpu, char *mem, char *vsz)
64  {
65      FILE *fp;
66      char path[1035];
67
68      char command[100];
69      sprintf(command, "ps u -p %s", arg);
70
71      /* Open the command for reading. */
72      fp = popen(command, "r");
73
74      if (fp == NULL)
75      {
76          printf("Failed to run command\n" );
77          exit(1);
78      }
79
80      char seps[]   = " ";
81      char *token;
82      int test =  0;
83      /* Read the output a line at a time - output it. */
84
85      while (fgets(path, sizeof(path)-1, fp) != NULL)
86      {
87          //printf("%s", path);
88
89          if(test == 1)
90          {
91              int i = 0;
92              token = strtok( path, seps );
93              while( token != NULL )
94              {
95                  /* While there are tokens in "string" */
96                  //printf( " %s\n", token );
97                  if(i == 2) {
98                      const char *ptr = token;
```

```
 99                      strcpy(cpu, ptr);
100                  } else if (i == 3){
101                      const char *ptr = token;
102                      strcpy(mem, ptr);
103                  } else if (i == 4){
104                      const char *ptr = token;
105                      strcpy(vsz, ptr);
106                  }
107                  /* Get next token: */
108                  token = strtok( NULL, seps );
109                  i++;
110              }
111
112          } else {
113              test = 1;
114          }
115      }
116      /* close */
117      pclose(fp);
118      return 0;
119  }
120
121  int runfree(char *used, char *freevar, char *available)
122  {
123      FILE *fp;
124      char path[1035];
125
126      /* Open the command for reading. */
127      fp = popen("free", "r");
128
129      if (fp == NULL)
130      {
131          printf("Failed to run command\n" );
132          exit(1);
133      }
134
135      char seps[]   = " ";
136      char *token;
137      int test =  0;
138      /* Read the output a line at a time - output it. */
139
140      while (fgets(path, sizeof(path)-1, fp) != NULL)
141      {
142          //printf("%s", path);
143
144          if(test == 1)
145          {
146              int j = 0;
147              token = strtok( path, seps );
148              while( token != NULL )
149              {
150                  /* While there are tokens in "string" */
151                  //printf( " %s\n", token );
152                  if(j == 2){
153                      const char *ptr = token;
154                      strcpy(used, ptr);
```

```
155                 } else if (j == 3){
156                     const char *ptr = token;
157                     strcpy(freevar, ptr);
158                 } else if (j == 6) {
159                     const char *ptr = token;
160                     strcpy(available, ptr);
161                 }
162 
163                 /* Get next token: */
164                 token = strtok( NULL, seps );
165                 j++;
166             }
167             test = 0;
168         }
169         else
170         {
171             test = 1;
172         }
173     }
174     /* close */
175     pclose(fp);
176     return 0;
177 }
178 int uptime(char *val)
179 {
180     FILE *fp;
181     char path[1035];
182 
183     /* Open the command for reading. */
184     fp = popen("cat /proc/uptime", "r");
185 
186     if (fp == NULL)
187     {
188         printf("Failed to run command\n" );
189         exit(1);
190     }
191 
192     char seps[]   = " ";
193     char *token;
194     int stop = 0;
195     while ((fgets(path, sizeof(path)-1, fp) != NULL) && stop == 0)
196     {
197         //printf("%s", path);
198 
199         int j = 0;
200         token = strtok( path, seps );
201         while( token != NULL )
202         {
203             /* While there are tokens in "string" */
204             //printf( " %s\n", token );
205             if(j == 0){
206                 const char *ptr = token;
207                 strcpy(val, ptr);
208                 stop = 1;
209                 break;
210             }
```

```
211
212             /* Get next token: */
213             token = strtok( NULL, seps );
214             j++;
215         }
216     }
217     /* close */
218     pclose(fp);
219     return 0;
220 }
221
222 int proct(char *pid, char *value, char *value2)
223 {
224     FILE *fp;
225     char path[1035];
226
227     char command[100];
228     sprintf(command, "cat /proc/%s/stat", pid);
229     /* Open the command for reading. */
230     fp = popen(command, "r");
231
232     if (fp == NULL)
233     {
234         printf("Failed to run command\n" );
235         exit(1);
236     }
237
238     char seps[]   = " ";
239     char *token;
240
241     while (fgets(path, sizeof(path)-1, fp) != NULL)
242     {
243         //printf("%s", path);
244
245         int j = 0;
246         token = strtok( path, seps );
247         while( token != NULL )
248         {
249             /* While there are tokens in "string" */
250             //printf( " %s\n", token );
251             if(j == 13){
252                 const char *ptr = token;
253                 strcpy(value, ptr);
254             } else if(j == 14){
255                 const char *ptr = token;
256                 strcpy(value2, ptr);
257                 break;
258             }
259
260             /* Get next token: */
261             token = strtok( NULL, seps );
262             j++;
263         }
264     }
265     /* close */
266     pclose(fp);
```

```
267     return 0;
268 }
269
270 int cpuUsage(int argc, char *argv[], char *res2)
271 {
272     struct timespec ts;
273     ts.tv_sec  = 1;
274     ts.tv_nsec = 0;
275     double utime=0, ctime=0, o_utime=0, o_ctime=0;
276     char utimestr[20], ctimestr[20], o_utimestr[20], o_ctimestr
        [20], time[20], time2[20];
277     for(int j=0; j<argc; j++){
278         proct(argv[j], o_utimestr, o_ctimestr);
279         o_utime += atof(o_utimestr);
280         o_ctime += atof(o_ctimestr);
281     }
282     uptime(time);
283     nanosleep(&ts, NULL);
284
285     for(int j=0; j<argc; j++){
286         proct(argv[j], utimestr, ctimestr);
287         utime += atof(utimestr);
288         ctime += atof(ctimestr);
289     }
290     uptime(time2);
291     double res = ((utime - o_utime)+(ctime - o_ctime) / (atof(time2
        )-atof(time)));
292     sprintf(res2, "%lf", res);
293     return 0;
294 }
295
296 void fetchIds(char *writerId, char *readerId, char *datagenId)
297 {
298     char path1[1035],path2[1035],path3[1035];
299
300     FILE *p1 = popen("ps -A | grep writer", "r");
301     fgets(path1, sizeof(path1)-1, p1);
302     const char *ptr = strtok( path1, " " );
303     strcpy(writerId, ptr);
304     printf("%s writerId\n", writerId);
305
306     FILE *p2 = popen("ps -A | grep reader", "r");
307     fgets(path2, sizeof(path2)-1, p2);
308     const char *ptr2 = strtok( path2, " " );
309     strcpy(readerId, ptr2);
310     printf("%s readerId\n", readerId);
311
312     FILE *p3 = popen("ps -A | grep datagen", "r");
313     fgets(path3, sizeof(path3)-1, p3);
314     const char *ptr3 = strtok( path3, " " );
315     strcpy(datagenId, ptr3);
316     printf("%s datagenId\n", datagenId);
317
318     pclose(p1);
319     pclose(p2);
320     pclose(p3);
```

```
321 }
322
323 void fetchIdsErl(char *erts, char *child, char *readerwriter1, char
        *readerwriter2)
324 {
325     char path1[1035],path2[1035],path3[1035];
326
327     FILE *p1 = popen("ps x | grep bin/beam", "r");
328     fgets(path1, sizeof(path1)-1, p1);
329     //printf("%s", path1);
330     //strtok( path1, " " );
331     const char *ptr = strtok( path1, " " );
332     strcpy(erts, ptr);
333     printf("%s erts\n", erts);
334
335     FILE *p2 = popen("ps x | grep erl_child", "r");
336     fgets(path2, sizeof(path2)-1, p2);
337     const char *ptr2 = strtok( path2, " " );
338     strcpy(child, ptr2);
339     printf("%s erl_child\n", child);
340
341     FILE *p3 = popen("ps x | grep readerwriterprg", "r");
342     fgets(path3, sizeof(path3)-1, p3);
343     //printf("%s", path3);
344     const char *ptr3 = strtok( path3, " " );
345     strcpy(readerwriter1, ptr3);
346     printf("%s readerwriter1\n", readerwriter1);
347     fgets(path3, sizeof(path3)-1, p3);
348     //printf("%s", path3);
349     const char *ptr4 = strtok( path3, " " );
350     strcpy(readerwriter2, ptr4);
351     printf("%s readerwriter2\n", readerwriter2);
352
353     pclose(p1);
354     pclose(p2);
355     pclose(p3);
356 }
357
358 int main(int argc, char *argv[] )
359 {
360     struct timespec ts;
361     ts.tv_sec  = 5;
362     ts.tv_nsec = 100;
363     struct timespec ts2;
364     ts2.tv_sec  = 1;
365     ts2.tv_nsec = 0;
366
367     char filename[50];
368
369     int i = 0;
370     char cpustr[20], memstr[20], vszstr[20], drsstr[20], rssstr
            [20], used[20], free[20], available[20], cpuproc[20];
371     if(atoi((argv)[1]) == 1)
372     {
373         char *fileArr[] = {"10s", "5s","2s","1s","750ms","500ms","
                350ms","250ms","200ms","175ms","150ms","125ms","100ms"};
```

```
374         int timeSArr[] = {10, 5, 2, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0};
375         int timeMsArr[] = {0, 0, 0, 0, 750, 500, 350, 250, 200,
                175, 150, 125, 100};
376
377         int timeErlMsArr[] = {10000, 5000, 2000, 1000, 750, 500,
                350, 250, 200, 175, 150, 125, 100};
378
379         for(int x=0; x<10; x++)
380         {
381             printf("Iteration x=%d\n", x );
382             for(int y=0; y<13; y++)
383             {
384                 FILE *pp = popen("./../clientserver/client", "r");
385                 pclose(pp);
386
387                 i=0;
388                 sprintf(filename, "../files/%darm_cpp_%s.txt", x,
                        fileArr[y]);
389                 FILE *f = fopen(filename, "w");
390                 if (f == NULL)
391                 {
392                     printf("Error opening file!\n");
393                     exit(1);
394                 }
395                 fprintf(f, "CPUProc\t\t\tCPU  \t\t\tMEM  \t\t\tVSZ
                        \t\tDRS  \t\tRSS  \t\tUSED  \t\tFREE  \t\
                        tAVAILABLE\t\n");
396                 fclose(f);
397                 FILE *p1 = popen("./../cpp_prototype/writer; ", "r"
                        );
398                 nanosleep(&ts, NULL);
399                 FILE *p2 = popen("./../cpp_prototype/reader", "r");
400
401                 char command2[100];
402                 sprintf(command2, "./../cpp_prototype/datagen %d %d
                        ", timeSArr[y], timeMsArr[y]);
403                 FILE *p3 = popen(command2, "r");
404
405                 char writerId[20], readerId[20], datagenId[20];
406                 fetchIds(writerId, readerId, datagenId);
407
408                 char *idsP[] = {writerId, readerId, datagenId};
409
410                 while(i < 120)
411                 {
412                     printf("cppIteration %d\n", i );
413                     int drsint=0, rssint=0, vszint = 0;
414                     double cpuvalue = 0, memvalue = 0;
415                     for(int j=0; j<3; j++){
416                         runps(idsP[j], drsstr, rssstr);
417                         runps2(idsP[j], cpustr, memstr, vszstr);
418                         drsint += atoi(drsstr);
419                         rssint += atoi(rssstr);
420                         cpuvalue += atof(cpustr);
421                         memvalue += atof(memstr);
422                         vszint += atoi(vszstr);
```

```
423                     }
424                     runfree(used, free, available);
425
426                     cpuUsage(3, idsP, cpuproc);
427                     FILE *ff = fopen(filename, "a");
428                     if (ff == NULL)
429                     {
430                         printf("Error opening file!\n");
431                         exit(1);
432                     }
433                     fprintf(ff, "%.3f \t\t\t%.3f\t\t\t%.3f\t\t\t%d
                            \t\t%d \t\t%d \t\t%s \t\t%s \t\t%s\t\n
                        ", atof(cpuproc),cpuvalue, memvalue, vszint,
                            drsint, rssint, used, free, available);
434                     fclose(ff);
435                     i++;
436                 }
437                 FILE *p11 = popen("pkill -f datagen", "r");
438                 pclose(p11);
439                 FILE *p21 = popen("pkill -f reader", "r");
440                 pclose(p21);
441                 FILE *p31 = popen("pkill -f writer", "r");
442                 pclose(p31);
443
444                 pclose(p1);
445                 pclose(p2);
446                 pclose(p3);
447
448                 FILE *pp1 = popen("./../clientserver/client", "r");
449                 pclose(pp1);
450
451
452                 sprintf(filename, "../files/%darm_erl_%s.txt", x,
                        fileArr[y]);
453
454
455                 char appFile[300];
456                 sprintf(appFile, "{application, app, [{vsn,
                        \"1.0.0\"}, {modules, [app, worker_sup,
                        subworker_sup, writer, reader, datagen,
                        serialport]}, {register, [app]}, {mod, {app, [%d
                        ]}}]}.", timeErlMsArr[y]);
457                 char filename3[30];
458                 sprintf(filename3, "../Erl_prototype2/app.app");
459                 FILE *f3 = fopen(filename3, "w");
460                 if (f3 == NULL)
461                 {
462                     printf("Error opening file!\n");
463                     exit(1);
464                 }
465                 fprintf(f3, "%s", appFile);
466                 fclose(f3);
467                 FILE *f2 = fopen(filename, "w");
468                 fprintf(f2, "CPUProc\t\t\tCPU \t\t\tMEM \t\t\tVSZ
                        \t\tDRS \t\tRSS \t\tUSED \t\tFREE \t\
                        tAVAILABLE\t\n");
```

```
469
470                 if (f2 == NULL)
471                 {
472                     printf("Error opening file!\n");
473                     exit(1);
474                 }
475                 fclose(f2);
476                 char command3[100];
477                 sprintf(command3, "cd ../Erl_prototype2; erl -
                        noshell -eval \"application:start(app)\" +P 1024
                        +Q 1024 +Mea min +L");
478                 FILE *p4 = popen(command3, "r");
479                 nanosleep(&ts, NULL);
480                 char erts[20], child[20], readerwriter1[20],
                        readerwriter2[20];
481                 fetchIdsErl(erts, child, readerwriter1,
                        readerwriter2);
482
483                 char *idsP2[] = {erts, child, readerwriter1,
                        readerwriter2};
484                 i = 0;
485                 while(i < 120)
486                 {
487                     printf("erllIteration %d\n", i );
488                     int drsint=0, rssint=0, vszint = 0;
489                     double cpuvalue = 0, memvalue = 0;
490                     for(int j=0; j<4; j++){
491                         runps(idsP2[j], drsstr, rssstr);
492                         runps2(idsP2[j], cpustr, memstr, vszstr);
493                         drsint += atoi(drsstr);
494                         rssint += atoi(rssstr);
495                         cpuvalue += atof(cpustr);
496                         memvalue += atof(memstr);
497                         vszint += atoi(vszstr);
498                     }
499                     runfree(used, free, available);
500                     cpuUsage(4, idsP2, cpuproc);
501                     FILE *f3 = fopen(filename, "a");
502                     fprintf(f3, "%.3f  \t\t\t%.3f\t\t\t%.3f\t\t\t%d
                            \t\t%d  \t\t%d  \t\t%s  \t\t%s  \t\t%s\t\n
                            ", atof(cpuproc),cpuvalue, memvalue, vszint,
                             drsint, rssint, used, free, available);
503                     fclose(f3);
504                     i++;
505                     //nanosleep(&ts, NULL);
506                 }
507                 char command[20];
508                 sprintf(command, "kill %s", erts);
509                 FILE *p41 = popen(command, "r");
510                 sprintf(command, "kill %s", child);
511                 FILE *p42 = popen(command, "r");
512                 sprintf(command, "kill %s", readerwriter1);
513                 FILE *p43 = popen(command, "r");
514                 sprintf(command, "kill %s", readerwriter2);
515                 FILE *p44 = popen(command, "r");
516
```

```
517                     pclose(p4);
518                     pclose(p41);
519                     pclose(p42);
520                     pclose(p43);
521                     pclose(p44);
522                 }
523             }
524         }
525         else
526         {
527             sprintf(filename, "../files/arm_%s.txt", "no_prg");
528             printf("No prg\n");
529             FILE *f = fopen(filename, "w");
530             fprintf(f, "CPUProc\t\t\tCPU \t\t\tMEM \t\t\tVSZ \t\tDRS
                    \t\tRSS \t\tUSED \t\tFREE \t\tAVAILABLE\t\n");
531
532             if (f == NULL)
533             {
534                 printf("Error opening file!\n");
535                 exit(1);
536             }
537             while(i < 120)
538             {
539                 printf("Iteration: %d\n", i);
540                 runfree(used, free, available);
541                 char s[10] = "      ";
542                 fprintf(f, "%s\t\t%s\t\t%s\t\t%s\t\t%s\t\t%s\t\t%s\t\t%
                        s\t\t%s\t\n", s, s, s, s, s, s, used, free,
                        available);
543                 i++;
544                 nanosleep(&ts2, NULL);
545             }
546             fclose(f);
547         }
548         printf("File is created\n");
549         return 0;
550     }
```