Linköping University | Department of Computer and Information Science
Master thesis, 30 ECTS | Software Engineering
2018 | LIU-IDA/LITH-EX-A--18/013--SE

A Study on the Performance and Architectural Characteristics of an Internet of Things Gateway

En studie om prestanda och arkitekturer hos Internet of Things gateways

Natanael Log

Supervisor : Petru Eles
Examiner : Petru Eles

External supervisor : Pär Wieslander

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

Abstract

This study focuses on the Internet of Things (IoT) gateway; a common middleware solution that bridges the gap between physical sensors and devices to internet applications. There is a shown interest in understanding the characteristics of different types of gateway architectures both from the research field and the industry, particularly the IT-consulting firm Attentec in Linköping, Sweden. A study has also been made on the open source C library *libuv*, used in the common web runtime engine *NodeJS*. The library has been used to study how asynchronous I/O operations can be used to improve the IoT gateway performance. A set of three general architectural approaches are identified. Common internal and external properties are identified based on state-of-the-art gateway implementations found in the industry. All of these properties are taken into account when a general gateway implementation is developed that is proposed to mimic any architectural level implementation of the gateway. A set of performance tests are conducted on the implementation to observe how different configurations of the gateway affect throughput and response time of data transmitted from simulated devices. The results show that the properties of the gateway do affect throughput and response time significantly and that libuv overall helps implement one of the best performing gateway configurations.

Acknowledgments

I want to give special thanks to my external supervisor Pär for your ideas, your time and support. Thank you Anders and Attentec for accepting my original idea, letting me conduct this thesis in your office and not to forget the incredible trip to New York. Thank you Eric and Jarle for giving me fast, continuous and informative feedback on my work. Thank you Petru for accepting this idea as a thesis and for your fast email responses. Last but not least: a great thank you to the love of my life, Sara. For you support, your joy and your love.

Contents

List of Figures

List of Tables

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] [2]. Ericsson foretold there would be around 400 million smart devices by the end of 2016 [3] 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” [4]. The complexity of a smart device varies from passive ID-tags which communicate through near field communication to full scale computers with multithreaded operating systems.

An important component in the IoT architecture is the IoT gateway [5]. It acts as a bridge between the ”things”, or devices, and the internet. Gateways can be seen as *event-driven*, or *reactive* systems, i.e. systems required to respond to stimuli in their environment [6] [7]. Specifying reactive systems can be done by designing them as *state machines* and implementation can be done in *reactive languages* [8]. These languages have abstracted away time management, in the same sense memory management is abstracted away in, e.g. Java, by using garbage collectors.

This thesis has been conducted at the Linköping-based company Attentec in Sweden. They focus on IoT-development and the gateways used by some of their customers are implemented on embedded, resource-constrained hardware. A common language for embedded systems is C as it allows the developer to access hardware without losing the benefits of high-level programming [9] [10]. The C library *libuv* can be a good choice when implementing reactive systems in C [11]. 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. This work examines the performance characteristics of the IoT gateway when its internal and external properties are varied and verifies if libuv is a good choice of approach when implementing an IoT gateway in C.

1.1 Motivation

Earlier work has attempted to quantify the performance characteristics of gateways, but most focus has been on either higher level architectural models [5] [12] or hardware performance characteristics [13]. Observing the performance while varying software-related properties of the gateway will therefore be a good contribution to the research-field. Since IoT and gateway technology is growing and as the demand for its services will largely increase [2], efficiency and performance is an important factor. There exist 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 applies to 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 rigorous evaluation of the different architectures a gateway can be developed on can therefore be of great value to find what suits best in IoT.

NodeJS, a popular universal runtime platform for *"front end, back end and connected devices [and] everything from the browser to your toaster"*, includes the major subsystem libuv [14]. Due to NodeJS' event-driven development style many developers in its field might have a better experience using libuv if transfer to embedded programming could be in question.

1.2 Aim

The aim of this thesis is to understand and map the internal and external properties of the gateway to its functionality and performance and to assess the legitimacy of using libuv as an implementation approach when developing a new IoT gateway.

1.3 Research questions

1. What internal and external properties affect the functionality and performance of the IoT gateway?

2. How can libuv be used in order to implement an IoT gateway? What are the benefits and disadvantages of doing so?

1.4 Delimitations

Performance will be the only attribute used to determine whether a gateway implementation is "good" or not. There are other ways of measuring programs as well, one of them are code measurements such as maintainability. However, this study only focuses on performance measures.

The context of the study is IoT gateways. This study can however be applicable on a larger spectrum of environments, e.g. web servers or enterprise systems.

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

The theory framework is presented here. It has been developed mainly in the initial phase of the study. Topics are presented here that are helpful for the reader to get a grasp on the background of the methods and techniques used in the study.

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 [15] and the popularity has since blossomed [16]. 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 [17]. IoT can typically be explained as "one paradigm with many visions" [17] 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"* [18]. 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."* [18].

3.1.1 Enabling technologies and challenges

The state-of-the-art technologies used today in the IoT context include sensor 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*.

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 [17]. 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. [19]

In a survey from 2001, Akyildiz et. al. [20] present a *Wireless Sensor Network* containing hardware nodes capable of monitoring their 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 to 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"* [21], 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 [17]. 4 bytes are used to address devices in IPv4, which means the protocol can support around 4 billion devices. IPv6 on the other hand uses 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. [22]

3.1.2 Gateways

A big challenge in IoT is, as described earlier, to connect every single device to the internet. For example, when 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 [23]. Chen et. al. [5] 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 happens; 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). For instance, 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 [24]. The gateways reside in the network domain. They act as a bridge between the devices and the internet.

3.2 Event-driven architectures

In software architecture, the terms *event-driven*, or *implicit invocation*, are used to express the mechanism of function invocation based on events [6]. In contrast, *explicit invocation* refers to a traditional function invocation by function name, e.g. `foo()`. However, with the event-driven approach, 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 [6]. However, in high-performance, I/O intensive networking systems, it is shown that event-based approaches perform much better than other architectures [25].

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 parallel on multicore systems. In *non-preemptive* (or *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 their 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 implement the cooperative approach. Events, e.g. network requests, are handled by calling associated functions. 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. [26]

There are two categories of I/O management: *synchronous* and *asynchronous* [26]. 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 NodeJS documentation [27]:

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 preemptive 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 infers 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 [26]. 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 elaborates on different system dichotomies, for instance deterministic and non-deterministic systems. Deterministic systems have unique actions that always generates the same output for a given input. 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 output, 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. [7]

3.3.1 Specifications for reactive systems

Harel and Pnueli 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 numerous possible input combinations, a large set of states must be created which in the long run is not appropriate 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 undergoes a transition to another state when an action is performed on that state. [7]

Ardis et al. [28] 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 fulfilling the requirements in a set of different languages. Most languages fit well for reactive systems since their semantics 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 [29] 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 [8]. It is derived from the *Synchronous Data Flow* (SDF) paradigm [30]. SDF is a data flow abstraction used to describe parallel computation. Functions are presented as nodes in a directed graph and the arcs represent data paths. Nodes can execute their computation whenever data is available on their 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 mutate any resource accessible by another node unless they are explicitly connected by an arc.

In the reactive programming paradigm, the notion of time management is abstracted away. In the same way memory management 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* [8]. 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. [8]

Reactive programming has two distinguishing features: *behaviors* and *events*. Behaviors refer to time-varying values, 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. [8]

3.4.1 A taxonomy

Bainomugisha et al. [8] present a taxonomy with six fundamental properties considered important features in reactive languages.

Basic abstractions

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

Evaluation model

When an event occurs, the evaluation model of a reactive language determines how change is propagated throughout the data dependencies. Propagation 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 with this model is to find out what nodes do 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 kind of inconsistencies, or glitches, should be avoided by the language.

Lifting operators

Lifting in reactive programming refers to the act of making previously incompliant 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, a reactive language built upon Javascript, lifting is performed on multiple places to support behaviors [31]. 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 [8].

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 as 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 [9] mention 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 appropriate 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 [10] states that the key advantage of C is that it allows the developer to access hardware without losing the benefits of high-level programming. Compared to C, C++ offers a better object-oriented programming style but can on the other hand be less efficient. [9]

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. The central part of libuv 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 it instantly returns and abstracts away any concurrency necessary. To handle network I/O libuv makes 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

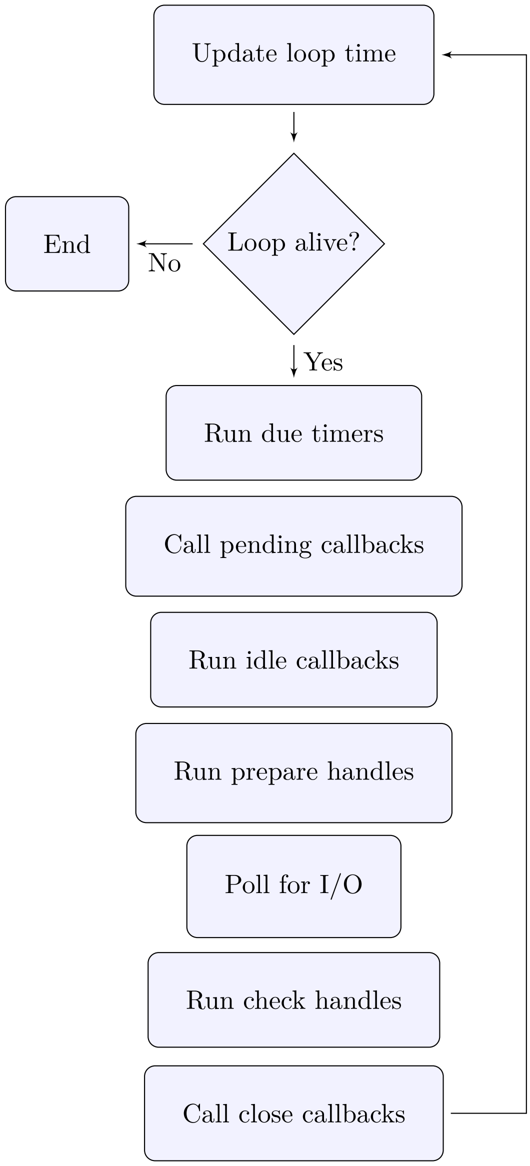

Figure 3.1: The stages each event loop iteration steps through in libuv.

I/O. libuv includes handles for *UDP* and *TCP* sockets, file and file system, *TTY*, timers and *child processes*. [32]

Figure 3.1 demonstrates the stages each event loop iteration steps 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 active handles the loop dies. The steps are described briefly: [32]

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 that 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 continues.

The following example provided by Marathe [33] 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 [33].

```
#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 [32].

3.7 Software testing

Testing software is a broad concept with many goals such as validating an implementation to conform to its specification, evaluating robustness and performance, or finding inconsistencies in the design [34]. It is a generally hierarchical process where testing of discrete modules (*unit testing*) to testing of the interaction between modules (*integration testing*) to testing of the composition of modules and the entire system (*system testing*) is done [35]. The specification of the *system under test* (SUT) can be unknown, thus only the inputs and the outputs of the system can be examined for test case generation (*black-box testing*) [36]. The codebase of the SUT can also be known, enabling verification of test case coverage so all paths in the code are tested (*white-box testing*) [36].

Software testing includes basic terms such as *error*, *fault*, *failure*, *incident*, *test* and *test case* [37]. People make errors and faults are the results of the errors. A good synonym for both is *bug*. When a program executes a fault a failure occurs. But the failure could have happened not only because bad code was executed (*faults of commission*) - it could also be due to the fact that the correct code was absent (*faults of omission*). An incident is the consequence of a failure that signals the user of its presence. Fundamentally the act of testing has two objectives: to locate failures and to verify correct execution of the program. The actual procedure that executes a test is the test case and it has a set of inputs and a set of expected outputs. [36]

Important to ask oneself when tests are being planned or about to be carried out is [34]:

- What is the *test objective*? To locate faults in the design or verify correct behavior?
- What test methods should be *selected*?
- What is an *adequate* test sample? What are the criteria to stop testing?
- What *levels* of testing should be conducted? Unit testing on separate modules or the entire system?
- In what environment should the tests be conducted?
- When should the tests be conducted? In the end of the software lifecycle in a waterfall fashion, or continuously between sprints in an agile development framework?

3.7.1 Performance testing

Performance testing is the process of measuring and evaluating certain performance related benchmarks in a system; usually by putting the system under I/O load such as simulated network traffic [38]. Software systems contain resources such as the CPU, processes, threads and I/O that all have limited capacities, which means they potentially can deny each others execution when several users require the same resource. Analyzing a system's performance requires quantification of such effects [39]. The goal of the tests could be to compare an existing system with a new proposed system to see which one performs better [40] or to, in an early phase of the development of a system, identify future performance problems upon deliverance [41]. Contrary to functional software testing where the values of the test inputs determine the test outcome, performance testing emphasizes the *workloads* and *frequencies* of the inputs [41]. Jiang and Hassan discuss performance testing along with two other similar testing techniques: *load testing* and *stress testing* [38]. Load testing is defined as the process of assessing a system's behavior under load to find both functional and non-functional related issues. Stress testing is about putting the system under extreme load conditions to verify robustness. Performance testing differs from the other two when it comes to verification of benchmarks. A performance test can for instance happen when a module has been reengineered and the performance must be verified to at least be as good as the previous version of the module [38].

Usually, performance tests objectives relate to I/O throughput and response time. Weyuker and Vokolos conducted performance tests on a telecommunication gateway by developing simulated telephone callers [41]. These callers performed calls and transactions on the systems that were measured to analyze performance. Performance metrics used by the popular performance testing tool *JMeter*[1], which is often used to test web applications and services, include [42]:

Number of Users: Measuring the number of active users can be done by creating *virtual users* in a testing program. This metric is at most interesting when analyzed together with Hits per Second.

Hits per Second: This is a measure on how many requests are generated by each user. The correlation between Number of Users and Hits per Second provides a valuable hint on the performance of the server.

Errors per Second: A high error rate when the user amount increases indicates a performance issue.

Response Time: The time it takes from a request to its response is, especially when multiple requests are fired concurrently, a good indication on the performance of the system.

Throughput: This measures the average bandwidth usage generated by the tests.

To conduct a performance test, not only are the metrics necessary, but a representative workload is needed as well [41]. Methods to design workloads often require historical data on what load the system undergoes in typical usage. But as performance testing usually is performed on novel systems, historical data is a rare commodity [43]. Alternative approaches can be to retrieve usage data from a beta version of the system, have interviews with domain experts or analyze usage data from competitors [38]. Barber synthesizes common approaches found in the industry and provides a, somewhat agile, framework to support design of a workload model where historical data is incomplete or unavailable [43]:

1. Identify existing historical data, preferably found in log files from production instances of the system.
2. If log files are unavailable, try to extract information from stakeholders such as salesmen (they might know of the most anticipated activities that can be used in the workload design) and users (they might know what activities they probably will do).
3. Look for performance related activities in design and requirements documents.
4. Whenever a new beta version of the system is available, retrieve workload data from it.
5. Look for workload data in competitors' statistics if available.
6. Conduct in-house experiments.
7. If no data is available, use personal experience as a substitute to design an appropiate model.
8. Create visual representations of the model for peer review.
9. Create three different workload models based on:
    a) Anticipated or expected usage.
    b) Best case usage.

[1] See `jmeter.apache.org`.

c) Worst case usage.

Jiang and Hassan suggest that workload models can either reflect the system's performance requirements (so called *aggregate-workload based load design*) or the system's expected usage (*use-case based load design*). An example of a performance requirement can be that CPU usage should not exceed 90 % when 10,000 transactions are concurrently handled. Expected usage can be, for a web shop application, that users typically spend 40 % of their time browsing and 60 % in the checkout. In the aggregate-workload approach loads are characterized as the *workload intensity*, i.e. the rate in which the requests fire, and the *workload mix*, which describes the different kinds of requests in the workload, e.g. 20 % of the requests come from resource A and 80 % from resource B. For cases where historical data or an operational profile is available, a workload model can be designed as *steady load* or *step-wise load*. In a steady load, only one intensity and mix configuration is used, normally to represent expected usage of the system. However, a better representation of a realistic workload would be one where the workload varies with time. A step-wise load include multiple levels of workload intensity (but keeps the workload mix) to mimic certain usage levels across a time period. The main disadvantage with the aggregate-workload approach is that loads can be unrealistic or hard to design. The use-case approach tries to solve this by introducing states and stochastic aspects into the design. One can for instance design loads using *use-case diagrams*, *Markov chains* and *Probabilistic Timed Automata* to create usage driven designs that resemble real-world workloads. [38]

3.8 Software development methodology

There exist 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* [44]. 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 [45].

Vijayasarathy and Butler [46] 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 both simple and complex products. Scrum is not only used to do software development, it can be applied to any development - even writing a book [47]. 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*. [47]

The artifacts recognized by Scrum are called *product backlog*, *sprint backlog*, *burn down charts* and *task board*. The product backlog consists of a list of 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 progresses 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 picture of what tasks are yet to be done. [47]

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 derives 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. [47]

4 Method

The goal of this study is to map out and understand how IoT gateway architectures can be developed and how they perform. Also, the asynchronous I/O library libuv will be studied to see how it can increase the performance of IoT gateways.

4.1 Research and development methodologies

The study was conducted with Scrum as its backbone. Both the writing of the thesis and the development of the applications to test happened in two-week sprints. The project's backlog was initially the fundamental requirements of a master's thesis (based on requirements from Linköping University) and the research questions. The forms of its user stories resembled traditional issues or requirements, but their scopes were wide and their acceptance criteria abstract. Every sprint they underwent refinement and abstract stories were split into concrete ones as new knowledge about them were acquired during the project. For instance, a starting user story (or issue) was *"write the Results chapter"*. Initially this story was very large, it was hard to do it in one sprint and it was hard to know where to start. As time progressed and the implementation to test had been developed and empirical data was gathered, the story was split into more precise issues like *"present the developed application"* and *"present a diagram with the test results"*. These issues were easier to do in one sprint.

The overall phases of the study were a design phase, a technological phase and a reflexive phase [48]. In the design phase, the concept and goals of the study were developed. Together with the author's own ideas and input from Attentec, the value of understanding IoT gateway performance was identified. The hypothesis that the libuv library will improve IoT gateway performance was stated and a theoretical framework was mapped out as previous research was studied.

The theoretical framework was developed in this phase to support claims and form a general direction of the entire study. Multiple databases were queried in order to find interesting material from previous research. Mainly the online library hosted by *Linköping University*[1] was used since it allows access to material otherwise non-viewable due to institutional login requirements. Query results from this library is a collection of query results from other research

[1] `www.bibl.liu.se`

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 [49] was 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.

Together with Attentec, a model of a general, or abstract, gateway was theoretically developed. The idea was that all IoT gateway implementations have some common features, constraints and environments that can be simulated and tested. Another hypothesis developed that different IoT gateway architectures performed differently in different environments, and that there is no ”one-size-fits-all” architecture.

The next phase, the technological phase, started with implementation of the abstract gateway in order to enable empirical data collection of different performance measures of different configurations of the abstract gateway. Once the implementation was considered done, the data collection began.

In the reflexive phase the empirical data was interpreted. There were efforts in filtering the data and prioritizing what was the most interesting and significant data that would lead to the best result. The results were then formulated in this thesis.

4.2 Related work

Previous attempts have been made to prove how certain programming languages perform better when used in a reactive context. Terber [50] 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 fulfill resource limitations in the system. Jagadeesan et al. [51] perform 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.

No previous research has been found regarding the internal architecture of the IoT gateway and how it affects performance. Most research regarding IoT gateways discuss architecture on an application level. Performance testing is also to a certain degree a non-explored area and most of the literature takes on empirical approaches, especially for distributed systems performance.

Chen et al. [5] describes the IoT gateway as the bridge between the sensing domain in the IoT architecture and the network domain and argues its importance as one of the most significant in the IoT architecture. They describe some common features of the gateway such as support for multiple network interfaces (2G/3G, LTE, LAN), protocol conversion and manageability. A reference model is presented where these features are taken into account. However, the paper is not focusing on the same architectural abstraction level as this thesis, but provides a more holistic view on the entire IoT gateway application and sub-applications.

[2] `dl.acm.org`
[3] `ebookcentral.proquest.com`
[4] `ieeexplore.ieee.org`

Zachariah et al. [12] argue that application specific IoT gateways are not the best approach towards a scalable IoT infrastructure. The same reason there isn't one web browser per web page, gateways should be generic and support all types of IoT devices and sensors. They propose a smartphone-centric architecture where IoT devices connect to the internet via smartphones. This paper is valuable because it challenges the traditional view of IoT gateways. It is however elaborating the gateway on a higher abstraction level than this thesis is.

Weyuker et al. [41] states that there has been very little research on software performance testing in general. In their paper, they discuss performance testing objectives, workload design and the role of requirements and specifications to eventually lead up to a case study where they test a gateway system without having access to historical usage data. They develop test cases based on usage scenarios to adhere their formulated performance objectives. The tests are conducted by running programs that simulate clients in the system, which send transaction requests in various sizes and frequencies. The rate of transactions are monitored and used to assess the performance of the entire system. Their reflections and discussions on performance testing at large are very valuable to this thesis. However, their paper focuses on an already existing application upon which performance testing will be used to assess whether it fulfils its functional and non-functional requirements. This thesis dives into how an abstract gateway can be modeled and implemented and how its different configurations affect performance for some given contexts.

Denaro et al. [52] present a systematic method to ascertain that a given distributed software architecture lives up to its performance requirements. They mention some performance metrics such as latency, throughput and scalability. They state that the overall performance of a distributed system is often determined by the middleware used, and that the same middleware can affect performance differently depending on its context of application. This is a good incentive to map and understand IoT gateway performance, which is done in this thesis.

Kruger et al. [13] discuss the common use of off-the-shelf components for implementing IoT devices and gateways. They state that little work has previously been done to explore the performance of these devices when different parameters are adjusted, such as system architecture and software components. They focus primarily on hardware adjustments (e.g. processor and memory speed) to identify performance bottlenecks in three common off-the-shelf brands: Raspberry Pi, Beaglebone and Beaglebone Black. They use *micro-benchmarks* to identify performance bottlenecks at architecture level and *macro-benchmarks* at application level. They reach the conclusion that gateway performance is most important when there is a large amount of data being transferred between the gateway and the IoT application.

Aguilera et al. [53] present an approach to identify performance bottlenecks in distributed systems where the components are seen as black boxes, i.e. sub-systems of unknown functionality. By monitoring the traffic of *remote procedure calls* in the system and store them in a log they are able to find the *causal paths* of messages and thereby understand the data flow in the system and where performance bottlenecks are located. The approach to capture log messages is used in this thesis, however the step of storing messages in a log file before computation is omitted and each log message is computed as it arrives.

Liu et al. [54] present an approach to predict distributed system performance empirically by running test suites. They map out some parameters that affect performance, such as client request load and thread pool size and some observable parameters such as throughput and response time. They reach the conclusion that gathering empirical results with simple test cases is an effective method to understand the performance profile of the system.

4.3 Two customer cases

Before presenting the architectural models used in this study, it is of importance to understand how they came about. Among the customers of Attentec, two were of interest for this study with each having implemented an IoT gateway suitable for its operating context. *Customer A* uses their IoT gateway to monitor industrial batteries used in forklifts. The main idea behind the event propagation model is that the gateway polls each battery for events and process them according to its specification (which is not relevant here). Referring to data propagation in reactive languages (see section 3.4.1), one can view the event propagation as *pull-based*, i.e. the gateway pulls events from the batteries whenever it finds it necessary. The batteries are *passive*, i.e. they do not emit any events on their own, only when asked to by the gateway. Figure 4.1 illustrates this.

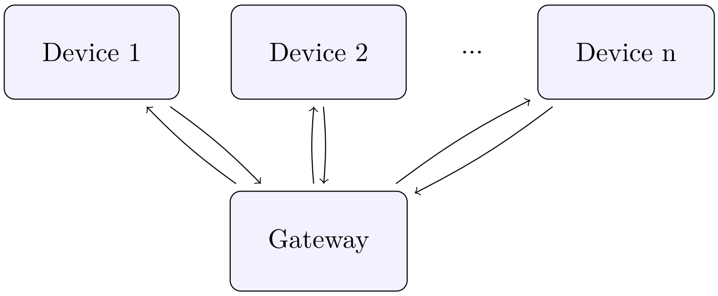

Figure 4.1: Pull-based event propagation with passive devices. The gateway pulls events from the devices by requesting them first.

Customer B, on the other hand uses a *push-based* approach to handle event propagation in their gateway. The gateway hosts a REST API to serve IoT devices. This means that the devices must be *active*, i.e. they emit events whenever they find it necessary. See Figure 4.2 for an illustration. Note that not as many communication requests are necessary in this approach compared to the pull-based one. The gateway does not need to know about how many devices there are or what their addresses are, but the devices need to know the address of the gateway. In contrast, the pull-based approach requires the gateway to know about each device, but the device does not need to know anything about the gateway.

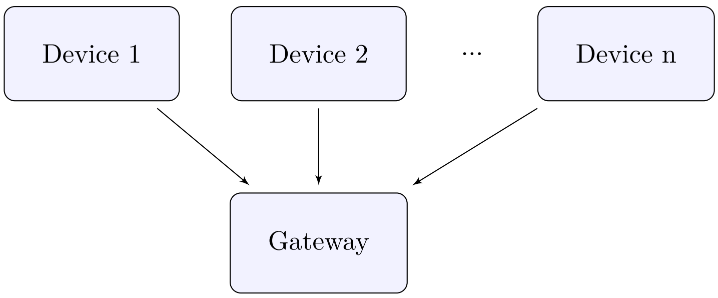

Figure 4.2: Push-based event propagation with active devices. Events are pushed from the devices to the gateway that listens for incoming requests.

4.4 Three event propagation models

Taking inspiration from the three task management approaches presented by Adya et al. [26] and described in Section 3.2.1, three event propagation models are proposed: *serial*, *preemptive* and *cooperative event propagation*. Before presenting a formal definition, an analogy might be well-suited here. Imagine a cafe that serves coffee. There are several customers in line and the barista takes the order from the next customer in line, brews the coffee and serves it. The coffee brewer can only brew one cup of coffee at a time. This is repeated for each customer in line with the constraint that no new order is accepted before the current order is served, see Figure 4.3. This can be seen as a *serial order handling*; only one order can be handled at a time. Imagine now that the cafe hires more baristas and more coffee machines. There is still one line (however, there can be several), but orders can be handled concurrently, so the next customer in line does not necessarily have to wait for the previous customer to be served before he can place his order, see Figure 4.4. This can be seen as *preemptive order handling*. Now lets keep the brewers but let go all baristas except for one. All customers place their order to the same barista which in turn start one of the brewers. While the brewer is working, the next customer in line places his order and the barista can start a second brewer, see Figure 4.5. This is called *cooperative order handling*.

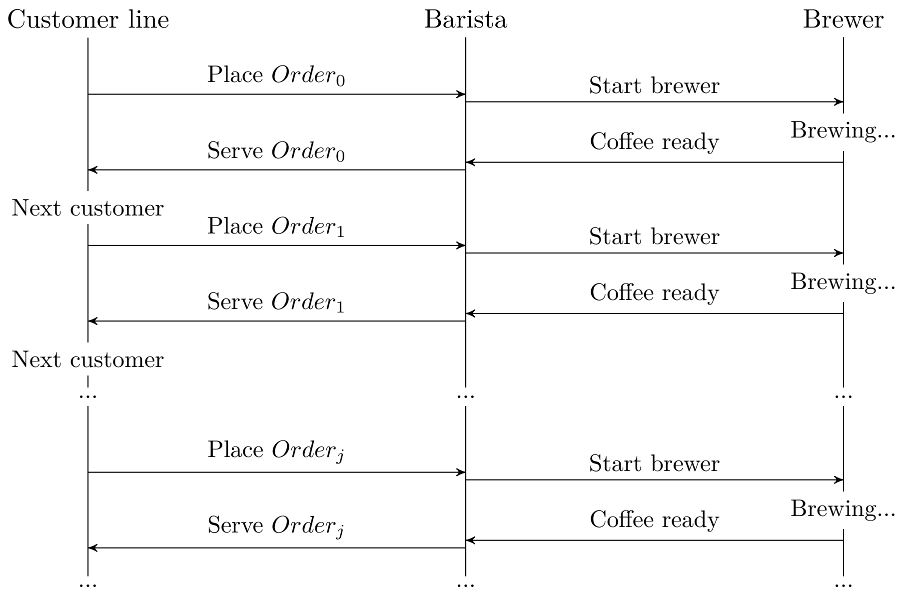

Figure 4.3: A sequence diagram of a serial cafe order line. Each customer is served one at a time and no customer is served before the previous customer receives its order.

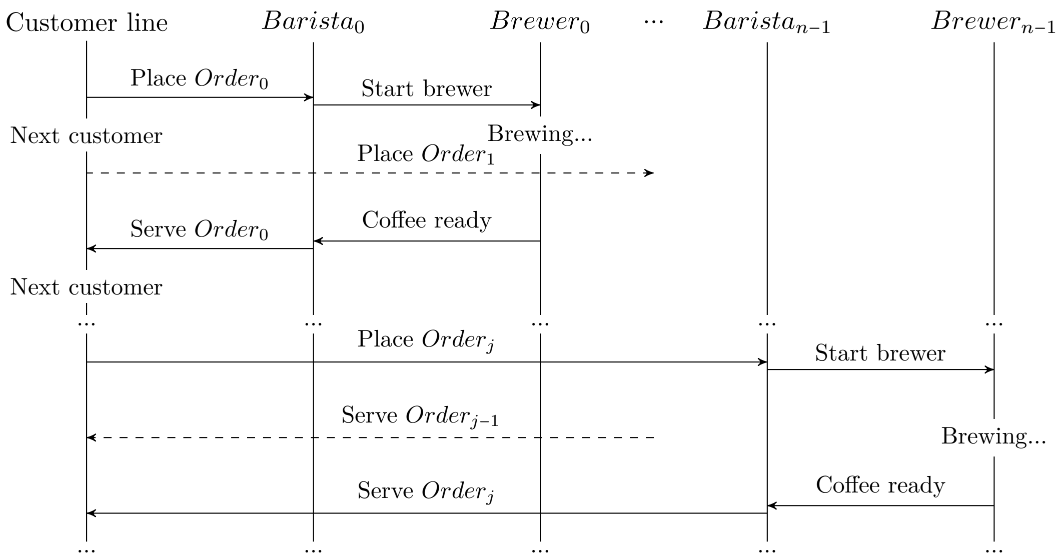

Figure 4.4: A sequence diagram of a preemptive cafe order line. There are n baristas and brewers that can handle orders concurrently and customers does not necessarily have to wait for previous orders to be returned before they can place their own.

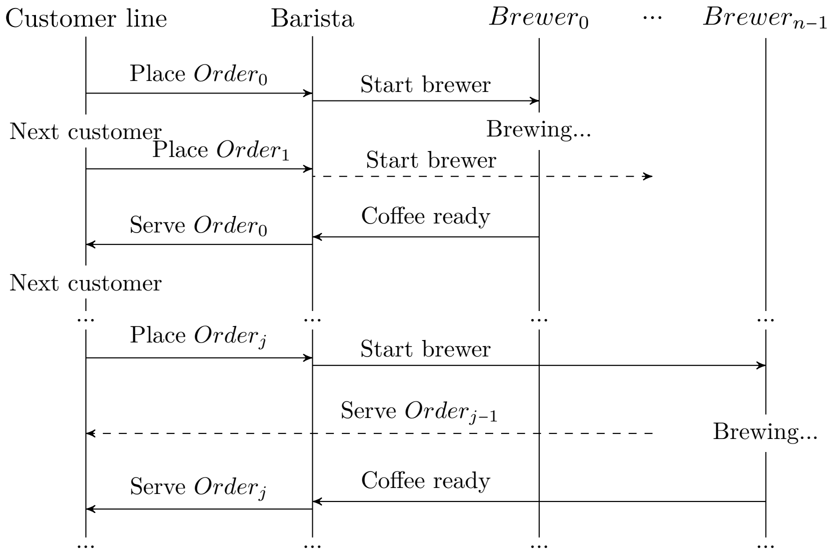

Figure 4.5: A sequence diagram of a cooperative cafe order line. There is only one barista but n brewers. As soon as a brewer is working, the next order can be handled.

Looking at all three of these order handling approaches, we find a few common entities. They all have orders, a customer line, one or several baristas and brewers. Lets generalize this a bit and call orders *events*, the customer line we call the *dispatcher*, the barista and brewer combined we call the *event handler* and the work done by the event handler on event ϵ_j we call Λ_j. The definition of the three event propagation models is as follows:

Serial event propagation: Given a set of m events $\{\epsilon_0, \epsilon_1, ..., \epsilon_j, ..., \epsilon_{m-1}\}$ no event ϵ_j is dispatched before the work Λ_{j-1} is finished, see Figure 4.6.

Preemptive event propagation: Given a set of m events $\{\epsilon_0, \epsilon_1, ..., \epsilon_j, ..., \epsilon_{m-1}\}$ each event ϵ_j can be dispatched independent of Λ_{j-1} and the work Λ_j can implicitly interleave and make room for work Λ_k where $k \neq j$.

Cooperative event propagation: Given a set of m events $\{\epsilon_0, \epsilon_1, ..., \epsilon_j, ..., \epsilon_{m-1}\}$ each event ϵ_j can be dispatched independent of Λ_{j-1} and the work Λ_j can explicitly make room for work Λ_k where $k \neq j$.

Common for all models is that the work Λ_j can not start before the event ϵ_j is dispatched. Each event follows the order $\epsilon_j \rightarrow dispatch \rightarrow \Lambda_j \rightarrow finish$. An example of workflow of a serial event propagation model is illustrated in Figure 4.6. Note that the dispatcher is not necessarily notified of the *finish*-operation. The event handler can finish work on ϵ_j without "returning" anything to the dispatcher.

4.4.1 The anatomy of an event

The event is the actual data transmitted from the device to the gateway. Imagine a small weather station used in private homes. It has two sensors measuring wind speed and air temperature. The gateway is the central system passing the data from each sensor to the application using the data. At a certain frequency, the gateway polls each sensor for new wind speed and temperature values. These values are the actual events in the system. In this toy example, each event might not only carry the actual value of the sensor it corresponds to, but also information on what sensor this data is coming from. The weather application might draw a plot on how the air temperate has changed over time, therefore each event must also

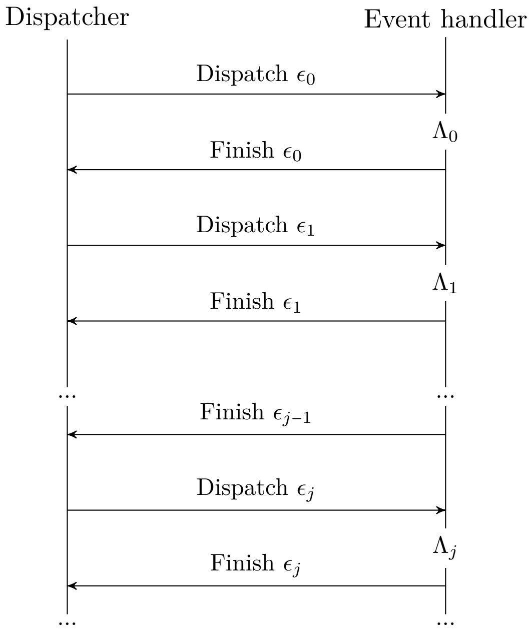

Figure 4.6: A sequence diagram of the serial event propagation model. Only one event ϵ_j is dispatched and handled at a time and no event ϵ_j can be dispatched before the previous work Λ_{j-1} is finished.

include the timestamp of each sensor value. In this study, however, there is no need for event data that represents some mimicked sensor value. Only the frequency of events are important [41]. What is needed is a way to track each event to see when it reaches certain places in its life. This is done by registering the timestamp when the event reaches those places, see Figure 4.7.

An event ϵ can generally be considered a vector:

$$\epsilon = \left[t_0, t_1, ..., t_{k-1}\right]$$

where t_i represents the timestamp when ϵ reached place i in its life. Let k be the number of different places the event can reach. In this study, five places are registered: the creation of the event, when the event leaves the device, when the event reaches the gateway, when the event is dispatched to the event handler and when the event is processed by the event handler.

4.4.2 The dispatcher

The purpose of the dispatcher is to listen to or poll for events and then dispatch them to the event handler. As mentioned in Section 4.3, there are two main approaches to this: pull-based or push-based. The devices generating the events are either active and can push each event to the gateway, which in turn has to listen to them, or the devices are passive and the gateway must pull events from them. In this study, the pull-based approach is used to mimic the gateway used by Customer A. This approach requires the dispatcher to have a reference to each device to be able to pull events from them.

An important question to consider is: how is the dispatcher related to the event propagation models? If the architecture of the gateway implements the serial event propagation model, the dispatcher should make sure it does not dispatch an event before the previous event is dispatched, see Figure 4.8. Both the preemptive and the cooperative event propagation models loosens this constraint and allows the dispatcher to dispatch events before previous events have been dispatched. In contrast to the serial approach in Figure 4.8, all devices can be polled for

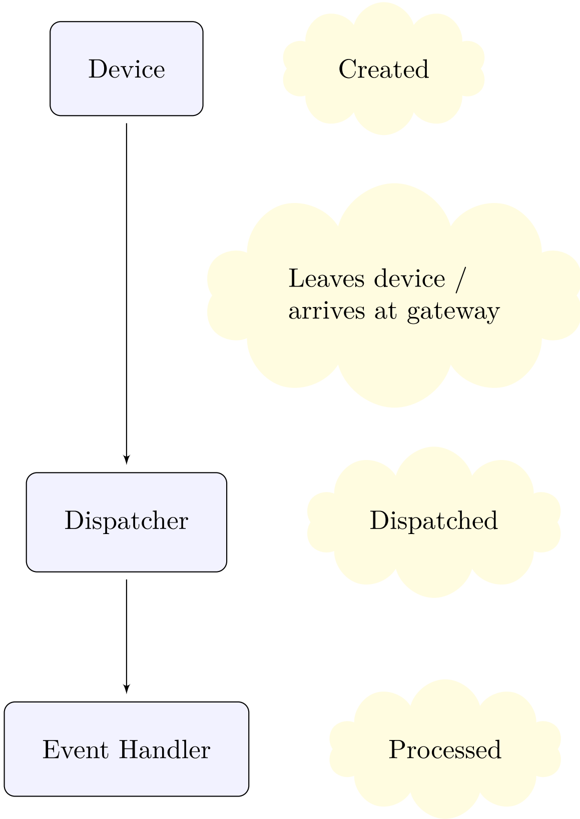

Figure 4.7: Overview of the event lifecycle. The event travels from the device down to the event handler. At each step the time point is registered, shown in the clouds.

events concurrently. There is no need, in theory, to loop through the list of device references and check them one by one. Instead, there can be one dispatch process for each device, see Figure 4.9.

What differentiates the cooperative dispatcher from the preemptive, though, is that the dispatcher interleaves between devices implicitly and explicitly respectively. The preemptive dispatcher can be implemented by creating one thread per device and simply run a serial dispatcher for each thread that only dispatch events from one device. The concurrency mechanism in the operating system scheduler will perform the interleaving, thus making it implicit for the user. The cooperative approach can be implemented with libuv, which has been done in this study. By creating one state machine (like the one in Figure 4.9) per device and by letting the edges represent asynchronous network transmissions, other device states can do work while the network transmission is on the line. The explicit interleaving is happening when the user initiates an asynchronous network transmission. The cooperative approach only requires one thread, utilizing the asynchronous features of the I/O drivers to the operating system.

Regarding resource usage in the different dispatcher models. The serial approach will only communicate with one device at a time, thus only one socket is required to be open concurrently. In resource usage, this can scale very well. If the list of device references becomes too large for memory, it can be stored in a database. The preemptive and cooperative approaches will in theory communicate with all devices concurrently, so there must be one socket available for each device concurrently. This does not scale well since only a limited amount of sockets are allowed by the operating system to be open. This can however be solved by breaking down the devices into chunks that are manageable by the gateway and process one chunk at a time.

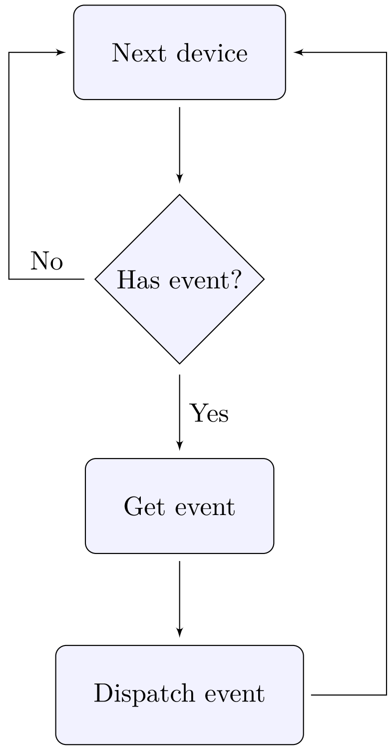

Figure 4.8: An illustration of the serial dispatcher. When ”Next device” has reached the end of the list it restarts from the beginning again, thus the loop never breaks. Each node in this model blocks the rest of the execution. For instance, getting an event means sending a request over the network to the device which in turn responds with the latest event. The whole process is then blocked during the time it takes for the message to travel across the network.

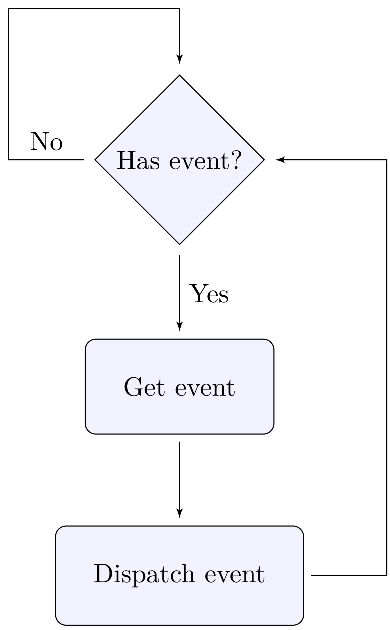

Figure 4.9: An illustration of a preemptive and cooperative dispatcher. Each device can have their own dispatch process, so there is no need to loop through the list of device references.

4.4.3 The event handler

The purpose of the event handler is to process the event ϵ. What type of processing to be done depends on the application and what the event represents. Returning to the weather station analogy; one application for the events received from the temperature sensor could be to monitor the change in temperature over time. This requires each event to be stored in some sort of persistent data storage, e.g. a database. This database could either be found locally on the gateway's filesystem or on some remote internet service. Perhaps the weather station has a feature that predicts the future wind velocity in real time. This requires some statistics or machine learning calculation to be done every time a new event arrives at the gateway. Generalizing this idea, we get two different types of calculations: one depending on the CPU and its memory, and one depending on some peripheral unit of the CPU, for instance the file system or the network unit. The first we call CPU-work, or λ_j^{cpu}, for it is work entirely performed by the CPU. The latter we call I/O-work, or λ_j^{io}, for it is work not necessarily done by the CPU but by some peripheral I/O unit. The definition then follows that the total work done by the event handler $\Lambda_j = \lambda_j^{cpu} + \lambda_j^{io}$ for an event ϵ_j. For simplification, it is assumed that work Λ_j is the same for each event. The j subscript is removed and for consistency with the rest of the paper the CPU- and I/O work are denoted λ_0 and λ_1, respectively. Worth noting is that it is not necessarily always the case that λ_0 is disjunct from λ_1, they might depend on each other to some extent.

The serial event handler will perform λ_0 and λ_1 synchronously, one after another, see Figure 4.10. The entire work will block the rest of the execution.

$$\lambda_0 \longrightarrow \lambda_1 \longrightarrow \text{Done}$$

Figure 4.10: Workflow of the serial event handler.

The preemptive event handler will conduct serial work on multiple threads, one thread per event, see Figure 4.11. The operating system scheduler will perform what the user perceives as implicit interleaving.

$$Thread_0 \dashrightarrow \lambda_0 \longrightarrow \lambda_1 \longrightarrow \text{Done}$$

$$Thread_1 \dashrightarrow \lambda_0 \longrightarrow \lambda_1 \longrightarrow \text{Done}$$

...

Figure 4.11: Workflow of the preemptive event handler. Serial work is done on multiple threads.

The cooperative event handler will perform CPU work on a single thread, see Figure 4.12. However, I/O work will be dispatched asynchronously to the associated I/O driver. Work can in that way be parallelized with only one working thread.

$$\begin{tikzcd} & \lambda_1 \\ \lambda_0 \arrow[ur, dashed] \arrow[r] & \text{Done} \end{tikzcd}$$

Figure 4.12: Workflow of the cooperative event handler. CPU work λ_0 is performed on a single thread, I/O work λ_1 is however performed asynchronously, thus letting event handler return immediately after λ_0 is done.

4.5 An abstract gateway

A model of an abstract gateway is proposed as follows: a gateway Γ is a six-tuple describing its internal properties and its environment.

$$\Gamma = \langle \Omega, \phi, \delta, K, \Lambda, \chi \rangle \tag{4.1}$$

Let Ω, ϕ and δ describe the properties of the environment of Γ, also called the *configuration* of Γ. The following parameters are listed in Table 4.1 for easier reading. Ω describes the devices communicating with Γ and can be expressed as a set $\Omega = \{\omega_0, \omega_1, ..., \omega_{q-1}\}$ where ω_i represents a device and q is the total number of devices. Let ϕ be the number of events ω_i generates each second and let δ be the time it takes for a network request to travel from ω_i to Γ, or vice versa. K and Λ describe the internal properties of Γ. Let K describe the event propagation model of Γ. It is expressed as $K = \{\kappa_0, \kappa_1\}$ where κ_0 and κ_1 represent the event propagation model of the dispatcher and the event handler respectively and $\kappa_i \in \{\mathit{serial}, \mathit{preemptive}, \mathit{cooperative}\}$. Let Λ describe the work induced on Γ from each event. It is expressed as $\Lambda = \{\lambda_0, \lambda_1\}$ where λ_0 and λ_1 represents the CPU intensity and the I/O intensity induced by each event, where $0 \le \lambda_i \le 1$. Let χ describe the hardware platform of Γ, specifically the number of cores on the CPU.

Table 4.1: An overview of the internal and external properties of Γ.

Parameter	Description
q	Number of devices.
ϕ	Frequency of generated events.
δ	The delay, or latency, added by the network.
κ_0	Event propagation model of the dispatcher.
κ_1	Event propagation model of the event handler.
λ_0	CPU intensity induced by each event.
λ_1	I/O intensity induced by each event.
χ	Number of CPU cores.

In addition to the rather static expression of an abstract gateway Γ, the function E is provided:

$$E(\Gamma, t_{start}, t_{end}) \to M = \left[\epsilon_0, \epsilon_1, ..., \epsilon_{m-1}\right]^T \tag{4.2}$$

By providing E with Γ and two points in time t_{start} and t_{end} where $t_{start} \le t_{end}$, a vector M of ordered events ϵ_j is returned, where $\epsilon_j = [t_0^j, t_1^j, t_2^j, t_3^j, t_4^j]$. Note that M is ordered by t_0^j and m is the total number of events. Each ϵ_j include five timestamps t_k^j that represents parts of the event lifecycle:

t_0^j represents the point in time when ϵ_j was created in ω_i.

t_1^j represents the point in time when ϵ_j left ω_i.

t_2^j represents the point in time when ϵ_j arrived at Γ.

t_3^j represents the point in time when ϵ_j was dispatched to the event handler in Γ.

t_4^j represents the point in time when Γ finished processing ϵ_j.

These timestamps hold a set of basic constraints:

$$t_{start} \le t_k^j \tag{4.3}$$

$$t_0^j \leq t_{end} \tag{4.4}$$

$$t_0^j \leq t_1^j \leq t_2^j \leq t_3^j \leq t_4^j \tag{4.5}$$

Let Γ_c be a specific gateway with the configuration c. Running $E(\Gamma_c, t_{start}, t_{end})$ returns the vector M_c, which contains the events produced by Γ_c.

4.5.1 Performance metrics

With M and ϵ_j it is possible to derive some performance metrics. Let π be a function such that:

$$\pi(\epsilon_j) = \begin{cases} 1, & \text{if } t_4^j \leq t_{end} \\ 0, & \text{otherwise} \end{cases}$$

If ϵ_j has been processed, i.e. its last timestamp t_4^j happened before the gateway stopped, π returns 1. Using this function, throughput T can be calculated as:

$$T = \frac{\sum_{\epsilon_j \in M} \pi(\epsilon_j)}{t_{end} - t_{start}} \text{ [events / s]}$$

It is also possible to analyze the response time for an event, how long time it spent in ω_i and how long time it spent in Γ. Let:

$d_0^j = t_4^j - t_0^j$ be the response time of ϵ_j, i.e. the total time between the event was created until it was processed.

$d_1^j = t_1^j - t_0^j$ be the wait time, i.e. the time ϵ_j spent in ω_i.

$d_2^j = t_4^j - t_2^j$ be the processing time, i.e. ϵ_j spent in Γ.

Given a vector of events M_c, with length m, created by running E on a gateway with the configuration c, let D_k^c be a vector containing $\left[d_k^0, d_k^1, ..., d_k^{m-1}\right]$, where $k = 0, 1, 2$, d_0^j is the response time, d_1^j is the wait time and d_2^j is the processing time for $\epsilon_j \in M_c$. Then:

$avg(D_k^c)$ is the average value of all $d_k^j \in D_k^c$.

4.5.2 Analyzing the load

A system is said to be *at load* when the system is working at full capacity and is not able to produce higher throughput. Due to the strict time limitation provided by t_{end} there might be cases where some events ϵ_j have $t_k^j > t_{end}$. These are events that haven't been entirely processed by Γ. The ratio between created events and processed events can give hint on how many events Γ can handle. Let P denote the ratio between created and processed events:

$$P = \frac{P_\pi}{m} = \frac{\sum_{\epsilon_j \in M} \pi(\epsilon_j)}{m}$$

P will always be in the interval $0 \leq P \leq 1$, because t_4^j can be greater than t_{end}, which means $\pi(\epsilon_j) = 0$, thus $m \geq P_\pi$. In cases where $P \simeq 1$, Γ is able to process most of the events and it might even be so that Γ is waiting for new events to be created. In that case, we can not be sure whether Γ is at a proper load or not. However, in cases where $P \ll 1$ we know for sure Γ is not able to process all events, so there must be events on the device waiting for Γ. The response time, $t_4^j - t_0^j$, must therefore grow for each new event that is created, leading to an unfair image of the response time. A proper load is therefore one where $P \simeq 1$, but where an increase in total amount of events m only reduces P.

4.6 Approach

The goal of this thesis is to study how different gateway architectures affect its performance in different environments. Using the abstract gateway model Γ, it is possible to derive the performance mapping. By running E with a set of different gateways Γ_c and analyze the throughput, load, response time, wait time and process time on the resulting vector M_c the performance mapping will be developed. The abstract gateway will be implemented to support a wide range of its possible configurations. In theory, the abstract gateway allows different combinations of dispatcher and event handler event propagation models. Due to lack of time, not all of them will be supported by the implementation, see Table 4.2.

Table 4.2: Overview of the combination of event propagation models available for the dispatcher and event handler.

Dispatcher	Event handler	Comment
Serial	Serial	Implemented
Serial	Preemptive	Implemented
Cooperative	Preemptive	Implemented
Cooperative	Cooperative	Implemented
Serial	Cooperative	Not implemented
Preemptive	Serial	Not implemented
Preemptive	Preemptive	Not implemented
Preemptive	Cooperative	Not implemented
Cooperative	Serial	Not implemented

To mimic the gateway hardware used by Customer A (see Section 4.3) a Raspberry Pi 3 Model B[5] is used to run the gateway implementation. The system that simulates the devices and analyses the resulting events (later known as the *test manager*, see Chapter 5) is run on a Macbook Pro[6]. The hardware used are described in Table 4.3.

Table 4.3: Overview of the hardware used in the performance tests.

Usage	Model	CPU	Memory size
Gateway	Raspberry Pi 3 Model B	Quad Core 1.2GHz Broadcom BCM2837 64bit	1 GB
Test manager	Macbook Pro	Dual Core 2.6GHz Intel Core i5	8 GB

The configuration parameter χ allows the number of cores to change. This is implemented by setting a flag in the operating system running the Raspberry Pi that sets the maximum number of cores used. Only wireless LAN is used between the test manager and the gateway.

[5] `https://www.raspberrypi.org/products/raspberry-pi-3-model-b/`
[6] `https://support.apple.com/kb/sp703?locale=sv_SE`

5 Implementation

To be able to reach the goal of this thesis, to map the performance of the abstract gateway Γ, a software system has been developed. The ultimate purpose of this system is to generate and store all events and their respective timestamps to be able to analyze the performance of Γ, which processed them. The system consists of two processes: a *test manager* and a *gateway* as well as an analyzing tool. The main purpose of the test manager is to simulate devices and store event data in a database while the purpose of the gateway is to pull events from the devices and process them. The analyzing tool is not discussed more than that it provides utility functions to assemble event timestamp information from one or several test runs available in the database and output them as comma separated values (*csv*).

The test manager is entirely built in Python with standard libraries. SQLite is used for the database. Figure 5.1 depicts the overall architecture of the test manager. Each part is explained in the following sections. The main thread of the program is the actual test manager that waits for new event lifecycle messages from the log server, which acts as the main source of event-related information. The test manager takes each new event lifecycle message and store it in the database. One thread is created for the name service, which is a TCP server hosting the name service API. One thread is also created for each device in the test as they are TCP servers hosting the device API used by the gateway. The producer is also working in its own thread, creating new events at a given frequency. The log server is a UDP server listening for UDP log messages in its own thread. The ”Net”-module implements the middleware functionality, enabling communication between the test manager and the gateway.

The gateway is entirely built in C with standard libraries and the libuv library. The overall architecture of the gateway is shown in Figure 5.2. The boot service is partly a short lived TCP server listening for timestamp request and start events from the test manager's boot service. The dispatcher and the event handler have different configurations, depending on the event propagation model, see Sections 5.7 and 5.8.

5.1 Configuration

The test manager and the gateway processes must be configurable in order to test different configurations of the gateway. Both the test manager and the gateway process are started from the command line and the configuration parameters are set with flags to the respective command, see Listings 5.1 and 5.2.

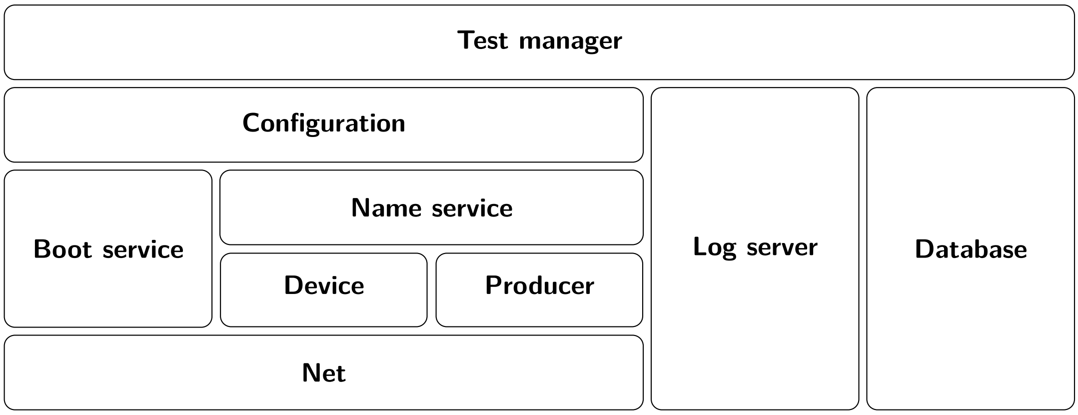

Figure 5.1: The architecture of the test manager.

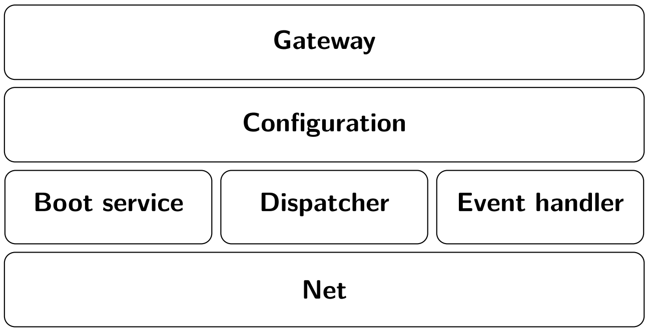

Figure 5.2: The architecture of the gateway.

Listing 5.1: Usage manual for the test manager process.

```
usage: ./run_test [<options>...]

OPTIONS
    -h
        Show this message.

    -q <value>
        The number of simulated devices to run in the test.

    -f <value>
        The frequency in which one device creates new events, expressed in
        times per seconds.

    -l <value>
        The delay expressed in seconds added to each TCP call.

    -d <architecture>
        The architecture of the event dispatcher. Can be one of the following:
            serial
            preemptive
            cooperative

    -e <architecture>
        The architecture of the event handler. Same alternatives as for the
        dispatcher.
```

```
    -c <value>
        The CPU intensity each event induce. Value between 0 and 1.

    -i <value>
        The I/O intensity each event induce. Value between 0 and 1.

    -p <value>
        The size of the thread pool. Defaults to 10.

    -t <time value>
        The duration for the test to run. Written in hours, minutes and
        seconds. E.g. 1h2m or 1m10s.

    -g <value>
        The log level to print to stdout. One of 0 (DEBUG), 1 (INFO, default)
        and 2 (ERROR).

    -b <path>
        The file path and name to the database. Defaults to "db".

    -r <value>
        The name of the report to connect this test scenario to. Defaults to
        "Undefined report".
```

Listing 5.2: Usage manual for the gateway process.

```
usage: ./gateway [<options>...]

OPTIONS
    -h
        Show this message.

    -d <architecture>
        The architecture of the event dispatcher. Can be one of the following:
            serial
            preemptive
            cooperative

    -e <architecture>
        The architecture of the event handler. Same alternatives as for the
        dispatcher.

    -c <value>
        The CPU intensity each event induce. Value between 0 and 1.

    -i <value>
        The I/O intensity each event induce. Value between 0 and 1.

    -t <address>
        The IP address (excluding port) of the test manager.

    -n <port>
        The port of the nameservice.

    -l <port>
        The port of the log server.

    -p <value>
        The size of the thread pool. Defaults to 10.
```

5.1.1 Simulating CPU intensity

The purpose of the CPU intensity λ_0 is to simulate CPU intensive work induced by each event. The intensity value can be set in the configuration to be a value between 0 and 1, where 0 means no intensity and 1 means maximum intensity on each event. CPU intensity is simulated by calculating the nth prime number, where $n = \lfloor 2^{12} \times \lambda_0 \rfloor$. Each calculation is done on one thread per event.

5.1.2 Simulating I/O intensity

The purpose of the I/O intensity λ_1 is to simulate I/O intensive work induced by each event. The intensity value is set to be between 0 and 1, where 0 means no I/O work and 1 means maximum I/O work for each event. I/O work can be any type of work performed by some peripheral unit to the CPU. In this case, file system operations were chosen. A buffer of size n, where $n = \lfloor 2^{28} \times \lambda_1 \rfloor$, is allocated, filled with letters and written to a file.

5.2 Communication and protocols

TCP and UDP technologies are used as communication methods between the test manager and the gateway, see Figure 5.3. UDP is not connection-oriented like TCP and is able to send messages with less overhead. UDP is therefore used as medium to send log messages to the log server, both by the test manager and the gateway. Each log message is sent in a specific format and is explained in Section 5.5. The TCP communication middleware is implemented using *Remote Method Invocation* (RMI) [55] and is placed in the ”Net”-module, see Figures 5.1 and 5.2. For the gateway to call an API function on the test manager (or vice versa), it must produce a JSON-string message in the format:

```
{
    "name": "hostnames",
    "args": []
}
```

If the gateway sends this message to the test manager, the API function `hostnames()` will be invoked and its result will be sent back to the gateway in the format:

```
{
    "result": [5000, 5001, 5002, 5003]
}
```

However, if the RMI induce an error while being processed by the API, the error is returned instead of the result. The error is sent back in the format:

```
{
    "error": {
        "name": "AttributeError",
        "args": ["The API-function is not available."]
    }
}
```

The name of the error and the arguments used to create the error are sent back and enable the receiving end to throw the error as if it was created locally.

5.3 The bootup service

The test manager and the gateway are run as two different processes on two different machines. They are started manually from the command line with the appropriate flags and configurations. Due to the human factor, there is a possibility that the gateway is started with a configuration different from the one on the test manager. The test manager can for instance

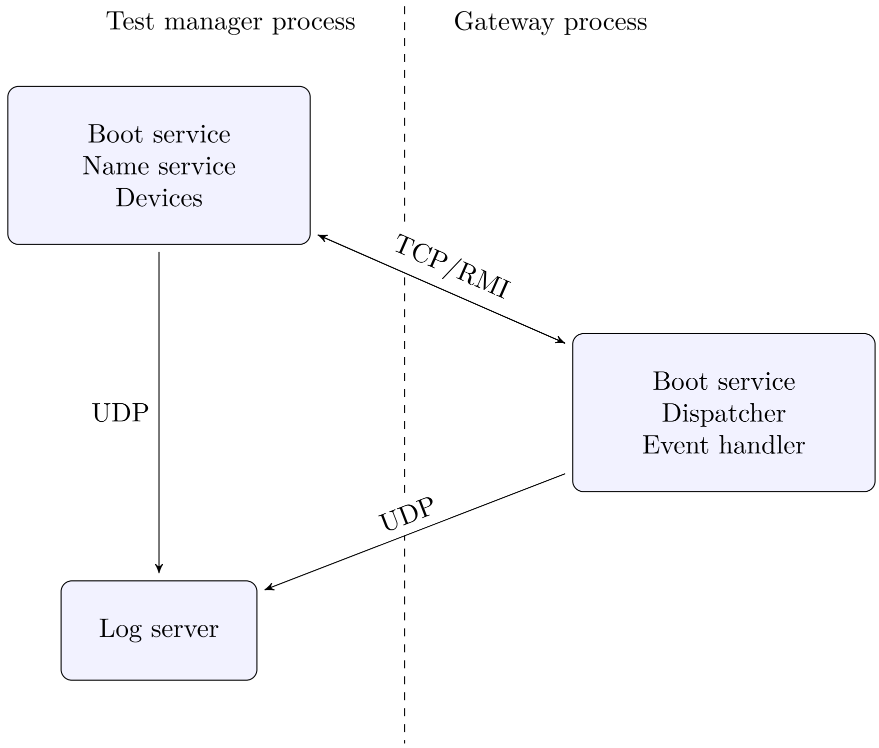

Figure 5.3: Illustration of the communication flow between the test manager and the gateway.

be started with an I/O intensity value of 0, while the gateway can be started with an I/O intensity of 1. The test report will then believe the test was run with a different configuration than it actually was. Another issue is that the timestamps recorded from the events are set on both machines. The ”Created” and ”Fetched” timestamps are set on the test manager machine, while the rest of the timestamps are set on the gateway machine. If the clocks differ on the two machines the performance analysis is not trustworthy. It is therefore important to get the time offset between the test manager and the gateway.

The test bootup process solves these two issues by verifying the configuration between the two instances and the time offset between them. The configuration verification is done by letting the gateway send its configuration to the test manager who checks that the configurations match. If they don't, the test will end with an error stating that the configurations did not match. If they match, however, the test manager will start a time synchronization procedure that calculates the time offset between the two machines. This offset will be added to each event timestamp retrieved from the gateway. The algorithm that calculates the time offset has been used in video games earlier and is as follows [56]:

1. The test manager saves current local time (t_{tm_0}) and requests the local time from the gateway.

2. Upon receipt, the gateway's local time (t_{gw}) is returned to the test manager.

3. The test manager saves the current local time (t_{tm_1}) again and calculates the latency with $t_l = \frac{t_{tm_1} - t_{tm_0}}{2}$. The current time offset $t_0 = t_{gw} - t_{tm_1} + t_l$ is stored in a list.

4. Steps 1-3 are repeated five times with a second pause between each time. This will populate a list with five offset values t_0 to t_4. The values are sorted incrementally and the median value is used as the final time offset value.

Once the configuration has been verified and the time offset is identified, the test is started by the test manager by calling the "start_test" API function on the gateway. The entire test scenario is stored in a database as a table with four attributes: *scenario ID*, *time offset*, *start time* of the test and its *end time*. The entire bootup sequence is illustrated in Figure 5.4.

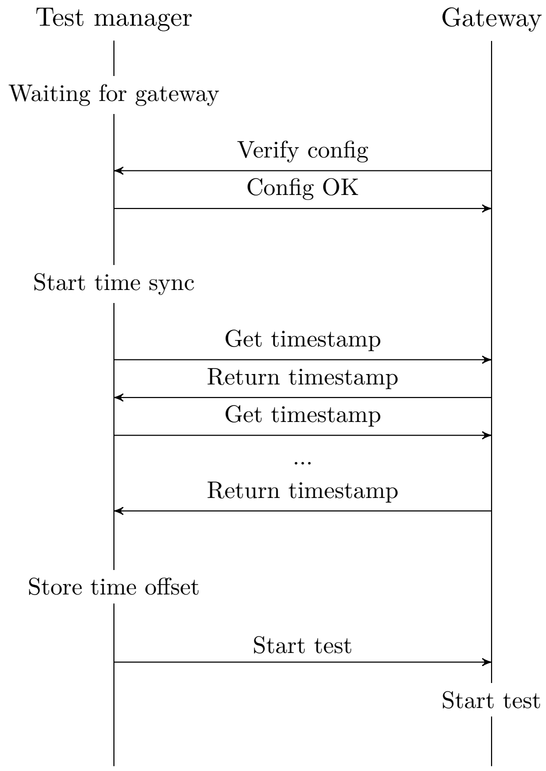

Figure 5.4: Sequence diagram of the bootup process. The test manager waits for the gateway to verify its configuration settings. If the configuration does not match the one on the test manager, the test is aborted (not shown in this figure). If the configuration matches, the test manager starts the time sync process that retrieves the timestamp from the gateway a couple of times with a 1 second pause between each call. The offset is then stored in the database and the test can start.

The gateway bootup process starts a TCP server hosting an API used by the test manager with the following functions:

`get_timestamp():` Returns the timestamp in milliseconds on the gateway machine.

`start_test():` Signals the gateway that the bootup process is finished and that the test can start. The gateway will start communication with the devices.

5.4 The name service

The name service is a TCP server instantiated and started by the test manager. Its purpose is to act as the main API available for the gateway to communicate to the test manager. It also holds references to all simulated devices the gateway pulls event information from. The name service provides a single API for the gateway:

`verify_gateway(configuration, address):` Verifies that the gateway configuration matches the test manager configuration. The address of the gateway API is passed in as well for later use. Returns true if the configuration matches, false otherwise.

`hostnames():` Returns a list of socket ports associated to all devices in the test.

When conducting performance tests, the payload data in each event is irrelevant, only the frequency of events [41]. The events are therefore nothing but an identifier. A device is implemented as a TCP server class with one primary attribute: an event queue. The event queue stores each event in a FIFO manner (first in, first out). An event is implemented as a class with a single attribute: its ID, which is a 64-bit UUID string. The device does not generate events by itself, instead an *event producer* runs on a separate thread and for a given time interval it pushes new events onto each device. A class diagram over these components is shown in Figure 5.5. Each device is its own TCP server, listening to TCP requests on an IP address. The gateway communicates to each device via its API:

`status():` Returns 1 if there is at least one event ready to be fetched from the device's event queue. Returns 0 otherwise.

`next_event()` Returns the next event in the queue. Throws and returns an error if the event queue is empty.

Note that the method `put_event()` in the Device class in Figure 5.5 is not part of the TCP server API of the device. This is because the method is only available locally for the EventProducer class.

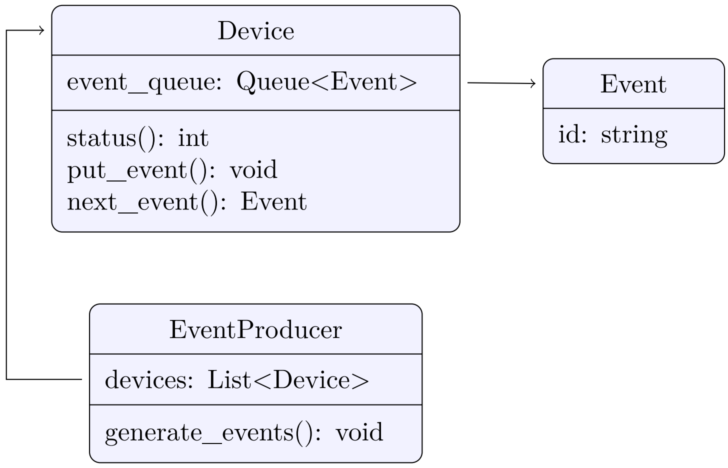

Figure 5.5: Class diagram of the dependencies between the Device, EventProducer and Event class.

5.5 The log server

The purpose of the log server is to act as a global logging portal for both the test manager and the gateway and to extract event lifecycle information from the logs. If the output flag is on, it prints each log message to the same standard output, regardless of whether the origin of the message is from the test manager or the gateway. It checks each message for event lifecycle information and sends the extracted data to the test manager for database storing. All log messages follow the same format: `<level>:<timestamp>:<message>`, e.g. `INFO:0123456789:Gateway configuration ok!`. There are four hierarchical, ordered "greater than" log levels available: `VERBOSE`, `DEBUG`, `INFO` and `ERROR`. They are hierarchical and ordered "greater than" in the sense that if the level is set to `VERBOSE`, all log messages in the higher levels are shown as well. If the level on the other hand is set to `INFO` the `VERBOSE` and `DEBUG` messages are not shown. Event lifecycle messages are in the format `<level>:<timestamp>:<function>:<keyword>:<event_id>` where keyword is one of:

EVENT_LIFECYCLE_CREATED: The event `event_id` was created at time `timestamp`. The log server extracts the relevant data from the message using regular expressions and sends them to the test manager who inserts them in the database.

EVENT_LIFECYCLE_FETCHED: The event `event_id` left the device at time `timestamp`.

EVENT_LIFECYCLE_RETRIEVED: The event `event_id` arrived at the gateway at time `timestamp`.

EVENT_LIFECYCLE_DISPATCHED: The event `event_id` was dispatched to the event handler at time `timestamp`.

EVENT_LIFECYCLE_DONE: The event `event_id` finished processing at time `timestamp`.

A regular expression is used both to check whether the message is an event lifecycle message and to extract the relevant values from it. The log server is run on a separate thread in the test manager process and listens for UDP packets. Both the test manager and the gateway send log messages as UDP packets to the log server.

5.6 The database

The primary use of the database is to store event lifecycle timestamps during the test run so they can be analyzed at a later stage. The database schema is designed to order event lifecycle data with their corresponding test scenario, see Figure 5.6. The test scenario, with its ID (sid), is built around the notion that a test run has a start and an end time and n amount of events created during the test. The event life cycle has 5 timestamps associated with it; created, fetched, retrieved, dispatched and done time. Each event has an event ID (eid) and is associated with a test scenario. Each scenario has a set of configuration values that are associated with their keys, which are: `CPU_INTENSITY`, `DEVICE_DELAY`, `DEVICE_FREQUENCY`, `DEVICE_QUANTITY`, `DISPATCHER`, `EVENT_HANDLER`, `IO_INTENSITY`, `POOL_SIZE`, `TEST_DURATION`. Test scenarios are grouped into reports. For instance, if a CPU intensity test is planned where different values of CPU intensity are run, each test scenario will be associated with a new report that is named based on the test, e.g. ”test cpu 0.1-0.5” if the CPU intensity is tested with values between 0.1 and 0.5.

When a test is about to start, the user specifies the configuration of the test in the command line, see Section 5.1. A new record for each configuration key is created in the configuration table. When the test starts, a new scenario record is created with its start time set. During the time sync process, see Section 5.3, the time offset between the test manager and the gateway is identified and set in the offset attribute in the scenario record. When a new event is created, the device sends a log message to the log server, which in turn extracts the relevant data from the message and creates a new eventlifecycle record with the created_time attribute set. The device and the gateway send log messages as the event passes the associated lifecycles and the log server updates the eventlifecycle record. When the test stops the test manager sets the end timestamp in the scenario record and an entire scenario is stored.

5.7 The dispatcher

Two event propagation models were implemented for the dispatcher: the serial and the cooperative. The serial dispatcher was implemented using standard libraries in C, see Listing 5.3. This function loops endlessly over each device retrieved from the `hostnames()` API function on the name service. In each iteration a `status()` call is made to the device. If the response equals 1, the event is retrieved by `next_event()` and dispatched to either the serial or the preemptive event handler, depending on the configuration.

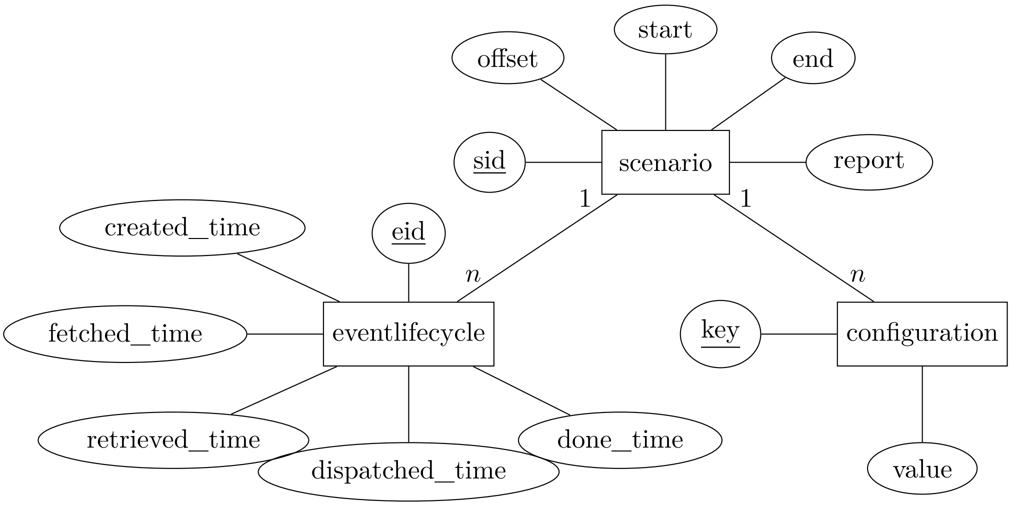

Figure 5.6: Entity relationship diagram of the database schema.

Listing 5.3: Code snippet from the serial dispatcher. Some parts has been left out in the `/* */` markings.

```
void dispatcher_serial(config_data_t* config, protocol_value_t* devices)
{
    int i = 0;

    /* ... */

    while (1) {
        net_tcp_context_sync_t* device = devices_context[i];
        int status_ok;

        /*
        make the status() RMI-call and set the result in status_ok...
        */

        if (status_ok) {
            /*
            make the next_event() RMI-call and set the result in device->event...
            */

            // send log messages to the log server
            // these two lines means there is theoretically no time difference
            // between the retrieved- and the dispatched-timestamp.
            log_event_retrieved(device->event);
            log_event_dispatched(device->event);

            // call serial event handler
            if (strcmp(config->eventhandler, "serial") == 0) {
                event_handler_serial(config->cpu, config->io);
            }
            // call preemptive event handler
            else if (strcmp(config->eventhandler, "preemptive") == 0) {
                /*
                wait for thread pool queue
                */

                event_handler_preemptive(device);
            }
        }
```

```
        i = (i + 1) % devices_len;
    }
}
```

The cooperative dispatcher was implemented using libuv. Listing 5.4 shows a snippet of the cooperative dispatcher function. The serial dispatcher performs socket operations synchronously, i.e. the entire process is paused while waiting for RMI responses to return. The cooperative dispatcher uses a state machine functionality, where nodes are represented as state callback functions. A state callback can make an asynchronous RMI call and pass in the edge to traverse to when the RMI response has returned. Each asynchronous RMI call wraps libuv socket operations and returns instantly. That allows the process to continue despite having socket transmissions on the line. The state machine approach is useful for reactive systems, such as the gateway, when many combinations of inputs are to be recognized [7].

Listing 5.4: Code snippet from the cooperative dispatcher. Some parts have been left out in the `/* */` markings.

```
void dispatcher_cooperative(config_data_t* config, protocol_value_t* devices)
{
    // create the cooperative state machine
    state_t* coop_dispatch = machine_cooperative_dispatch();

    // just loop each device once
    for (int i = 0; i < devices_len; ++i) {
        machine_coop_context_t* context;

        /* ... */

        // start the state machine for this device, this function returns
        // immediately
        state_machine_run(coop_dispatch, context);
    }

    // start the libuv event loop, this function will not return
    uv_run(loop, UV_RUN_DEFAULT);
}
```

5.8 The event handler

Three event propagation models were implemented in the event handler: the serial, preemptive and cooperative. The serial event handler performs the CPU work and the I/O work, one after another synchronously, blocking the rest of the execution, see Listing 5.5.

Listing 5.5: Code snippet inspired by the code from the serial event handler implementation.

```
void event_handler_serial(double cpu_intensity, double io_intensity)
{
    do_cpu_work(cpu_intensity);
    do_io_work(io_intensity);
}
```

The preemptive event handler dispatches each event to a thread pool and returns immediately, see Listing 5.6. If all threads in the pool are busy with previous events, the event handler waits for a thread to be free and blocks the execution on the main thread. The work each thread performs is identical to the serial event handler.

Listing 5.6: Code snippet inspired by the code from the preemptive event handler implementation.

```
void event_handler_preemptive(double cpu_intensity, double io_intensity)
{
    thpool_add_work(
        threadpool,
        &event_handler_serial,
        cpu_intensity,
        io_intensity);
}
```

The cooperative event handler will perform CPU work first, on the same thread, blocking the rest of the execution, followed by the I/O work that is performed asynchronously and returns instantly, see Listing 5.7. The I/O work will be performed using libuv's filesystem API. libuv performs filesystem operations in its built-in threadpool. That means the only difference between the preemptive and the cooperative event handler is that the cooperative performs CPU work first, before dispatching the work further to the threadpool. The internal threadpool of libuv gets the same size as the number of threads configured when the gateway starts, but if the threadpool is full the cooperative event handler will not wait for a thread to be free before the I/O work can be dispatched. It will queue the work and continue the normal dispatcher/event handler execution.

Listing 5.7: Code snippet inspired by the code from the cooperative event handler implementation.

```
void event_handler_cooperative(double cpu_intensity, double io_intensity)
{
    do_cpu_work(cpu_intensity);
    do_io_work_async(io_intensity); // returns immediately
}
```

6 Results

The results of the performance tests conducted are presented here. The chapter starts with a table describing each test case (Table 6.1) and what variable was adjusted in it, followed by sections describing each test case and explaining their respective outcome. The figures include four series from the four tested event propagation models and the legend is interpreted as:

s/s: Serial dispatcher with a serial event handler.

s/p: Serial dispatcher with a preemptive event handler.

c/c: Cooperative dispatcher with a cooperative event handler.

c/p: Cooperative dispatcher with a preemptive event handler.

Table 6.1: Overview of the performance test cases and their configurations. The columns with arrows (→) are the variables being adjusted in that specific test case.

#	Related figures	Cores χ	Quantity q	Delay δ [s]	CPU int. λ_0	I/O int. λ_1
1	6.1, 6.2, 6.3	4	$1 \rightarrow 50$	0	0.1	2×10^{-4}
2	6.4, 6.5	4	25	$0.01 \rightarrow 0.1$	0.1	2×10^{-4}
3	6.6, 6.7, 6.8, 6.9	4	25	0	$0.1 \rightarrow 0.5$	10^{-4}
4	6.6, 6.7, 6.8, 6.9	1	25	0	$0.1 \rightarrow 0.5$	10^{-4}
5	6.10, 6.12, 6.11	4	25	0	0.05	$10^{-4} \rightarrow 10^{-3}$
6	6.10, 6.12	1	25	0	0.05	$10^{-4} \rightarrow 10^{-3}$

After a test has finished, each event has timestamps associated with its lifecycle in the database. This data is used to calculate throughput, load, response time, time on device and time on gateway, see Section 4.5.1. Response time is defined as the done timestamp subtracted with the created timestamp. Time on device is defined as the fetched timestamp subtracted with the created timestamp. Time on gateway is defined as the done timestamp subtracted with the retrieved timestamp. The retrieved timestamp is registered when the dispatcher has read the TCP message on the socket, not when the message arrived on the socket. This has the implication that time on device plus time on gateway is not necessarily equal to the total response time. Some time is ”lost” between the fetched timestamp and the retrieved timestamp.

Each test has a load above 90 %. This means that more than 90 % of all created events in the test were processed. The reason for having a load on this level is that a load smaller than 90 % will give an unfair image of the response time, since many events will be queued up on the devices because they are created too fast (see Section 4.5.2). The way to achieve this was to modify the event frequency parameter ϕ so the gateway was at its peek load while still letting most events be processed. Each test run was run twice: first with a high ϕ value, meaning the load became smaller than 90 %, and second with $\phi = \frac{T}{q}$ where q is the amount of devices in the test and T is the throughput result of the previous test run. Most of the time, ϕ will get a proper value on the second run and the load will be above 90 %. However, due to high traffic in the network and other processes on the testing platform, the resulting load can be smaller than expected and a new test run must be executed.

6.1 Test case 1: Increasing device quantity

This test was conducted by running the SUT for 30 seconds 11 times, and for each run increase the number of devices by $1, 5, 10, 15, ..., 50$. Figure 6.1 shows clearly that the cooperative/preemptive gateway is by far faster than the rest for this given configuration of delay, CPU and I/O intensity (0, 0.1 and 20^{-4} respectively). Each test had the SUT under full load, which means the SUT could not deliver higher throughput for the given quantity. Due to the restriction that events from the same device must be handled serially, i.e. even though the dispatcher is able to pull more events from a device it cannot if a previous event from the device is still being processed, the throughput is generally lower for small quantity values. However the throughput stabilizes when the quantity reaches a certain level.

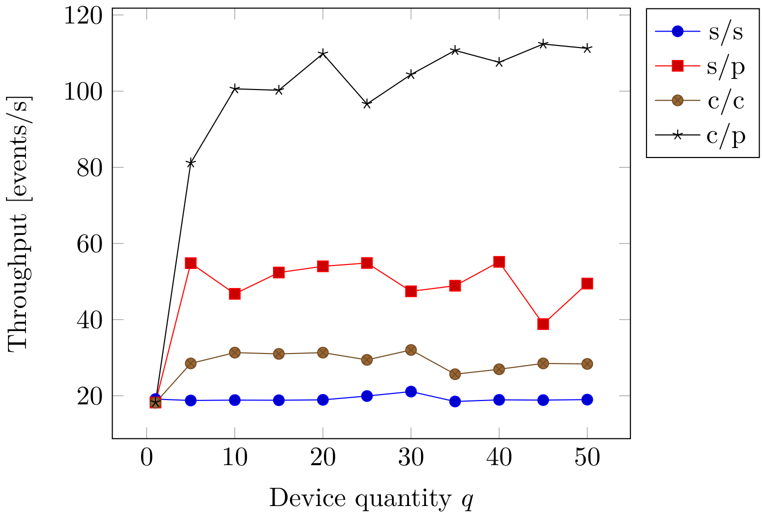

Figure 6.1: Throughput result of the four event propagation models on a quad core CPU as device quantity was increased.

The response time scales slowest for the preemptive event handlers, see Figure 6.2. There is a correlation between the throughput and the response time: the two models with highest throughput also have the lowest response time. The design of the dispatcher is not affecting the response time; both cooperative dispatch models give distinct different outcomes in response time, same for the serial. However, the cooperative/cooperative gateway scales linearly with the device quantity, see Figure 6.3, compared to the rest which seem unaffected by it. The reason is due to two things: the cooperative dispatcher will fetch events fast as soon they are available on each device and the single-threaded cooperative event handler will not be able to keep up with the incoming events, thus they must wait on the gateway. Comparing this to the serial event handler; each event is being processed directly as soon as it lands on the gateway because the serial dispatcher will not fetch any new event before the previous is processed. Therefore the time each event spends on the gateway is independent of the number of devices. The preemptive event handlers are able to keep the time on the gateway low, despite the dispatcher design. This is because they, similar to the serial event handler, will pause the dispatcher when the thread work queue is full. If the thread pool is configured to hold four threads, only four events are processed concurrently. The fifth event must wait for a thread to be free before being dispatched. This wait-task will pause the entire dispatcher, thus only as many events the gateway can handle concurrently are dispatched.

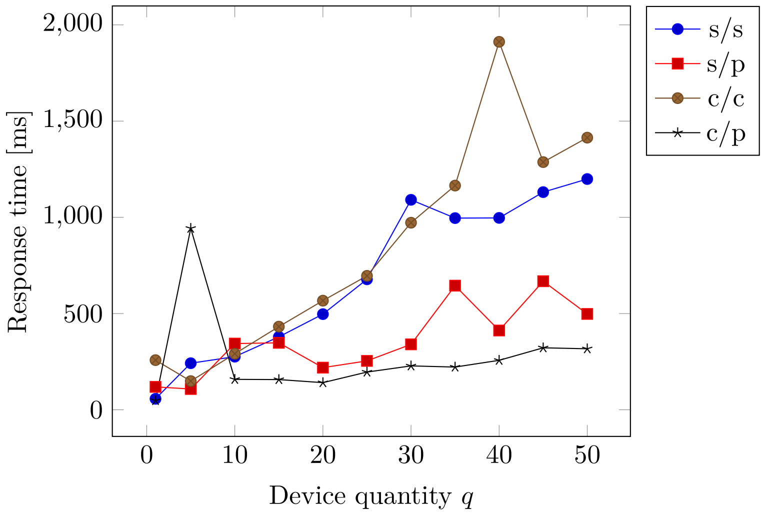

Figure 6.2: The average response time of the four event propagation models on a quad core CPU as device quantity was increased.

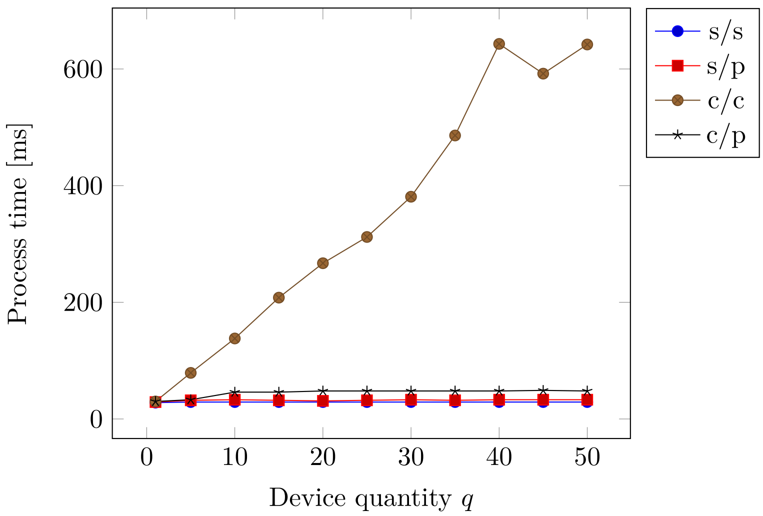

Figure 6.3: The average time each event spent on the gateway on a quad core CPU as device quantity was increased.

6.2 Test case 2: Increasing network delay

This test was conducted by running the SUT for 30 seconds 10 times, and each time the network delay was increased by 10 milliseconds. Only the quad core CPU was tested and the rest of the configuration was set to $q = 25$, $\lambda_0 = 0.1$ and $\lambda_1 = 2 \times 10^{-4}$. Figure 6.4 shows the throughput result of the tests and it clearly shows that the cooperative approaches perform much better as network delay is increased. There is a clear distinction between the cooperative and the serial dispatcher as the serial/preemptive gateway converge to the same throughput as the serial/serial gateway quite fast. The serial dispatcher is the main bottleneck, despite the event handler design. The cooperative dispatcher is able to concurrently send out TCP requests and it lets the network and the devices work while the event handler can take care of incoming events. Figure 6.5 shows how the response time depends on the dispatcher design, not the event handler.

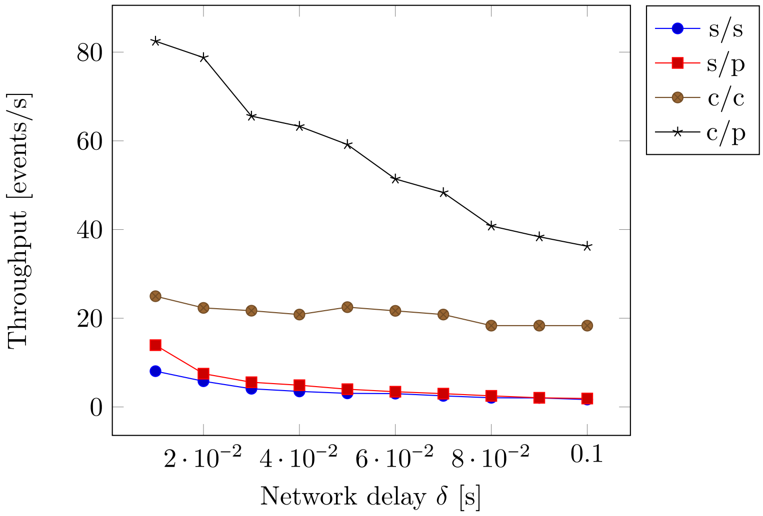

Figure 6.4: Throughput result of the four event propagation models on a quad core CPU as network delay was increased.

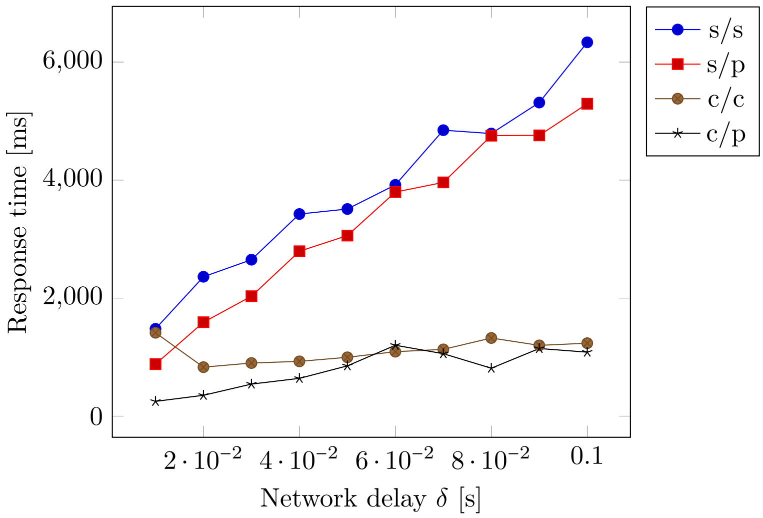

Figure 6.5: The average response time of the four event propagation models on a quad core CPU as network delay was increased.

6.3 Test case 3 and 4: Increasing CPU intensity

This test was conducted by running the SUT for 30 seconds 9 times, and each time the CPU intensity λ_0 was increased from 0.1 to 0.5 with steps of 0.05. Both quad and single core CPUs were tested and the rest of the configuration was set to $q = 25$, $\delta = 0$ and $\lambda_1 = 10^{-4}$. The throughput result in Figure 6.6 shows how the preemptive event handler performs much better than the cooperative and the serial on the quad core CPU. Both preemptive event handler designs converge as CPU intensity grows, however it is possible to see the serial dispatcher bottleneck in the serial/preemptive design when CPU intensity is low. The combination between a cooperative dispatcher and a preemptive event handler really shines in this case. For the single core CPU test, it is clear that the operating system cannot optimize CPU work despite using preemptive event handlers.

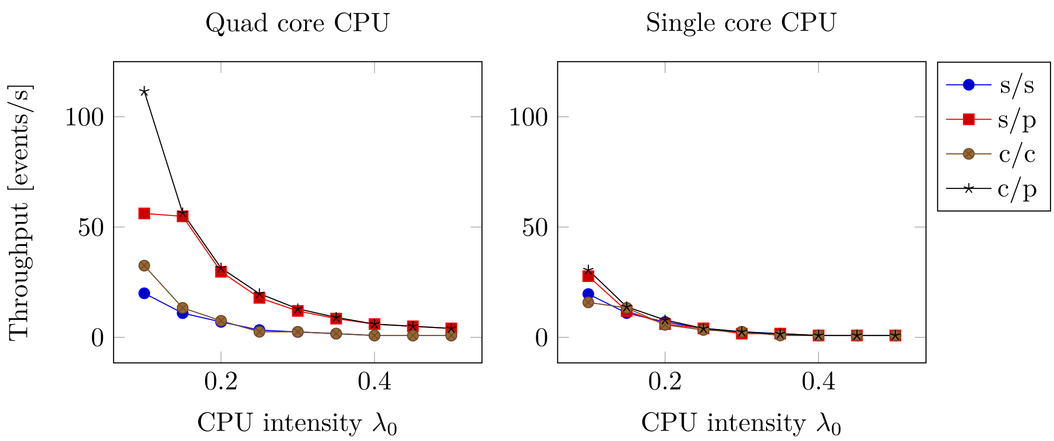

Figure 6.6: Throughput result of the four event propagation models as CPU intensity was increased.

Figure 6.7 shows how the response time for the cooperative/cooperative and serial/serial design is independent on number of cores. The cooperative/cooperative response time scales worse than serial/serial however. This can be explained by how the cooperative dispatcher works. It will fetch events as soon as they are available on the devices, leaving them queued up on the gateway. And as the cooperative event handler only runs on a single thread, as CPU intensity grows, so will the queue of events on the gateway. They will therefore wait significantly longer than the serial event handler. Figures 6.8 and 6.9 shows where the events are waiting. For the serial/serial design they spend longer time on the device than they do on the gateway. For the cooperative/cooperative design its the opposite: they spend longer time on the gateway than they do on the device.

6.4 Test case 5 and 6: Increasing I/O intensity

This test was conducted by running the SUT for 30 seconds 10 times, and each time the I/O intensity λ_1 was increased from 10^{-4} to 10^{-3} with steps of 10^{-4}. Both quad and single core CPUs were tested and the rest of the configuration was set to $q = 25$, $\delta = 0$ and $\lambda_0 = 0.05$. The preemptive and cooperative approaches perform much better than the serial/serial design and they all perform about the same, especially as I/O increases, despite the amount of cores, see Figure 6.10. Because the amount of work performed on each event by the event handler is mainly I/O oriented, there is no significant difference between the cooperative/cooperative and the cooperative/preemptive design. This is also why the number of cores does not affect throughput significantly. The main work is not done by the CPU, but by the filesystem. This also proves why the response in Figure 6.12 is not that different between the number of cores.

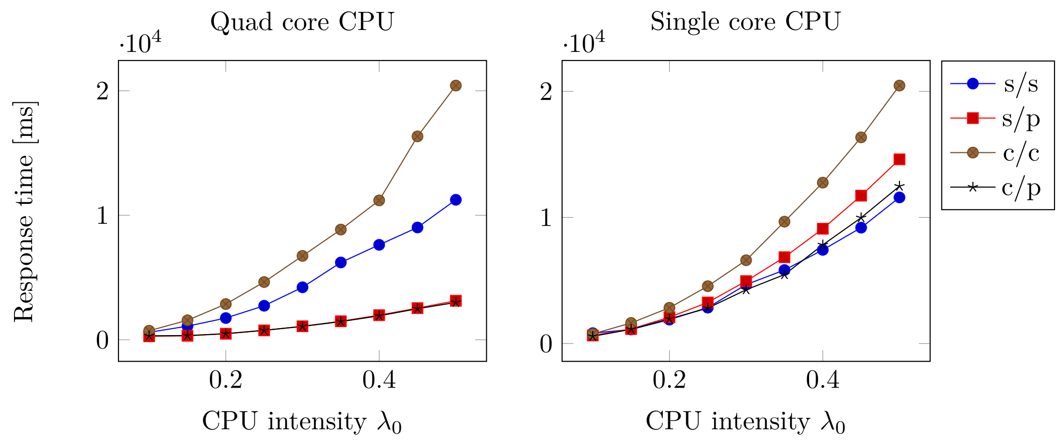

Figure 6.7: The average response time of each event expressed in milliseconds as CPU intensity increased.

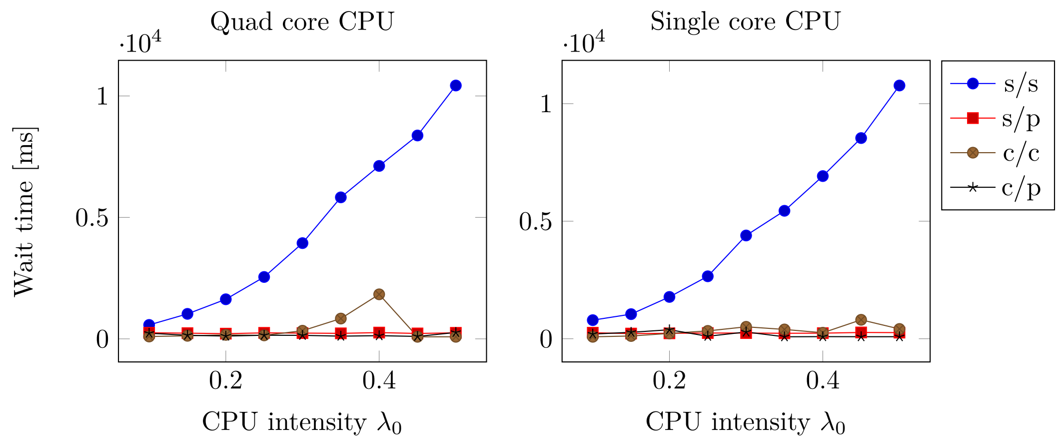

Figure 6.8: The average time each event spends on the device as CPU intensity increased.

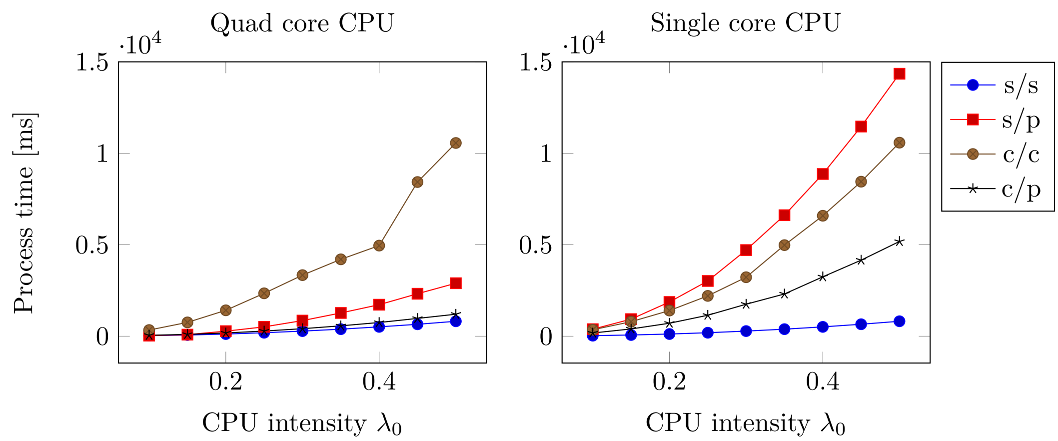

Figure 6.9: The average time each event spends on the gateway as CPU intensity increased.

There are a number of spikes in the response time measurements that correlate with the load in Figure 6.11. Load is the number of processed events divided by the number of created events. When the load decreases, there are events waiting to be processed either on the device or on the gateway. This increases response time and is the reason why the spikes correlate between the quad core CPU graph in Figure 6.12 and Figure 6.11.

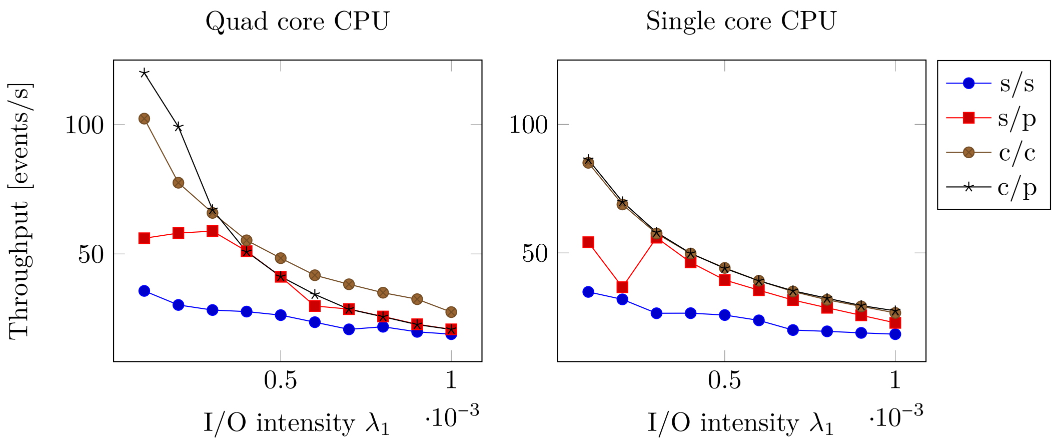

Figure 6.10: Throughput result of the four event propagation models as I/O intensity was increased.

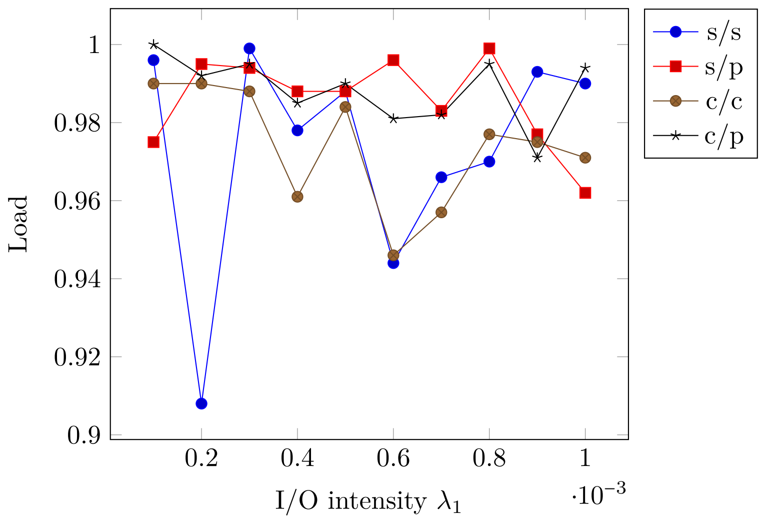

Figure 6.11: The load result as I/O intensity increased on a quad core CPU. This diagram shows the direct correlation low values has on response time, see the quad core chart in Figure 6.12.

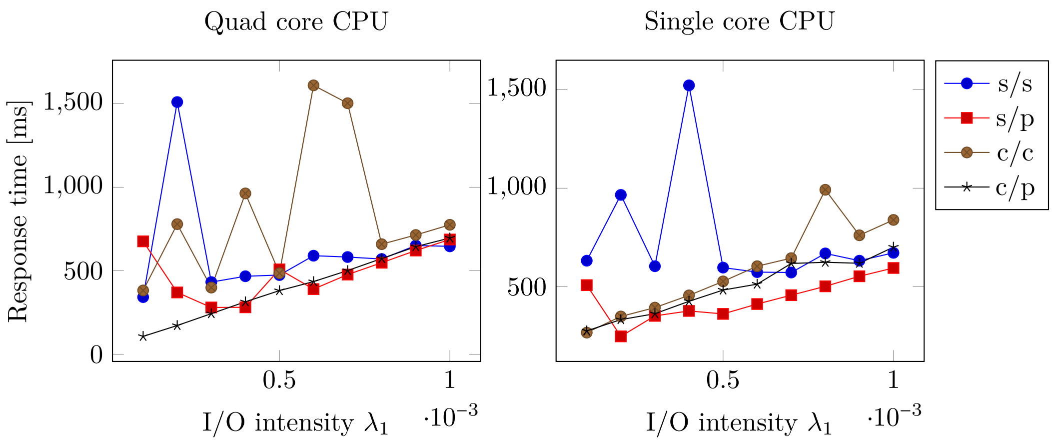

Figure 6.12: The average response time of each event expressed in milliseconds as I/O intensity increased.

7 Discussion

This chapter contains discussions on the result, the method and some elaboration on the work in a wider context.

7.1 Results

The results are satisfactory and provides insight into how different configurations of the gateway affect performance. There are however details in the production method of the data that could be improved. Each data point is the result of a single test scenario. Some diagrams have outliers caused by small delays and hiccups in the network and the operating system during the test. If every data point instead was the average value of several test scenarios with the same configuration, smoother diagrams would have been obtained.

Regarding the choice of data. There were a total of 18 diagrams that could have been presented in the result chapter. The most interesting ones were chosen and they were selected on the criterions:

1. They had unique outcomes. Only one of several diagrams that show the same result was chosen.
2. They were possible to explain within the time frame of the project.
3. They show significant difference between at least one of the event propagation models.

The choice of values for the parameters that varied in the tests required some amount of testing in order to get sensible results. Too small steps lead either to no significant change in performance or too many steps in order to see some change in performance. 10 steps took 2-3 hours to fully test all possible event propagation models. Many tests were redone to calibrate best possible step size and value window of the parameter. For instance, changing CPU intensity from 0.01 to 0.1 with 0.01 steps produced very small change in performance, compared to changing the window to 0.1-0.5.

7.2 Method

Some new ideas are proposed in this study: the three event propagation models and the abstract gateway. They have all been verified to work as expected *in this context*. There are limitations that have not been thoroughly tested yet, e.g. the push-based dispatching approach where devices actively push data to the gateway. There is currently no parameter to the abstract gateway that takes active/passive devices into account. This should be further developed.

Regarding the work Λ_j discussed in Section 4.4.3. The simplification that all events induce the same amount of work is not necessarily correct in practice. Wu et al. [2] discuss how large numbers of devices will be available on the market and if different types of devices are connected to the same gateway, it is obvious that they will induce different amount of work on it. Also, as noted in the same section, there can be dependency between CPU work and I/O work. The implementation done in this study has considered the CPU- and I/O work to be independent on each other.

7.3 The work in a wider context

Internet of Things is not only applicable for industries with economic gain in mind, but also for institutions like healthcare, education and public safety [16]. Improving the important gateway component of the IoT infrastructure and understanding its performance characteristics can help the development of IoT applications in those important areas of society.

8 Conclusion

To understand what internal and external properties affect the functionality and performance of the IoT gateway, a combination of observations of state-of-the-art technologies and consulting from experienced professionals led to a list of common parameters in most IoT gateway systems. These parameters were used to develop a theoretical model called the abstract gateway, which is proposed to model *any* type of gateway application. This model laid the foundation to a configurable gateway implementation and a performance test system. The test system was used to test the performance of different configurations of the gateway implementation. The internal and external properties of the gateway are:

1. The number of devices.
2. The frequency of generated events on each device.
3. The delay added to each request on the network.
4. The event propagation model of the dispatcher.
5. The event propagation model of the event handler.
6. The CPU intensity induced by each event.
7. The I/O intensity induced by each event.
8. The number of CPU cores on the gateway machine.

The abstract gateway has two major building blocks: the dispatcher and the event handler. The dispatcher handles incoming events from the devices communicating with the gateway and dispatch them forward to the event handler. The event handler performs some CPU- and I/O intensive work on each event. Three event propagation models are proposed that describe the fundamental architecture of the dispatcher and the event handler. The serial event propagation model communicates with devices and handles events serially, i.e. one device at a time. The preemptive model can communicate with several devices and handle several events concurrently on different threads and the operating system scheduler interleaves between the working threads. The definition of the cooperative model states that CPU and I/O work induced by each event can *explicitly* interleave and make room for other work. The cooperative model is implemented with libuv and allows multiple devices and events to be handled in parallel while still only running on one major thread.

Increasing the number of devices will increase the response time drastically for the serial and cooperative event handlers compared to the preemptive ones. The cooperative dispatcher gives much better response time when network delay is increased, however. For single core CPUs, all event propagation models perform equally in terms of throughput when CPU intensity is increased. The cooperative dispatcher combined with a cooperative event handler creates worst response time, despite the number of cores. For I/O intensive work, both the cooperative approaches perform best in terms of throughput.

The best approach to implement libuv in this context is to model the gateway as a state machine. As libuv uses callback functions to let the user know when asynchronous I/O work is done, the callbacks can be seen as vertices in the state machine. Whenever an asynchronous I/O request is initiated in libuv, the user can explicitly choose other tasks to run while the I/O request is working. This makes libuv a good choice to implement the cooperative dispatcher. The results show that the cooperative event handler is not ideal for CPU intensive events. libuv is primarily single-threaded; it is not the best alternative for implementing cooperative event handlers if the CPU intensity of events is high. However, utilizing the built-in threadpool in libuv one can build a preemptive event handler on top of a cooperative dispatcher. The results of the performance tests show that it performs better or equal to the next best event propagation model in all test cases.

8.1 Future work

The preemptive dispatcher was not developed and it would be interesting to do so in future work. The performance of a preemptive dispatcher would probably match the cooperative one, at least on multicore systems. This study only focused on passive devices that needed to be polled in order to retrieve their events. The alternative are active devices that push data to the gateway by themselves. This approach would be interesting to study as well as it resembles typical client-server architectures that are common to web solutions.

Bibliography

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

[2] Geng Wu, Shilpa Talwar, Kerstin Johnsson, Nageen Himayat, and Kevin D Johnson. “M2M: From mobile to embedded internet”. In: *IEEE Communications Magazine* 49.4 (2011).

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

[4] 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.

[5] 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.

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

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

[8] 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.

[9] 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.

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

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

[12] Thomas Zachariah, Noah Klugman, Bradford Campbell, Joshua Adkins, Neal Jackson, and Prabal Dutta. “The internet of things has a gateway problem”. In: *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications.* ACM. 2015, pp. 27–32.

[13] Carel P Kruger and Gerhard P Hancke. “Benchmarking Internet of things devices”. In: *Industrial Informatics (INDIN), 2014 12th IEEE International Conference on*. IEEE. 2014, pp. 611–616.

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

[15] Kevin Ashton. “That ‘internet of things’ thing”. In: *RFiD Journal* 22.7 (2011).

[16] 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.

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

[18] 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.

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

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.

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

[31] 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.

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

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

[34] Antonia Bertolino. "Software testing research: Achievements, challenges, dreams". In: *2007 Future of Software Engineering*. IEEE Computer Society. 2007, pp. 85–103.

[35] Pascal Raymond, Xavier Nicollin, Nicolas Halbwachs, and Daniel Weber. "Automatic testing of reactive systems". In: *Real-Time Systems Symposium, 1998. Proceedings. The 19th IEEE*. IEEE. 1998, pp. 200–209.

[36] Paul C Jorgensen. *Software testing: a craftsman's approach*. CRC press, 2016.

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

[38] Zhen Ming Jiang and Ahmed E Hassan. "A survey on load testing of large-scale software systems". In: *IEEE Transactions on Software Engineering* 41.11 (2015), pp. 1091–1118.

[39] Murray Woodside, Greg Franks, and Dorina C Petriu. "The future of software performance engineering". In: *2007 Future of Software Engineering*. IEEE Computer Society. 2007, pp. 171–187.

[40] Alberto Avritzer and Elaine J Weyuker. "Deriving workloads for performance testing". In: *Software: Practice and Experience* 26.6 (1996), pp. 613–633.

[41] Elaine J Weyuker and Filippos I Vokolos. "Experience with performance testing of software systems: issues, an approach, and case study". In: *IEEE transactions on software engineering* 26.12 (2000), pp. 1147–1156.

[42] Itay Mendelawy. *Understanding Your Reports: Part 1 - What are KPIs?* 2016. URL: `https://www.blazemeter.com/blog/load-testing-kpis-part-1-what-are-kpis?utm_source=Blog&utm_medium=BM_Blog&utm_campaign=performance-testing-load-testing-stress-testing-spike-testing-soak-testing` (visited on 02/19/2018).

[43] Scott Barber. "Creating effective load models for performance testing with incomplete empirical data". In: *Web Site Evolution, Sixth IEEE International Workshop on (WSE'04)*. IEEE. 2004, pp. 51–59.

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

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

[46] 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.

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

[48] Alexander M Novikov and Dmitry A Novikov. *Research methodology: From philosophy of science to research design*. Vol. 2. CRC Press, 2013.

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

[50] 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.

[51] 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.

[52] Giovanni Denaro, Andrea Polini, and Wolfgang Emmerich. "Performance testing of distributed component architectures". In: *Testing Commercial-off-the-Shelf Components and Systems*. Springer, 2005, pp. 293–314.

[53] Marcos K Aguilera, Jeffrey C Mogul, Janet L Wiener, Patrick Reynolds, and Athicha Muthitacharoen. "Performance debugging for distributed systems of black boxes". In: *ACM SIGOPS Operating Systems Review* 37.5 (2003), pp. 74–89.

[54] Yan Liu, Ian Gorton, Anna Liu, Ning Jiang, and Shiping Chen. "Designing a test suite for empirically-based middleware performance prediction". In: *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications*. Australian Computer Society, Inc. 2002, pp. 123–130.

[55] George F Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems: concepts and design*. pearson education, 2005.

[56] ZB Simpson. "A stream based time synchronization technique for networked computer games". In: *URL: http://www. mine-control. com/zack/timesync/timesync. html* (2004).