# A study on the effects on maintainability the libuv library has on ARM-based IoT-devices

–

Natanael Log

Supervisor : Petru Eles
Examiner : Petru Eles

External supervisor : Pär Wieslander

# Abstract

Abstract.tex

# Acknowledgments

`Acknowledgments.tex`

# Contents

# List of Figures

# List of Tables

# Todo list

# 1    **Introduction**

The Internet of Things (IoT) is a new dimension of the internet which includes *smart devices* - physical devices able to communicate with the outside world through internet [1]. Ericsson foretold there will be around 400 million smart devices by the end of 2016 [2] and a great challenge will be to serve all of these. IoT applications are already influencing our everyday lives; smart cities, smart grids and traffic management are only a few of the large variety of applications in this "multi-tiered heterogeneous system based on open architectural platforms and standards" [3]. The complexity of a smart device varies from passive ID-tags which communicates through near field communication to full scale computers with multithreaded operating systems.

Advanced RISC Machine (ARM) processors are used in different types of electronic devices. RISC is short for *Reduced Instruction Set Computer* and its supported instructions are simple, have uniform length and almost all instructions execute in one clock cycle. This reduces the complexity of the chip and gives more room to performance enhancing features [4]. In an IoT context, ARM processors can be a good choice to be embedded on a smart device due to its large variety in cost, energy consumption, performance and size.

From a network perspective, smart devices can be seen as clients in a traditional client-server architecture. They emit data to a server through an internet protocol. The server receives the data and processes it. When building these kinds of network services it is important for the server to avoid blocking concurrent requests [5]. These requests are both on a network level but also on the general operating system level. Requests from a smart device might be logged on a file or written to a database, meaning the server must start new processes to handle these requests without blocking new requests from the network. A server does not necessarily have to be a web server serving requests on the internet protocol. The operating system itself can be viewed as an I/O (input/output) server receiving requests from the I/O layer. This can be data on serial ports and general purpose I/O pins on the processor. A device monitoring a physical environment using sensors can take use of this feature by letting the registers call interrupt handlers when they are ready for reading or writing. Thus an application running on an embedded system listening for data from sensors can be viewed as a web server listening for internet requests. The same challenge regarding non-blocking request handlers holds also for these mentioned applications.

To implement a non-blocking server, the *libuv* C library can be a good choice. With it the user can implement an *event driven, reactive* architecture with request handlers reacting to certain events in the system, e.g. I/O events from the operating system. [6]

To determine the maintainability of a software system one can, among others, look at the source code and its age since release, the number of non-commented source code statements and its rate of failures per time unit. [7]

## 1.1 Motivation

Since IoT is growing and communication will largely increase, efficiency and performance is an important factor. There exists a number of communication protocols for IoT network architectures. A big challenge is to support massive data streams with minimal overhead in transport. However, this implies for the transport and communication perspective on IoT, but another significant perspective is I/O (file writes, database queries) processing on the machines themselves. If the demand for high performance IoT services is increasing, then development towards embedded systems will too. A rigor evaluation on two common architectures can therefore be of great value to find what suits best in IoT.

libuv is the major subsystem to *Node.js*, a popular universal language for *"front end, back end and connected devices [and] everything from the browser to your toaster"* [8]. Due to Node.js' event-driven development style many developers in its field might have a better experience using libuv when if need to transfer to embedded programming will be in question.

As software maintenance is a fundamental part of the software life cycle and is becoming a more important factor for businesses and open source projects for building products that can be further developed by others, it is of high interest to study the maintainance aspect of a libuv implementation [9].

## 1.2 Aim

To find whether libuv is a good option in terms of maintainability when selecting underlying architecture for IoT devices.

## 1.3 Research questions

1. Given a state of the art reactive system from the industry - how does an equivalent implementation using libuv compare in terms of software maintainability?

## 1.4 Delimitations

Maintainability will be the main metric used. Other metrics might be useful for evaluation of the architectures, but due to time limitations only maintainability will be used. Only one or two types of devices will be used for implementation and testing. This might decrease data resolution.

# 2 Background

This thesis was proposed by the author and accepted by the IT consulting firm Attentec in Linköping, Sweden. A personal interest in embedded Linux, NodeJS and reactive architectures (as well as an interest in the company) led the author to Attentec and a proposal of this study.

Attentec resides in Linköping and Stockholm in Sweden and in Oslo, Norway. They focus on three major business areas: *Internet of Things*, *Streaming Media* and *Modern Software Development*. Their interest in this study lies in expanding their knowledge base regarding Internet of Things.

# 3 Theory

Theories used in this study are presented here. They include software development methodology, software metrics, reactive systems and some general information regarding the *libuv* library.

## 3.1 Internet of Things

The Internet of Things (IoT) is a relatively new paradigm of the internet where not only people are interconnected, but *everything* is. The term was first heard in 1999 in a presentation by Kevin Ashton [10] and the popularity has since blossomed [11]. IoT can be conceptualized as context-aware *things* communicating with each other and human-centered applications. The enabling applications are vast and include domains from many areas of society - transportation, healthcare, homes, offices and social domains to name a few. A good example is electricity consumption control in homes. Electricity prices can be monitored and utilities such as washing machines and heating can be optimized and only used when prices are low [12]. IoT can typically be explained as "one paradigm with many visions" [12] as there is no official definition of the term. IoT is more than ubiquitous computing, embedded devices and applications. It should not only be accessible by a small group of stakeholders but embedded in today's open internet infrastructure. Referring to logistics an IoT definition can be viewed as *"[asking] for the right product in the right quantity at the right time at the right place in the right condition and at the right price"* [13]. Uckelmann et. al. discusses these approaches and defines IoT as:

*"Internet of Things links uniquely identifiable things to their virtual representations in the Internet containing or linking to additional information on their identity, status, location or any other business, social or privately relevant information at a financial or non-financial pay-off that exceeds the efforts of information provisioning and offers information access to non-predefined participants. The provided accurate and appropriate information may be accessed in the right quantity and condition, at the right time and place at the right price. The Internet of Things is not synonymous with ubiquitous / pervasive computing, the Internet Protocol (IP), communication technology, embedded devices, its applications, the Internet of People or the Intranet / Extranet of Things, yet it combines aspects and technologies of all of these approaches."* [13].

A bit messy intro, perhaps mention different viewpoints of IoT and try to find common demoninators among them.

4

### 3.1.1 Enabling technologies and challenges

The state-of-the-art technologies used today in the IoT context include sensoring equipment and networks such as the *Radio Frequency Identification* (RFID) and the *Wireless Sensor Network* (WSN), communication techniques like the *IPv6 over Low-Power Wireless Personal Area Network* (6LoWPAN) and *Representational State Transfer* (REST) and architectural approaches like *middleware* and *gateways*.

The RFID is a technique used to identify an object by letting a reader generate an electromagnetic wave which in turn generates a current in a microchip that transmits a code back to the reader. The code, also known as an *Electronic Product Code* (EPC), can be used to identify the object. EPC is developed by EPCglobal, a global non-profit organization, with the purpose to spread the use of RFID and create global standards for modern businesses [12]. An RFID tag can be passive, i.e. work without battery power, or active, i.e. work with battery power. It basically consists of an antenna, a microchip attached to the antenna and some form of encapsulation. They vary in size and shape; from centimeter long 3D-shaped capsules to 2D-shaped stickers. The fact that they can work without a battery, meaning they theoretically can work forever, make them very practical to use in different environments. [14]

In a survey from 2001, Akyildiz et. al. [15] presents a *Wireless Sensor Network* containing hardware nodes capable of monitoring its environment and transmit the received data to a server or host acting as a sink. The sensor could be one of many types and be able to measure temperature, humidity, movement, lightning condition, noise levels etc. This leads into one of the major challenges in IoT: communication and networking. Following the semantical meaning of IoT, which states *"[it is] a world wide network of interconnected objects uniquely addressable, based on standard communication protocols"* [16], it is clear that every node will produce its own content and should be retrievable regardless of location. This puts pressure on the identification methods and the overall network infrastructure for IoT. As remaining unused IPv4 addresses are closing in to zero, the IPv6 protocol can be a natural next step for IoT devices to use [12]. 4 bytes are used to address devices in IPv4, wich means the protocol can support around 4 billion devices. IPv6 on the other hand use 16 bytes to address devices, meaning around $10^{38}$ devices can be uniquely addressed.

6LoWPAN is a wireless technology suitable for resource-constrained nodes in the IoT network. It is designed for small packet sizes, low bandwidth, low power and low cost. It assumes a large amount of nodes will be deployed and that they can be unreliable in terms of radio connectivity and power drain, thus making this technology a good choice for wireless sensor nodes. [17]

REST?

### 3.1.2 Gateways

A big challenge in IoT is, as described earlier, to connect every single device to the internet. As for example devices in a WSN are deployed they do not necessarily have the power and capacity to directly communicate to the internet. A common approach to solve this problem is to add a gateway between the devices and the internet, acting as an amplifier for the transmissions from the devices [18]. Chen et. al. [19] describes three layers of domains in the IoT architecture: the *sensing domain*, the *network domain* and the *application domain*. The sensing domain is where the primitive communication between the actual devices in the network happen; for instance RFID and 6LoWPAN. This domain is layered under the network domain and each piece of information sent from the sensing domain is aggregated, filtered and wrapped in the network domain before being sent to the application domain. Note that the information protocols used between the sensing- and network domain and the network- and application domain does not necessarily match. As some protocols are more suited for resource-constrained devices, they might require modification to be able to communicate with more standardized protocols, like the *Internet Protocol* (IP). E.g. the EPC code generated by

the RFID can not be put on an IP stack as is, but it could be mapped to an IPv6 address via a gateway and thereby be identified and usable through the internet [20].

The gateways reside in the network domain. They act as a bridge between the devices and the internet. The approaches towards implementing IoT gateways vary

Should list different approaches in the industry, from large companies to smaller projects.

## 3.2 Event-driven architectures

In software architecture, the term *event-driven*, or *implicit invocation*, is used to express the mechanism of function invocation based on events [21]. In contrast, *explicit invocation* refers to a traditional function invocation by function name, e.g. `foo()`. However, a function can be mapped to an event so that when the event occur, the function is invoked implicitly. Hence the name implicit invocation. The announcers of events do not know what functions are registered to the event and cannot make assumptions on event ordering or how data will be transformed due to the event. This is one disadvantage of this architecture [21]. However, in high-performance, I/O intensive networking systems, it is shown that event-based approaches perform much better [22].

### 3.2.1 Task management and stack management

A program task is an encapsulation of control flow with access to some common shared state. This can be viewed as a closure in a program language or a function. Managing these tasks can be done in several ways. In *preemptive* task management, tasks can be scheduled to interleave and make room for other tasks or be run in parallell on multicore systems. In *serial* task management, tasks are run until they are finished before any other task can start its execution. A hybrid approach is *cooperative* task management. Tasks can yield control to other tasks in defined points in its execution. This is useful in I/O intensive systems where tasks can make way for other tasks while waiting for I/O. Event-driven systems usually implements the cooperative approach. Events, e.g. network requests, are handled by calling an associated function. A benefit with event-driven systems that run on a single thread is that mutual exclusion conflicts never occur. However, the state in which the task was in before yielding control to another task is not necessarily the same as when the task resumes control. [23]

There are two categories of I/O management: *synchronous* and *asynchronous* [23]. If a function calls an I/O operation and then blocks the rest of the execution while waiting for the I/O operation to finish, the function is synchronous. Here is an example of a synchronous read file-function from the Node.js documentation [24]:

Listing 3.1: Synchronous I/O example in Node.js.

```
const fs = require('fs');
const file = fs.readFileSync('path/to/file');

console.log(file);
console.log('end of code');
```

The `readFileSync()` function blocks the rest of the execution while the file is being read, i.e. the file content will be printed before *"end of code"*. In asynchronous I/O, the function calling the I/O operation returns immediately but is provided a function to be run when the I/O is ready. Here is an equivalent implementation but with an asynchronous function:

Listing 3.2: Asynchronous I/O example in Node.js.

```
const fs = require('fs');
```

```
function callback(err, file) {
  if (err) throw err;
  console.log(file);
}

fs.readFile('path/to/file', callback);
console.log('end of code');
```

`readFile()` does not in this case return anything, but instead let the event handling system invoke the callback function when the I/O is ready. The execution continues directly, so *"end of code"* will be printed before the file content.

An interesting notice to Listing 3.2 is that, say a global state is introduced and is dependent upon by the callback function, there is no guarantee the state remains unchanged between the `readFile()`- and the `callback()`-invocation. This is similar to the mutual exclusion problem in preemeptive task management where multiple agents wants to access the same resource concurrently, and problems like read/write conflicts can occur. But since cooperative task management not necessarily infer multithreaded environments, concurrent problems can be avoided. There are however other problems related to understanding and scalability of the code. In Listing 3.1 one can observe that all operations happen in the same scope. This is called *automatic stack management* and it means the task is written as a single procedure with synchronous I/O [23]. The current state of the program is always kept in the local stack and there is no possibility it can be mutated by other parties. In *manual stack management* the task is ripped into callbacks, or *event handlers*, which are registered to the event handling system. This can have a negative impact on the control flow and the global state of the program as tasks are being broken into functions with separate scopes.

## 3.3 Reactive systems

Systems required to continually respond to its environment are called *reactive* systems. Harel and Pnueli [25] elaborates on different system dichotomies, for instance deterministic and non-deterministic systems. Deterministic systems have unique actions which always generates the same output. Nondeterministic systems do not have that property and it can cause them to be more difficult to handle. Another dichotomy is sequential and concurrent systems. Concurrent systems cause problems which are easily avoidable in sequential ones. For instance the mutual exclusion problem where it is not always trivial what process owns what resource and whether it is safe to modify a resource that can be used by another process. A more fundamental dichotomy is presented by Harel and Pnueli: *transformational* and *reactive* systems. A transformational system transforms its input and creates new output. It can continually prompt for new input. A reactive system is prompted the other way around: from the outside. It does not necessarily compute any input, but it maintains a relationship with its outside environment. A reactive system can be deterministic or nondeterministic, sequential or concurrent, thus making the transformational/reactive system dichotomy a fundamental system paradigm. [25]

### 3.3.1 Specifications for reactive systems

Harel and Pnueli [25] presents a method for decomposing complex system using *statecharts*. Drawn from the theories of *finite state automata*, a system can be expressed using states. This is particularly useful for reactive systems that can react to a large amount of different inputs. One powerful feature with statecharts is its ability to handle hierarchical states. If a system has a numerous amount of input combinations, a large set of states must be created which in the long run is not appropiate in terms of scalability. One solution to this is hierarchical states in which a state can hold a number of other states. Fundamentally, a state undergo a transition to another state when an action is performed on that state. [25]

Ardis et al [26] created a specification for a telephone switching system which later was tested on a number of languages. They started by describing some general properties of the system in a list. In general, the list had the format:

1. When event $X_1$ occurs,

    a) If some condition or property of the system holds true, call action $Y_1$

    b) Otherwise, call action $Z_1$

2. When event $X_2$ occurs,

    a) Call action $Y_2$

3. And so on...

They then implemented a system fullfilling the requirements in a set of different languages. Most languages fit well for reactive systems since their semantic allows and makes it easy to build and define different state machines. They also did one implementation in C in which they had two solutions: one using arrays to hold all the different actions and states and one using switch-blocks.

However, specification of primitive components can still be a challenge. Non-trivial steps might be required to transition from one state to another. Hummel and Thyssen [27] presents a formal method to specify a reactive system using stream based I/O-tables. In addition to generating a simple and understandable description of the system, the method helps formalize inconsistent requirements.

## 3.4 Reactive programming

Reactive programming has been proposed to be the solution to implement event-driven architectures [28]. It is derived from the *Synchronous Data Flow* (SDF) paradigm [29]. SDF is a data flow abstraction used to describe parallell computation. Functions are presented as nodes in a directed graph and the arcs represent data paths. Nodes can execute its computation whenever data is available on its incoming arcs, and nodes with no incoming arcs can execute at any time. This leads to concurrency since multiple nodes can execute in the same time. One constraint on this system is that of *side-effects*, which are not allowed. Nodes cannot mutade any resource accessible by another node unless they are explicitly connected by an arc.

In the reactive programming paradigm, the notion of time is abstracted away. In the same way memory can be abstracted away by garbage collection in languages like Java, the programmer does not have to instruct the program *when* things will execute, but rather *how* [28]. Data dependencies and change propagation can be handled automatically by the language. Consider a simple example:

```
int i = 1;
int j = 1;
int k = i + j;
```

In a language like C, $k$ will be computed to the value 2. If $i$ later on is redefined to another value, say 3, $k$ will still hold the same value 2. In a reactive setting, $k$ will be updated with the latest value of either $i$ or $j$, even if they change later on in time. It can be viewed such that $k$ is *dependent* on $i$ and $j$. [28]

Reactive programming has two distinguishing features: *behaviors* and *events*. Behaviors refer to time-varying vales, e.g. time itself. Events on the other hand, happen in discrete points in time and describes what happened. This could be a file that is ready to be written to, a network request that is available or a keyboard button that was pressed. [28]

### 3.4.1 A taxonomy

Bainomugisha et. al. [28] presents a taxonomy with six fundamental properties considered important features in reactive languages.

#### Basic abstractions

Imperative languages holds basic abstractions in the form of primitives such as primitive operators and values. Reactive languages holds basic abstractions such as behaviors and events.

#### Evaluation model

When an event occur, the evaluation model of a reactive language determines how change is propagated throughout the data dependencies. Propagatation can either be *push-based*, meaning that data is pushed to its dependencies as soon as an event occur, or *pull-based*, meaning that computation nodes pull data when needed. The former is a *data driven* model and is often implemented using callbacks. A challenge is to find out what nodes does not need to be recomputed. The pull-based model is said to be *demand driven* as data is propagated upon request. A disadvantage with this model is that delays between events and reactions can be long.

#### Glitch avoidance

A *glitch* is when inconsistencies in the data occur due to the way updates are propagated among the dependencies. Consider the following example:

```
int i = j;
int k = i + j;
```

$k$ is dependent on $i$ and $j$ and $i$ is on another level dependent on $j$. If $j$ is updated, the case can be that $k$ is updated before $i$, thus leading to an inconsistency between $k$ and $i$. Here is an example:

```
// j == 1
int i = j;      // i == 1
int k = i + j;  // k == 2

// j == 2, k updated before i
int i = j;      // i == 1
int k = i + j;  // k == 3, should be 4
```

These kinds of inconsistencies, or glitches, should be avoided by the language.

#### Lifting operators

*Lifting* in reactive programming refers to the act of making previously uncompliant operators or functions work with behaviors. Lifting transforms the type signature of the function and registers the topology of its dependencies in the dataflow graph. A function $f$ with a non-behavior type $T$ can be lifted as:

```
f(T) -> flift(Behavior<T>)
```

Where *Behavior* is a behavior-type holding a type $T$. In Flapjax, an reactive language built upon Javascript, lifting is performed on multiple places to support behaviors [30]. Here is an example from the language:

```
let time = timerB(1000);
```

This looks like traditional, imperative Javascript. However, $timerB$ with the argument 1000 starts a timer that yields every second and $time$ will always be kept up to date with the yielded value. Flapjax performs lifting on, among others, the addition (+) operator:

```
let time = timerB(1000) + 1;
```

The compiler transforms the expression using the $liftB$ function, looking somewhat like this:

```
liftB(function(t) { return t + 1 }, timerB(1000));
```

This is called *implicit lifting*, meaning that the programmer does not have to worry about whether the add operator was used on a behavior or not [28].

### Multidirectionality

A property of reactive languages is to support data propagation both from derived data, as well as the data it is derived from. Consider a distance converting function:

```
double miles = km * 1.61;
```

Whenever an updated value on either $miles$ or $km$ is available, the other one is recalculated as well.

### Support for distribution

As interactive applications such that web applications are becoming more popular (whose nature is distributed) it is important for reactive languages to also support distributed mechanisms.

## 3.5 Choosing programming language for embedded systems

In embedded systems programming, both hardware and software is important. Nahas and Maaita [31] mentions a few factors to be considered when choosing a programming language for an embedded system:

- The language must take the resource constraints of an embedded processor into account

- It must allow low-level access to the hardware

- It must be able to re-use code components in the form of libraries from other projects

- It must be a widely used language with good access to documentation and other skilled programmers

There is no scientific method for selecting an appropiate language for a specific project. Selection mostly relies on experience and subjective assessment from developers. It was however shown in 2006 that over 50 % of embedded system projects were developed in C and 30 % were developed in C++. Barr [32] states that the key advantage of C is that it allows the developer to access hardware without loosing the benefits of high-level programming. Compared to C, C++ offer a better object-oriented programming style but can on the other hand be less efficient. [31]

## 3.6 libuv

libuv is an asynchronous I/O cross-platform library written in C. It is one of the major core systems of Node.js, a popular JavaScript runtime engine. libuv's central part is the single-threaded event loop which contains all I/O operations. The event loop notifies any I/O events in a callback fashion using abstractions called *handles* and *requests*. Handles are long-lived structures and are operated on using requests. Requests usually only represent one I/O operation on a handle. Each I/O operation performed on the event loop is *non-blocking*, meaning they instantly return and abstracts away any concurrency necessary. To handle network I/O libuv make use of platform specific functionality, such as *epoll* for Linux. For file I/O libuv utilizes its *thread pool* to make the operation non-blocking. All loops can queue work in this thread pool making it ideal for external services or systems that do not want to block any I/O. libuv include handles for *UDP* and *TCP* sockets, file and file system, *TTY*, timers and *child processes*. [33]
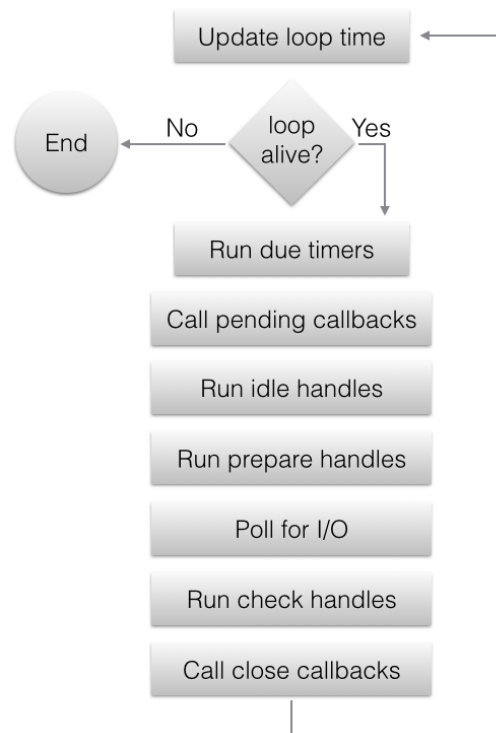


Figure 3.1: The stages each event loop iteration step through in libuv [34].

Figure 3.1 demonstrates the stages each event loop iteration step through. A loop is considered to be alive if there are any active handles or requests. For instance, if a TCP socket is closing its connection the handle structure is freed from the heap and both the request and handle are no longer active. If they were the last being active the loop dies. The steps are described briefly: [33]

1. The loop time is updated and stored for the entire iteration.

2. If the loop is alive the iteration starts, otherwise it dies.

3. All active timers who are scheduled to timeout have their callbacks called.

4. Pending callbacks are called; they are I/O callbacks deferred from the previous iteration.

5. Callbacks registered for idle handles are called. Idle handles are good for low priority activity.

6. Callbacks that should be called right before polling for I/O can be registered with prepare handles, whose callbacks are called here.

7. The time to poll for I/O is calculated here. It either blocks for 0 seconds, the timeout for the closest timer or infinity. Depending on whether there are active idle handles or requests or if the loop is about to close, the time is 0.

8. I/O is polled and the loop is blocked. All handles monitoring I/O will have their callbacks called.

9. Callbacks that should be called right after polling for I/O can be registered with check handles, whose callbacks are called here.

10. Handles being closed have their close callbacks called.

11. If the loop was configured to only run once and no I/O callbacks were called after the poll, timers might be due since time could have elapsed during the poll. Those timers gets their callbacks called.

12. Depending on the loop's configuration and if it is still alive, the loop exits or continue.

The following example provided by Marathe [35] shows an idle handler and its corresponding callback. As seen in Figure 3.1 and explained in step 7 idle handles and their callbacks are called every iteration with 0 poll time.

Listing 3.3: libuv example with an idle handle [35].

```c
#include <stdio.h>
#include <uv.h>

int64_t counter = 0;

// this callback is called every time step 5 is reached
void wait_for_a_while(uv_idle_t* handle) {
  counter++;

  // when the counter is large enough, the handle is closed and the loop will
  // die
  if (counter >= 10e6)
    uv_idle_stop(handle);
}

int main() {
  uv_idle_t idler;

  uv_idle_init(uv_default_loop(), &idler);
  uv_idle_start(&idler, wait_for_a_while);

  printf("Idling...\n");

  // this function will not return until the loop is no longer alive
  uv_run(uv_default_loop(), UV_RUN_DEFAULT);

  uv_loop_close(uv_default_loop());
  return 0;
}
```

libuv includes a number of data types and functions which can be found in its documentation [33].

## 3.7 Software maintainability

Maintaining software has two distinct viewpoints: it could either be the act of refactor and modify existing software to correct faults and improve existing features, or it could point to the attribute of a software system that expresses its ease of modification for the same purpose [36]. Riaz et. al. [37] compiled a list of how other authors that have studied maintainability in different settings defined the term maintainability. The definitions could be *"the ease with which a software application or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment"* and *"how difficult it is to make small or incremental changes to an existing software object without introducing errors in logic or design"* [37].

Previous research show that software process improvements is key for both large and small successful companies. However, it requires a balance between formal process and informal practice. Large companies tend to lean more towards the formality of things and employees are given less space to be creative. On the other hand, small companies tend to do the opposite: employees are given freedom to explore solutions while formal processes are being put aside. Dybå [38] suggests that "formal processes must be supplemented with informal, inter-personal coordination about practice". It is also important to note how failure is handled. Failure is essential for improving learning and questioning the status quo inside both companies and software. If failure is unacceptable, the organizational competance can decrease when facing change in the environment. When a company is successful it will grow larger with repeated success and there is no financial reason to change what already works and generates revenue. However, eventually when the environment changes there might be good reasons to dust off legacy systems and introduce some new patches. The cost of doing so might depend on whether the system was initially built with maintainability in mind. [38]

> Continue here with stuff about metrics in general and the problem with assessing them

### 3.7.1 Metrics

In all engineering fields, measurement is important. Measuring software is a way for an engineer to assess its quality and a large number of software metrics have been derived during the years [39]. Maintainability can be measured in many ways and Aggarwal et al. [39] suggests that some metric results can be derived from others. In the same way the area and diagonal can be derived from the width and height of a table, some metrics may provide redundant information. It is common, when selecting metrics to use in a study, that some metrics can be superfluous in certain contexts or be substituted with similar metrics [40] [41].

> Some sort of leading text to the following subchapters

**Halstead's metrics**

Metrics proposed by Halstead [42] in 1977 are widely used metrics that work both for object-oriented systems and non-object-oriented systems [43] [40]. It consists of six *based* measures and ten *derived* measures [43]. Based measures are measures defined as an attribute and the method used to measure it, and derived measures are defined as functions of base measures [43]. The based measures are:

Number of unique operators $n_1$

Number of unique operands $n_2$

Total number of operators $N_1$

Total number of operands $N_2$

Number of potential operators $n_1^*$

Number of potential operands $n_2^*$

These are self explanatory, except perhaps for the last two. $n_1^*$ and $n_2^*$ are defined as the *minimum* possible number for $n_1$ and $n_2$. This depends on the programming language, but for C, where at least a `main()` function must be defined, $n_1^* = 2$ (the function name and the parantheses symbols) and $n_2^* = 0$ (the number of required arguments to `main()`) [44]. To better understand the measures, consider this sample program in C:

```c
int main() {
    char s[12] = "Hello world!";
    printf(s);
    return 0;
}
```

The unique operators $(n_1)$ are `int`, `main`, `()`, `{}`, `char`, `[]`, `=`, `"`, `;`, `printf` and `return`. The unique operands $(n_2)$ are `s`, `12`, `"Hello world!"` and `0`. So $n_1 = 9$, $n_2 = 4$, $N_1 = 14$, $N_2 = 5$.

The derived measures proposed by Halstead are:

Program length $N = N_1 + N_2$

Vocabulary $n = n_1 + n_2$

Volume $V = N \log_2 n$

Potential volume $V^* = (2 + n_2^*) \log_2(2 + n_2^*)$

Level $L = \dfrac{V^*}{V}$

Difficulty $D = \dfrac{1}{L}$

Level estimator $\hat{L} = \dfrac{2n_2}{n_1 N_2}$

Intelligent content $I = \hat{L} \times V$

Programming effort $E = \dfrac{V}{L} = \dfrac{n_1 N_2 N \log_2 n}{2n_2}$

Programming time $T = \dfrac{E}{S} = \dfrac{n_1 N_2 N \log_2 n}{2n_2 S}$ ($S$ is usually set to 18 seconds [45])

Some critique on these metrics include a scattered counting strategy among its users due to, among others, a vague definition of the difference between operators and operands. Also, some of the derived measures are mathematically equivalent which can cause confusion on their semantics. [43]

### Cyclomatic complexity

Presented by McCabe in 1976 [46], the cyclomatic complexity measure is a way to assess and measure the complexity of a code block. It is defined as

$$V(G) = e - n + 2p$$

Where $e$ is the number of edges, $n$ the number of nodes and $p$ the number of connected components. A software program can be presented as a *control graph* where each node represents a basic code block and each edge represents branches taken by predicates such as if-statements. It is assumed that the graph has only one entry point and one exit point and that each node can be reached from the entry node, and that each node can reach the exit node. If the exit node is connected to the entry node the graph is said to be *strongly connected*, and the cyclomatic complexity can be defined as the number of linearly independent circuits. [46]

In Figure 3.2 a example is provided. Node $A$ is the entry point and node $E$ is the exit point. Excluding the dotted edge connecting $E$ with $A$, the graph has 6 edges, 5 nodes and represents
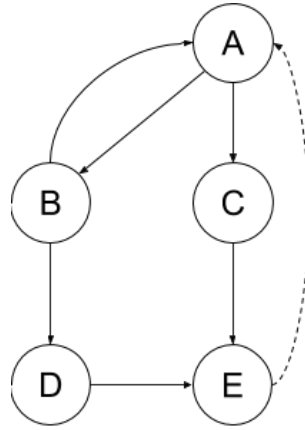
Figure 3.2: Control graph example

one connected component; i.e. $V(G) = 6 - 5 + 2 = 3$. Utilizing the connection between $E$ and $A$, the same complexity value can be calculated by counting the number of linearly independent circuits in the graph, i.e. the independent paths the program can take. In this example the paths are $ABA$, $ABDEA$, $ACEA$, which corresponds to the same result as $V(G)$.

Figure 3.2 can be seen as a single program or control flow, e.g. a function or class. Calculating complexity in a composition of functions or classes, or the complexity of a function that calls other functions is fortunately mathematically easy. The combined complexity of the graphs $G_1$ and $G_2$ is the sum of their respective complexities: $V(G_1 \cup G_2) = V(G_1) + V(G_2)$. [46]

The number of nodes and edges can be also be calculated by counting the number of functions, predicates and collecting nodes in the program [47]. McCabe applies those relationships to the cyclomatic complexity definition and reach the conclusion that the cyclomatic complexity equals the number of predicates in the program plus one [46]. So complexity can be calculated as

$$V(G) = \pi + 1$$

where $\pi$ is the number of predicates in the program.

**Weighted Methods per Class (WMC)**

This metric relates to object oriented designs. Let $n$ be the number of methods in a class and $c_i$ the complexity of method number $i$. Then the metric is defined as:

$$WMC = \sum_{i=1}^{n} c_i$$

The more the methods and the higher their complexity value is, the higher the maintainability effort is. [48]

**Depth of Inheritance Tree (DIT)**

In object oriented programming, a class can inherit another class, i.e. be a subclass of a superclass. This metric measures how many classes are inherited all the way up to the root class. In programming languages like C++ where classes can inherit multiple classes, the DIT is the longest distance to any root class. [48]

**Number of Children (NOC)**

This metric counts the number of classes that directly inherit a class, i.e. its children. An advantageous viewpoint of this metric is that the more children a class has, the greater the reuse of the class is. However, if a class has a large amount of children it has a great effect on the design of the system. [48]

**Coupling Between Object Classes (CBO)**

An object is *coupled* to another object of another class if it uses methods or instance variables of that class. Coupling relates to the modularity of a class, i.e. if a class has a high coupling value it is unlikely to be reused or modified without having an effect on the coupled objects. [48]

**Response for a Class (RFC)**

When an object receives a message, e.g. a method is invoked, a potential set of other methods can be called from that method. This metric measures the number of those methods. A large RFC value can predict testing and debuging effort of the class and its complexity. [48]

**Lack of Cohesion in Methods (LCOM)**

Cohesion relates to the similarity of methods in a class. It is undesireable to have a class design whose methods are not related to each other. This can be measured by looking at the instance variables of the class and see how the methods use them. If a class has $n$ methods, let $\{I_j\}$ be the set of instance variables used by class $M_i$. Let $P$ be the number of instance variable sets that has nothing in common, i.e. their intersection is $\varnothing$. Let $Q$ be the opposite, i.e. the number of instance variable sets whose intersection $\neq \varnothing$. Then $LCOM = |P| - |Q|$. [48]

$$P = \{(I_i, I_j)|I_i \cap I_j = \varnothing\}$$
$$Q = \{(I_i, I_j)|I_i \cap I_j \neq \varnothing\}$$
$$LCOM = |P| - |Q|, \text{ if } |P| > |Q|, 0 \text{ otherwise}$$

### 3.7.2 Assessing maintainability

Resulting numbers from a set of metrics applied on a software system can be difficult to interpret; is the system maintainable or not? An alternative to structural metrics can be subjective measures given by experts in the field. Anda [41] studied the correlation between structural measures and expert assessments on four different software systems with equivalent functionality and found that the both types of measures gave a similar result. Interesting results have been found in a study by Sjøberg et. al [49] where certain bad design choices (or *code smells*) was hypothesized to lead to a greater maintenance effort. However, the best predictor of maintenance effort turned out to be code size and number of revisions of the software, i.e. small code bases are easier to maintain. Benestad et. al. [40] applied several measurement strategies in a similar context and cross examined the result. They used class-level metrics, i.e. metrics applied on source code to make a system-level assessment. An overall assessment strategy can be divided into four categories; *selection*, *combination*, *aggregation* and *interpretation*, which are explained further in the following subchapters [40].

How can the confidence behind an assessment be determined? Benestad et. al. argue that if different assessment methods leads to the same result, one can be confident in the result. They call it *triangulation* and they use it in their study. [40]

More metrics like LOC, Maintainability Index and perhaps some machine learning approaches. Also metrics that does not infer object oriented languages.

**Selection**

It is important to select metrics that best fit the systems under study. The tools used to perform the metrics can play a big role in what metrics are used. Also specific set of metrics, like the CK metrics [48], are often used. Using statistical analysis, an initial set of metrics can be analysed to see if they discriminate between different designs. To avoid applying superfluous, or *unorthogonal*, metrics that does not influence the resulting maintainability value to a certain degree, *Principal Component Analysis* can be used. [40]

> Write about PCA!

**Combination**

As ideal metrics only should measure one aspect of the design, and as maintainability is affected by multiple aspects of the design, a combination of several metrics are required to provide a fair image of a system's maintainability value [40].

Oman and Hagemeister [7] proposes a quantitative approach where multiple metrics can be combined into a unified value. Software maintainability can be divided into three broad categories:

- The management practices being employed

- Hardware and software environments involved in the target software system

- The target software system

Oman and Hagemeister further derive the target software system into additional categories:

- Maturity attributes

- Source code

- Supporting Documentation

Maturity attributes include metrics such as the *age* of the software since release, *size* in terms of non-commented source statements and *reliability* which can be measured as the rate of failures per hour. Source code include metrics on how the software is decomposed into algorithms and how they are implemented, e.g. the number of modules and the cyclomatic complexity averaged over all modules. It also includes metrics on the information flow in the system, e.g. the number of global data types and structures, the number of data structures whose type is converted and the number of lines of code dedicated to I/O. Coding style metrics are also included, e.g. the percentage of uncrowded statements (only one statement per line), the number of blank lines and the number of commented lines are taken into account. Supporting documentation are evaluated in a subjective manner. Metrics include traceability to and from code implementation, verifiability of the implementation and consistency of the writing style and comprehensibility of the document. [7]

To assess these metrics, Oman and Hagemeister introduces a formula:

$$\prod_{i=1}^{m} W_{Di}\left(\frac{\sum_{j=1}^{n} W_{Aj}M_{Aj}}{n}\right)_i$$

where $W_{Di}$ is the weight of influence of software maintainability category $D_i$, e.g. source code control structure. $W_{Aj}$ is the weight of influence of maintainability attribute $A_j$, e.g. software age. $M_{Aj}$ is the measure of maintainability attribute $A_j$. The values should be structured so that they range from 0 to 1, thereby showing a percentage of their correctness. Worth noting is that not all attributes have to be measured, only those of significant value. [7]

Coleman et. al. [50] proposes a polynomial model that includes four metrics. By using regression analysis on historical data they were able to adjust the values in the polynomial to make a good maintainability prediction. The formula is expressed as:

$$\text{Maintainability} = 171$$
$$-5.2 \times \ln(aveVol)$$
$$-0.23 \times aveV(g')$$
$$-16.2 \times \ln(aveLOC)$$
$$+50 \times \sin(\sqrt{2.46 \times perCM})$$

Where $aveVol$ is Halstead's volume metric averaged over all source code modules, $aveV(g')$ is the extended cyclomatic complexity , $aveLOC$ is the average number of lines of code and $perCM$ is the percentage of comments across all modules. [50]

**Aggregation**

Aggregating results from software metrics is a statistical task. Metric values can be distributed in various ways, making fair interpretation hard. The most popular approach is taking the mean value of a specific metric, across all modules. A problem with this approach is, say one module has a very high complexity value while another has a very low, the resulting mean value will be a compensation right in between. A better approach in that case can be to measure the standard deviation of each metric that will give a hint on how evenly a certain metric is spread out across the modules, which leads to a notion of how balanced the design is. [40]

Table 3.1: Aggregation example found in an article by Anda [41]. The values can be read on the format $< mean > / < standard deviation >$.

|          | A         | B         | C         | D         |
|----------|-----------|-----------|-----------|-----------|
| **WMC**  | 6.9/11.2  | 7.8/10.3  | 11.4/12.5 | 4.9/4.5   |
| **NOC**  | 0.46/2.75 | 0.59/2.37 | 0/0       | 0.76/3.81 |
| **DIT**  | 0.46/0.5  | 0.75/0.81 | 0/0       | 0.83/0.54 |

Table 3.1 shows the result of some metrics applied on four systems A-D, each developed by different software teams but following the same specification. System C uses no inheritance functionality, hence the zero value for number of children (NOC) and depth of inheritance tree (DIT). All systems have a relatively high standard deviation, meaning the measures differ between classes and are quite unbalanced. [41]

**Interpretation**

Interpretation can be used as a final step in the maintainability assessment procedure in order to find a relationship between the internal, structural measures and the external quality attributes of the system. A level of uncertainty will always be present in interpretation and depending on the intended use of the measurement results, the appropiate uncertainty level changes. By simply looking at the resulting values from a combined or aggregated measurement, a maintainability assessment can be made. Low values can indicate bad maintainability and vice versa (or reverse). This is called *relative interpretation* and, though being simple, has its drawbacks. In object oriented languages, a deep class inheritance tree can be interpeted to be complex and result in bad maintainability. But in the opposite case, a flat class hierarchy means object oriented features are not used, which also can be interpreted as bad maintainability. Another interpretation strategy is the use of *threshold values* that are discrete intervals in

18

the possible value space of the structural measures. Thresholds can be labeled *Good*, *Acceptable*, *Suspicious* and *Unacceptable*. *Sensitivity analysis* may be needed when threshold values are used. Depending on the thresholds used the outcome can differ and therefore it can be valuable to try different threshold values. If the thresholds produce very sensitive outcomes the confidence of the maintainability result is low. [40]

### 3.7.3 Measuring maintainability in C++

Some metrics might not be suitable for object-oriented (OO) languages and C++ in particular. Wilde and Huitt [51] studies some of the main difficulties regarding maintenance as in "post deployment support" of OO-languages. These languages introduces the concepts of *object class inheritance hierarchy* and *polymorphism* which in some sense helps give programmers a better understanding of their programs, but the maintenance burden will unlikely disappear completely. One issue with analyzing OO source code is *dynamic binding*. An object variable might not necessarily refer to its declared class type, but it can also refer to any of its descendants in the class hierarchy. This makes static analysis complicated, e.g. it is not always known what implementation of a method will be used when the method is called on an object.

Rajaraman and Lyu [52] have also studied the shortcomings on some traditional maintainability metrics on OO-languages. For instance, they look on the metric *statement count* which can predict a software's complexity. It is built on the assumption that "the more detail that an entity possesses, the more difficult it is to understand" [52]. The metric simply counts the number of statements in a program or module. They criticize the metric for not taking the program's context into account and that it is not easy to determine what exactly a statement is. They also criticize McCabe's [46] *cyclomatic complexity measure* that calculates a program's complexity by taking the number of edges in a program flow graph, the number of nodes and the number of connected components into account (nodes are abstractions of sequential blocks of code and edges are conditional branches in the program). The critique involves, as stated earlier regarding statement count, that the metric ignores the context in the program and has no support to take the complexity of each statement into account.

Rajaraman and Lyu [52] defines four measures of *coupling* in their paper. They define the term coupling as "[...] a measure of association, whether by inheritance or otherwise, between classes in a software product". Abstracting the program to a directed multigraph, each node represents a class and each edge represents a reference from one node to another through variable references and method calls. Two of the proposed measures, *Class Coupling* and *Average Method Coupling* resulted in highest correlation (though not statistically significant) with perceived maintainability when tested on a number of C++ programs. Class Coupling for a class $C$ is defined as the sum of each outgoing edge from $C$; i.e. the sum of all global variable references, all global function uses, all calls to other class methods and all local references to other class instances. Average Method Coupling is a ratio number between a class $C$'s Class Coupling and its total number of methods, i.e. $AMC = CC/n$ where $n$ is the total number of methods declared in $C$.

## 3.8 Software development methodology

There exists a numerous amount of ways to develop software. In plan-based methodologies, including the *waterfall model*, the way of working is highly inspired by traditional engineering such as manufacturing and construction. Given a set of phases in the development, each phase must be done before the next phase starts. These phases include *requirement definition*, *design*, *implementation*, *testing* and *release* [53]. On the other end of the spectra lies the *agile* methodologies. Initially developed as a response to the frustration of the static, slow-going process of the well-used waterfall model, it is based on the understanding that software requirements are highly dynamic and most certainly change over time [54].

Vijayasarathy and Butler [55] found in an online survey they conducted in 2016 that around a third of all software projects were using the waterfall model as main software methodology. Following were the agile methodologies *Agile Unified Process* and *Scrum*. They also found that multiple methodologies were often used in the same project. For instance the agile method *Joint Application Development* was used in one project to identify requirements, while the waterfall model was used in the remainder of the project.

### 3.8.1   Scrum

Scrum is an agile framework utilized by small teams to develop simple and complex products. Scrum is not only used to do software development, it can be applied to any development - even a book [56]. In Scrum there are three main components: *roles*, *artifacts* and *the sprint cycle*. Roles are taken by people in the team, artifacts are tools used by the team and the fundamental pulse of the project is the sprint cycle. Three distinct roles are recognized: *product owner*, *Scrum master* and *team member*. The product owner is the team's representation of the *stakeholders* of the product (mainly the business related stakeholders). He is responsible for making the team always do most valuable work, he holds the vision of the product, he owns the *product backlog* and creates acceptance criteria for the backlog items. The scrum master is the team's coach guiding them to become a high-performing, self-organizing team. He helps the team apply and shape the agile practices to the team's advantage. A team member is responsible for completing *user stories*, create *estimates* and decides what tools to use in the upcoming *sprint*. [56]

The artifacts recognized by Scrum are called *product backlog*, *sprint backlog*, *burn down charts* and *task board*. The product backlog consists of a list deliverables that can be anything of value for the product, e.g. features and documentation changes. Items in the list are also called *user stories* and they are ordered by priority. They include information such as who the story is for, what needs to be built, how much work is needed to implement it and what the acceptance criteria is. Prior to a *sprint* user stories are derived into practical *tasks* usually performed by a single team member. The sprint backlog is populated by these tasks and they are expected to be finished within the time limit of the sprint. To monitor the status of the sprint or the entire project, the burn down chart is used. It is a diagram showing how much work is left to be done. The Y-axis shows the amount of tasks and the X-axis the time. As time progress the amount of tasks are hopefully decreasing. The task board is used to help the team inspect their current situation. It usually consists of three columns: *To do*, *Doing* and *Done*. Each column consists of tasks (or symbols of tasks) in the current sprint, and gives the team a visual on what tasks are yet to be done. [56]

A Scrum project is split into a series of sprints. They are time limited smaller versions of the entire project where a pre-defined amount of tasks are to be completed within the sprint. When a sprint is finished a potentially working product is demonstrated. The benefit of short-lived sprints is that the team receives frequent feedback on their work, giving them a better presumption to improve future sprints. From a business perspective, the sprint method provides greater freedom to decide if a product should be shipped or further developed. A sprint consists of a number of meetings: *sprint planning*, *daily Scrum*, *story time*, *sprint review* and *retrospective*. A sprint always starts with a sprint planning. It is a meeting where the product owner and the team decides which user stories will be a part of the sprint backlog in the upcoming sprint. This is also where the team derive the user stories into tasks. Every day the team also has brief daily Scrum meetings. Each team member shares what tasks they completed since the previous meeting, what tasks they expect to complete before the next and what difficulties they are currently facing. Every week, during story time, the product backlog is again reviewed by the team and the product owner. This time each user story is evaluated to refine its acceptance criteria. If there are large stories in the backlog they are also split into smaller stories to make them easier to understand and easier to complete within a sprint. The sprint review marks the public end of the sprint. Here the completed user stories are

demonstrated to the stakeholders and feedback is received. After this meeting the team has a retrospective meeting where they discuss what strategy changes can be made to improve the next sprint. [56]

# 4 Method

The aim of this study is to compare libuv to current state of the art techniques for applications in IoT. Previous attempts have been made to prove how certain programming languages perform better when used in a reactive context. Terber [57] discusses the lack of function-oriented software decomposition for reactive software. With an industrial application as context, he replaces legacy code with code written in the *Cèu* programming language, reaching the conclusion that Cèu preserves fundamental software engineering principles and is at the same time able to fullfill resource limitations in the system. Jagadeesan et al [58] performs a similar study but with a different language: *Esterel*. They reimplement a component in a telephone switching system and reach the conclusion that Esterel is better suited for analysis and verification for reactive systems.

## 4.1 Development methodology

This entire study will be conducted with Scrum as its backbone. Both the writing of the thesis and the development of the applications to test will happen in sprints. The project's backlog will initially be the fundamental requirements of a master's thesis (based on requirements from Linköping University) and the research questions. The forms of its user stories will resemble traditional issues or requirements, but their scope can be wide and their acceptance criteria abstract. Every week they undergo refinement and abstract stories will be split into concrete ones as new knowledge about them is acquired during the project. For instance, a starting user story (or issue) will be *"write the Results chapter"*. Initially this story is very large, it is hard to do it in one sprint and it is hard to know where to start. As time progress and the application to test have been developed and tests have been conducted, the story can be split into more precise issues like *"present the developed application"* and *"present a diagram with the test results"*. These issues are (subjectively) easier to do in one sprint and it is also easier to know when they are finished.

The author will take on all three traditional Scrum roles: product owner, Scrum master and team member. Other stakeholders of the project are representatives from Attentec, the examiner and a mentor from Linköping University. Each sprint will have a length of 2 weeks and the sprint planning will, unlike the traditional sprint planning [56] where only the product owner and team members are present, include a mentor from Attentec to help plan the

upcoming sprint. At the end of each sprint the current status of the project will be presented to the stakeholders.

## 4.2 Literature study

A major literature study will be conducted prior and during the implementation. The theoretical framework will be vindicated in this phase to support claims and form a general direction of the entire study. Multiple databases will be queried in order to find interesting material from previous research. Mainly the online library hosted by *Linköping University*[1] will be used since it allow access to material otherwise unviewable due to institutional login requirements. Query results from this library is a collection of query results from other research databases such as *ACM Digital Library*[2], *ProQuest Ebook Central*[3] and *IEEE Xplore Digital Library*[4]; so it acts as a gateway to a global collection of scientific research.

The *three-pass approach* presented by Keshav [59] will be used as a basic approach to find interesting material. It helps the reader grasp the paper's content in three *passes*. The first pass' purpose is to give the reader an overview of the paper. The title, abstract, introduction, headings, sub-headings, conclusions and references are read. This information should help the reader understand the paper's category and context and help decide whether to continue read this paper or leave it. If the reader choose to continue read it, the second pass starts. Here the paper is read more thoroughly. The figures and diagrams are examined and after this pass the reader should be able to summarize the paper, with leading evidence, to someone else. The purpose of the third pass is to fully understand the paper. By making the same assumptions as the author, the paper is virtually re-created. It helps identify the true innovations of the paper, as well as the hidden failures.

## 4.3 Implementation

The main approach this study will undertake to answer its question is to:

1. find a state of the art reactive system in the industry with an appropiate level of complexity

2. create a specification of the system

3. reimplement the system with libuv

4. create and conduct tests

5. apply and evaluate the maintainability metrics on both systems

6. present the results

With help from Attentec, a multi-sensor monitor application used by one of their clients will be specified and its source code will be used to compare its software maintainability to a reimplementation with libuv. If no appropiate system can be found in the industry, Attentec will aid in creating a specification for a similar system as well as a state of the art technique to implement it. The specification will follow the same pattern presented by Ardis et al [26].

Even though libuv is written in C, C++ will be used as the main programming language for the libuv implementation. With the dynamic programming style C++ offers with classes and templates, it will be more suitable for an implementation that hopefully will attract web developers.

---

[1] `www.bibl.liu.se`
[2] `dl.acm.org`
[3] `ebookcentral.proquest.com`
[4] `ieeexplore.ieee.org`

A testing environment will be created to simulate an IoT context where multiple sensors are connected to the monitor application. A software testing suite will also be setup to create unit tests.

## 4.4 Evaluation

Same software maintainability metrics will be applied on the system found in the industry and the reimplementation of it. The results of the metrics will be compared and presented.

# 5 Results

This chapter presents the results. Note that the results are presented factually, striving for objectivity as far as possible. The results shall not be analyzed, discussed or evaluated. This is left for the discussion chapter.

In case the method chapter has been divided into subheadings such as pre-study, implementation and evaluation, the result chapter should have the same sub-headings. This gives a clear structure and makes the chapter easier to write.

In case results are presented from a process (e.g. an implementation process), the main decisions made during the process must be clearly presented and justified. Normally, alternative attempts, etc, have already been described in the theory chapter, making it possible to refer to it as part of the justification.

# 6 Discussion

This chapter contains the following sub-headings.

## 6.1 Results

Are there anything in the results that stand out and need be analyzed and commented on? How do the results relate to the material covered in the theory chapter? What does the theory imply about the meaning of the results? For example, what does it mean that a certain system got a certain numeric value in a usability evaluation; how good or bad is it? Is there something in the results that is unexpected based on the literature review, or is everything as one would theoretically expect?

## 6.2 Method

This is where the applied method is discussed and criticized. Taking a self-critical stance to the method used is an important part of the scientific approach.

A study is rarely perfect. There are almost always things one could have done differently if the study could be repeated or with extra resources. Go through the most important limitations with your method and discuss potential consequences for the results. Connect back to the method theory presented in the theory chapter. Refer explicitly to relevant sources.

The discussion shall also demonstrate an awareness of methodological concepts such as replicability, reliability, and validity. The concept of replicability has already been discussed in the Method chapter (4). Reliability is a term for whether one can expect to get the same results if a study is repeated with the same method. A study with a high degree of reliability has a large probability of leading to similar results if repeated. The concept of validity is, somewhat simplified, concerned with whether a performed measurement actually measures what one thinks is being measured. A study with a high degree of validity thus has a high level of credibility. A discussion of these concepts must be transferred to the actual context of the study.

The method discussion shall also contain a paragraph of source criticism. This is where the authors' point of view on the use and selection of sources is described.

In certain contexts it may be the case that the most relevant information for the study is not to be found in scientific literature but rather with individual software developers and

open source projects. It must then be clearly stated that efforts have been made to gain access to this information, e.g. by direct communication with developers and/or through discussion forums, etc. Efforts must also be made to indicate the lack of relevant research literature. The precise manner of such investigations must be clearly specified in a method section. The paragraph on source criticism must critically discuss these approaches.

Usually however, there are always relevant related research. If not about the actual research questions, there is certainly important information about the domain under study.

## 6.3 The work in a wider context

There must be a section discussing ethical and societal aspects related to the work. This is important for the authors to demonstrate a professional maturity and also for achieving the education goals. If the work, for some reason, completely lacks a connection to ethical or societal aspects this must be explicitly stated and justified in the section Delimitations in the introduction chapter.

In the discussion chapter, one must explicitly refer to sources relevant to the discussion.

# 7 Conclusion

This chapter contains a summarization of the purpose and the research questions. To what extent has the aim been achieved, and what are the answers to the research questions?

The consequences for the target audience (and possibly for researchers and practitioners) must also be described. There should be a section on future work where ideas for continued work are described. If the conclusion chapter contains such a section, the ideas described therein must be concrete and well thought through.

# Bibliography

[1]    Hermann Kopetz. *Design Principles for Distributed Embedded Applications.* Springer, 2011.

[2]    Ericsson. *Internet of Things Forecast.* URL: `https://www.ericsson.com/en/mobility-report/internet-of-things-forecast` (visited on 11/14/2017).

[3]    Gordana Gardašević, Mladen Veletić, Nebojša Maletić, Dragan Vasiljević, Igor Radusi-nović, Slavica Tomović, and Milutin Radonjić. "The IoT architectural framework, design issues and application domains". In: *Wireless Personal Communications* 92.1 (2017), pp. 127–148.

[4]    T. Jamil. "RISC versus CISC". In: *IEEE Potentials* 14.3 (Aug. 1995), pp. 13–16. ISSN: 0278-6648. DOI: `10.1109/45.464688`.

[5]    Khaled Elmeleegy, Anupam Chanda, Alan L Cox, and Willy Zwaenepoel. "Lazy Asyn-chronous I/O for Event-Driven Servers." In: *USENIX Annual Technical Conference, General Track.* 2004, pp. 241–254.

[6]    The libuv team. *libuv.org.* URL: `http://libuv.org/` (visited on 11/17/2017).

[7]    Paul Oman and Jack Hagemeister. "Metrics for assessing a software system's maintain-ability". In: *Software Maintenance, 1992. Proceedings., Conference on.* IEEE. 1992, pp. 337–344.

[8]    The Linux Foundation. *Node.js 2016 User Survey Report.* URL: `https://nodejs.org/static/documents/2016-survey-report.pdf` (visited on 11/17/2017).

[9]    Pierre Bourque, Richard E Fairley, et al. *Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3.0.* IEEE Computer Society Press, 2014.

[10]   Kevin Ashton. "That 'internet of things' thing". In: *RFiD Journal* 22.7 (2011).

[11]   Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. "Internet of Things (IoT): A vision, architectural elements, and future directions". In: *Future generation computer systems* 29.7 (2013), pp. 1645–1660.

[12]   Luigi Atzori, Antonio Iera, and Giacomo Morabito. "The internet of things: A survey". In: *Computer networks* 54.15 (2010), pp. 2787–2805.

[13]   Dieter Uckelmann, Mark Harrison, and Florian Michahelles. "An architectural approach towards the future internet of things". In: *Architecting the internet of things.* Springer, 2011, pp. 1–24.

[14]   Roy Want. "An introduction to RFID technology". In: *IEEE pervasive computing* 5.1 (2006), pp. 25–33.

[15]   Ian F Akyildiz, Weilian Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. "Wireless sensor networks: a survey". In: *Computer networks* 38.4 (2002), pp. 393–422.

[16]   Alessandro Bassi and Geir Horn. "Internet of Things in 2020: A Roadmap for the Future". In: *European Commission: Information Society and Media* 22 (2008), pp. 97–114.

[17]   Nandakishore Kushalnagar, Gabriel Montenegro, and Christian Schumacher. *IPv6 over low-power wireless personal area networks (6LoWPANs): overview, assumptions, problem statement, and goals.* Tech. rep. 2007.

[18]   Qian Zhu, Ruicong Wang, Qi Chen, Yan Liu, and Weijun Qin. "Iot gateway: Bridging wireless sensor networks into internet of things". In: *Embedded and Ubiquitous Computing (EUC), 2010 IEEE/IFIP 8th International Conference on.* IEEE. 2010, pp. 347–352.

[19]   Hao Chen, Xueqin Jia, and Heng Li. "A brief introduction to IoT gateway". In: *Communication Technology and Application (ICCTA 2011), IET International Conference on.* IET. 2011, pp. 610–613.

[20]   Sang-Do Lee, Myung-Ki Shin, and Hyoung-Jun Kim. "EPC vs. IPv6 mapping mechanism". In: *Advanced Communication Technology, The 9th International Conference on.* Vol. 2. IEEE. 2007, pp. 1243–1245.

[21]   David Garlan and Mary Shaw. "An introduction to software architecture". In: *Advances in software engineering and knowledge engineering.* World Scientific, 1993, pp. 1–39.

[22]   James C Hu, Irfan Pyarali, and Douglas C Schmidt. "Measuring the impact of event dispatching and concurrency models on web server performance over high-speed networks". In: *Global Telecommunications Conference, 1997. GLOBECOM'97., IEEE.* Vol. 3. IEEE. 1997, pp. 1924–1931.

[23]   Atul Adya, Jon Howell, Marvin Theimer, William J Bolosky, and John R Douceur. "Cooperative Task Management Without Manual Stack Management." In: *USENIX Annual Technical Conference, General Track.* 2002, pp. 289–302.

[24]   Node.js contributors. *Node.js v9.4.0 Documentation.* URL: https://nodejs.org/api/ (visited on 01/31/2018).

[25]   David Harel and Amir Pnueli. *On the development of reactive systems.* Weizmann Institute of Science. Department of Applied Mathematics, 1985.

[26]   Mark A. Ardis, John A. Chaves, Lalita Jategaonkar Jagadeesan, Peter Mataga, Carlos Puchol, Mark G. Staskauskas, and James Von Olnhausen. "A framework for evaluating specification methods for reactive systems-experience report". In: *IEEE Transactions on Software Engineering* 22.6 (1996), pp. 378–389.

[27]   Benjamin Hummel and Judith Thyssen. "Behavioral specification of reactive systems using stream-based I/O tables". In: *Software Engineering and Formal Methods, 2009 Seventh IEEE International Conference on.* IEEE. 2009, pp. 137–146.

[28]   Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. "A survey on reactive programming". In: *ACM Computing Surveys (CSUR)* 45.4 (2013), p. 52.

[29]   Edward A Lee and David G Messerschmitt. "Synchronous data flow". In: *Proceedings of the IEEE* 75.9 (1987), pp. 1235–1245.

[30]   Leo A Meyerovich, Arjun Guha, Jacob Baskin, Gregory H Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. "Flapjax: a programming language for Ajax applications". In: *ACM SIGPLAN Notices.* Vol. 44. 10. ACM. 2009, pp. 1–20.

[31]  Mouaaz Nahas and Adi Maaita. "Choosing Appropriate Programming Language to Implement Software for Real-Time Resource-Constrained Embedded Systems". In: *Embedded Systems-Theory and Design Methodology*. InTech, 2012.

[32]  Michael Barr. *Programming embedded systems in C and C++.* " O'Reilly Media, Inc.", 1999.

[33]  libuv contributors. *libuv Documentation*. URL: http://docs.libuv.org (visited on 12/29/2017).

[34]  libuv contributors. *loop_iteration.png*. URL: https://github.com/libuv/libuv/blob/v1.x/docs/src/static/loop_iteration.png (visited on 12/29/2017).

[35]  Nikhil Marathe. *An Introduction to libuv*. URL: https://nikhilm.github.io/uvbook/ (visited on 12/29/2017).

[36]  Jane Radatz, Anne Geraci, and Freny Katki. "IEEE standard glossary of software engineering terminology". In: *IEEE Std* 610121990.121990 (1990), p. 3.

[37]  Mehwish Riaz, Emilia Mendes, and Ewan Tempero. "A systematic review of software maintainability prediction and metrics". In: *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*. IEEE Computer Society. 2009, pp. 367–377.

[38]  Tore Dybå. "Factors of software process improvement success in small and large organizations: an empirical study in the scandinavian context". In: *ACM SIGSOFT Software Engineering Notes*. Vol. 28. 5. ACM. 2003, pp. 148–157.

[39]  KK Aggarwal, Yogesh Singh, Arvinder Kaur, and Ruchika Malhotra. "Empirical Study of Object-Oriented Metrics." In: *Journal of Object Technology* 5.8 (2006), pp. 149–173.

[40]  Hans Christian Benestad, Bente Anda, and Erik Arisholm. "Assessing software product maintainability based on class-level structural measures". In: *International Conference on Product Focused Software Process Improvement*. Springer. 2006, pp. 94–111.

[41]  Bente Anda. "Assessing software system maintainability using structural measures and expert assessments". In: *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*. IEEE. 2007, pp. 204–213.

[42]  Maurice H Halstead. "Elements of software science". In: (1977).

[43]  Rafa E Al Qutaish and Alain Abran. "An analysis of the design and definitions of Halstead's metrics". In: *15th Int. Workshop on Software Measurement (IWSM'2005). Shaker-Verlag*. 2005, pp. 337–352.

[44]  Tim Menzies, Justin S Di Stefano, Mike Chapman, and Ken McGill. "Metrics that matter". In: *Software Engineering Workshop, 2002. Proceedings. 27th Annual NASA Goddard/IEEE*. IEEE. 2002, pp. 51–57.

[45]  Peter G Hamer and Gillian D Frewin. "MH Halstead's Software Science-a critical examination". In: *Proceedings of the 6th international conference on Software engineering*. IEEE Computer Society Press. 1982, pp. 197–206.

[46]  Thomas J McCabe. "A complexity measure". In: *IEEE Transactions on software Engineering* 4 (1976), pp. 308–320.

[47]  Harlan D Mills. "Mathematical foundations for structured programming". In: (1972).

[48]  Shyam R Chidamber and Chris F Kemerer. "A metrics suite for object oriented design". In: *IEEE Transactions on software engineering* 20.6 (1994), pp. 476–493.

[49]  Dag IK Sjøberg, Aiko Yamashita, Bente CD Anda, Audris Mockus, and Tore Dybå. "Quantifying the effect of code smells on maintenance effort". In: *IEEE Transactions on Software Engineering* 39.8 (2013), pp. 1144–1156.

[50] Don Coleman, Dan Ash, Bruce Lowther, and Paul Oman. "Using metrics to evaluate software system maintainability". In: *Computer* 27.8 (1994), pp. 44–49.

[51] Norman Wilde and Ross Huitt. "Maintenance support for object oriented programs". In: *Software Maintenance, 1991., Proceedings. Conference on.* IEEE. 1991, pp. 162–170.

[52] Chandrashekar Rajaraman and Michael R Lyu. "Reliability and maintainability related software coupling metrics in c++ programs". In: *Software Reliability Engineering, 1992. Proceedings., Third International Symposium on.* IEEE. 1992, pp. 303–311.

[53] Edward Crookshanks. *Practical Software Development Techniques: Tools and Techniques for Building Enterprise Software.* Apress, 2014.

[54] ABM Moniruzzaman and Dr Syed Akhter Hossain. "Comparative study on agile software development methodologies". In: *arXiv preprint arXiv:1307.3356* (2013).

[55] Leo R Vijayasarathy and Charles W Butler. "Choice of software development methodologies: Do organizational, project, and team characteristics matter?" In: *IEEE Software* 33.5 (2016), pp. 86–94.

[56] Chris Sims and Hillary Louise Johnson. *Scrum: A breathtakingly brief and agile introduction.* Dymax, 2012.

[57] Matthias Terber. "Function-Oriented Decomposition for Reactive Embedded Software". In: *Software Engineering and Advanced Applications (SEAA), 2017 43rd Euromicro Conference on.* IEEE. 2017, pp. 288–295.

[58] Lalita Jategaonkar Jagadeesan, Carlos Puchol, and James E Von Olnhausen. "A formal approach to reactive systems software: A telecommunications application in Esterel". In: *Formal Methods in System Design* 8.2 (1996), pp. 123–151.

[59] S Keshav. "How to read a paper". In: *ACM SIGCOMM Computer Communication Review* 37.3 (2007), pp. 83–84.