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

Abstract

`Abstract.tex`

Acknowledgments

`Acknowledgments.tex`

Contents

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. How does an event-driven architecture using libuv compare to a multi-threaded environment in terms of performance?

2. How does an event-driven architecture using libuv compare to a multi-threaded environment in terms of 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 study was conducted in the offices of the IT consulting firm Attentec in Linköping, Sweden. With offices in three cities: Linköping, Stockholm and Oslo, Attentec focuses 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. A personal interest in embedded Linux, NodeJS and event-driven architectures (as well as an interest in the company) led me to Attentec and a proposal of this study.

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 Reactive systems

Systems required to continually respond to its environment are called *reactive* systems. Harel and Pnueli [9] elaborates on different system dichotomies, for instance deterministic and nondeterministic 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 the 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. [9]

3.2 Programming embedded systems

In embedded systems programming, both hardware and software is important. Nahas and Maaita [10] 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 [11] 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. [10]

3.3 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 [12].

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. [13]

3.4 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 [14].

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 [15] 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. [15]

Maintainability

Maintainability can be measured in many ways. Aggarwal et al. [14] 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:

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

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.

Measuring maintainability in C++

Some metrics might not be suitable for object-oriented (OO) languages and C++ in particular. Wilde and Huitt [16] 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 [17] 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" [17]. 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 [18] *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 [17] 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.

3.5 Hardware performance 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 [19] 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 [20] 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 Creating a specification

Harel and Pnueli [9] 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. [9]

Ardis et al [21] 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.

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.

4.2 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.

4.3 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.

4.4 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.

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

[10] 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.

[11] Michael Barr. *Programming embedded systems in C and C++.* ” O’Reilly Media, Inc.”, 1999.

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

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.