Linköping University | Department of Computer and Information Science
Master thesis, 30 ECTS | Datateknik
202017 | LIU-IDA/LITH-EX-A--2017/001--SE

# A study on the effects on performance and maintainability the libuv library has on low-cost ARM-based IoT-devices

–

Natanael Log

Supervisor : Min handledare
Examiner : Min examinator

LINKÖPING UNIVERSITY

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

## Upphovsrätt

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår. Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art. Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart. För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/.

## Copyright

The publishers will keep this document online on the Internet – or its possible replacement – for a period of 25 years starting from the date of publication barring exceptional circumstances. The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility. According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement. For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

© Natanael Log

## Abstract

Abstract.tex

# Acknowledgments

Acknowledgments.tex

# Contents

# List of Figures

# 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* 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.

## 1.2 Aim

To conclude whether libuv is a good option in terms of performance and 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

   a) software maintainability?

   b) performance?

   c) power consumption?

## 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 metrics, electronic hardware metrics, previous research on event-driven-related architectures and some general information regarding C++ and the *libuv* library.

## 3.1 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* [9]. 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 [10].

Vijayasarathy and Butler [11] 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.

### 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 [12]. 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*. [12]

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. [12]

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. [12]

## 3.2 Reactive systems

Systems required to continually respond to its environment are called *reactive* systems. Harel and Pnueli [13] 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. [13]

### Specifications for reactive systems

Harel and Pnueli [13] 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 particularily 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. [13]

Ardis et al [14] 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 [15] 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.

## 3.3 Embedded systems

### Choosing programming language

In embedded systems programming, both hardware and software is important. Nahas and Maaita [16] mentiones 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 [17] 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. [16]

## 3.4 Multithreading

Multithreading can be achieved both in software and in hardware. In software, multithreading is achieved by the operating system. Each process corresponds to a thread and each process can also spawn new threads [18].

The component maintaining all these threads is called the *scheduler* and its main purpose is to balance the load of threads between the CPU cores. The big problems the scheduler faces is 1) to make sure the most important thread is currently running and 2) to minimize the number of context switches, i.e. switching the current working thread. Linux uses a scheduling algorithm called *Completely Fair Scheduling* (CFS) which, in a single-core context, is quite simple. However, for multi-core systems the algorithm has proven to get very complex and several bugs has been found in it. [19]

## 3.5 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*. [20]

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: [20]

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.

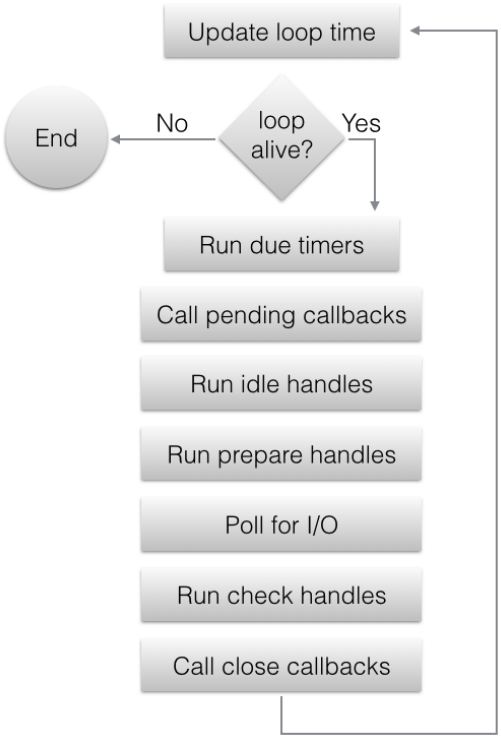



Figure 3.1: The stages each event loop iteration step through in libuv [21].

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 [22] 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 3.1: libuv example with an idle handle [22].

```
#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 [20].

## 3.6 Software 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 [23].

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 [24] 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 maintenability in mind. [24]

### Maintainability

Maintainability can be measured in many ways. Aggarwal et al. [23] 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. 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:

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

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.

### Measuring maintainability in C++

Some metrics might not be suitable for object-oriented (OO) languages and C++ in particular. Wilde and Huitt [25] 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 [26] 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" [26]. 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 [27] *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 [26] 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 $C$.

## 3.7 Hardware metrics

# 4 Method

The aim of this study is fundamentally to compare a single-threaded implementation to a multi-threaded implementation of an embedded software application. The application is of a reactive nature and is relevant to modern industry. Previous attempts have been made to prove how certain programming languages perform better when used in a reactive context. Terber [28] 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 [29] 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.

The main approach this study will undertake to answer its questions is 1) to create a specification of a system that will be implemented, 2) perform the implementation with the chosen libraries and platforms, 3) test the implementations, 4) apply the metrics and 5) perform evaluation of the metrics.

## 4.1 Pre-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 [30] 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

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

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.

## 4.2 Creating a specification

This very same method will be used in this study. A multi-sensor monitor application will be specified. Its basic purpose is to listen for data from different sensors and publish it to a web service. The application will be developed on a Raspberry Pi device with an ARM processor. Each sensor will be connected to the device's *GPIO* ports and the software will poll data in specific time intervals. The application will have a set of events which are called when a specific timeout or request is issued.

- $DATA\_READY$: This event is dispatched when sensor data is ready to be read. The data is a tuple consisting of the sensor ID and its current value: $< ID, value >$.
- $SYSTEM\_STATUS$: This event is dispatched from a remote client on the web.

Following is a description of the general properties of the system.

1. When event $DATA\_READY$ occurs,
   a) If a remote client exists, send the data to it.
   b) Do nothing.
2. When event $SYSTEM\_STATUS$ occurs,
   a) If the last time $DATA\_READY$ was dispatched is outside the accepted time frame, send $STATUS\_NOT\_OK$ to the remote client.
   b) Send $STATUS\_OK$ to the remote client.

## 4.3 Implementation

Two different implementations written in C++ will be tested in this study: one using the *libuv* library, and one with no specific third party library. The first implementation will utilize the *event-loop* inside the libuv library to handle timers and event callbacks. The second implementation will instead create new threads to handle events.

## 4.4 Testing the implementation

A testing environment will be created. This will include actual hardware sensors for humidity and temperature that will be connected to the Raspberry Pi device. A web client will also be developed to be able to communicate with the device to receive sensor data and request system status. Multiple test cases will be written in both software and as a specification for higher level integration tests.

## 4.5 Applying the metrics

Maintainability will be the main software metric to apply on the implementations. To measure performance, a benchmark test will be done were a large amount of sensor data polls and remote client requests will be issued.

## 4.6 Evaluate the metrics

The values received from both the software metric analysis and the performance analysis will be compared.

# 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

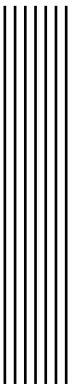
There must be a section discussing ethical and societal aspects related to the work. This is important for the authors to demonstrate a professional maturity and also for achieving the education goals. If the work, for some reason, completely lacks a connection to ethical or societal aspects this must 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 Radusinović, 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 Asynchronous 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 maintainability". In: *Software Maintenance, 1992. Proceerdings., 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] Edward Crookshanks. *Practical Software Development Techniques: Tools and Techniques for Building Enterprise Software*. Apress, 2014.

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

[11] 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.

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

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

[14] 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.

[15] 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.

[16] 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.

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

[18] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating system concepts essentials*. John Wiley & Sons, Inc., 2014.

[19] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. "The Linux scheduler: a decade of wasted cores". In: *Proceedings of the Eleventh European Conference on Computer Systems*. ACM. 2016, p. 1.

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

[21] 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).

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

[23] 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.

[24] 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.

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

[26] 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.

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

[28] 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.

[29] 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.

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