OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG
FACULTY OF COMPUTER SCIENCE
Communication and Networked Systems

Communication and Networked Systems

Bachelor Thesis

Extending Battery Life by Employing Fog Computing in CoAP

Mo Shen

Supervisor: Prof. Dr. rer. nat. Mesut Güneş
Assisting Supervisor: MSc. Marian Buschsueweke

Institute for Intelligent Cooperating Systems, Otto-von-Guericke-University Magdeburg

8. February 2019

Abstract

Abstract

Mobile devices are one of the most important components in the vision of Internet of Things (IoT). With the rapid development of wireless communication protocols, building a network consisting of a massive number of sensors is not a great challenge any more. The Constrained Application Protocol (CoAP) is one of the emerging protocols that focusses on the efficiency and reliability of Machine-to-Machine (M2M) communications. The CoAP Option Observe allows to subscribe to a resource instead of polling it constantly. Still, Observe lacks a flexible and standardised interface to specify which data is of interest. This often results in unnecessary high levels of network traffic and power consumption. Thus, more work can be done to optimize existing approaches or explore new possibilities. This thesis is motivated by the concept of fog computing and attempts to pursue a widely applicable method in this domain. This method aims to be energy efficient in dealing with long-term resource-monitoring tasks and focuses on extending battery life of embedded devices. As a result, an Application Programming Interface (API) is introduced and implemented. Two kinds of applications of this API are assumed and simulated in the experiments. With the help of this API, CoAP devices are able to process collected data at the data source. This reduces the necessary wireless communication and is effective to reduce power consumption. Additionally, for those computation-intensive tasks, this API gives mobile devices the ability to transfer part of work to remote servers via dynamic code migration. This method avoids a shortened battery lifetime caused by the sustained high CPU load. Through analyzing the obtained results of the two experiments, the amount of power that could be saved turns out to be significant. The response time of computation-intensive tasks could be reduced up to 10% through dynamic code migration.

Contents

List of Figures

List of Tables

Listings

Acronyms

API Application Programming Interface. iii, 3, 5, 6, 8, 10–12, 14, 15, 19–21

CoAP The Constrained Application Protocol. iii, 1–6, 8, 11, 12, 17, 19–21

HTTP Hypertext Transfer Protocol. 1

IoT Internet of Things. iii, 1–4, 10, 20, 21

M2M Machine-to-Machine. iii, 1, 3

WSN Wireless Sensor Network. 3

Glossary

DoS attack A denial-of-service attack (DoS attack) is a cyber-attack in which the perpetrator seeks to make a machine or network resource unavailable to its intended users by temporarily or indefinitely dsirupting services of a host connected to the Internet. 10

ESP8266 The ESP8266 is a low-cost Wi-Fi microchip with full TCP/IP stack and microcontroller capability produced by Shanghai-based Chinese manufacturer Espressif Systems. 5, 11

RIOT RIOT is a small operating system for networked, memory-constrained systems with a focus on low-power wireless Internet of Things (IoT) devices. 10

CHAPTER 1

Introduction

Internet of Things (IoT), first introduced by Ashton [1] in 1999, becomes nowadays one of the most exciting and promising research topics. According to Gubbi et al. [2], the number of interconnected devices on the planet has overtaken the actual number of people in 2011 and this number is expected to reach 24 billion by 2020. To achieve the complete IoT vision, an efficient, scalable and secure network is essential.

Thanks to recent technology advancements and rapid standardization processes in the domain of wireless communication, it is already possible to build a reliable, energy-efficient and low-cost sensor network. This network has the capabilities to collect, process and analyze valuable information in various environments. It brings a lot more possibilities to existing applications and changes the way we interact with mobile devices. However, which cannot be ignored is, alongside the countless opportunities and enormous potentials, there are also challenges and problems.

Wireless networks, on one hand, compared to wired networks, has its dominant advantage in mobility, cost efficiency and ease of deployment. Wireless devices can be accessed from anywhere, be deployed where a wire-network is hard to reach and work together effectively even in a frequently changing environment. On the other hand, wireless networks have the drawback of reduced reliability compared to wired networks, which is a result of interference, latency and reachability.

Besides, a limited power supply is another concern. Gubbi et al. [2] state that, most devices are based on battery supplies. As the wireless communication is the main contributor to the total power consumption [3, 4], there is an urgent requirement to improve the efficiency of wireless communication. With these characteristics, Hypertext Transfer Protocol (HTTP) is inapt due to its lack of multicast support, high overhead and complexity. New wireless communication protocols are needed to suit better into this situation. CoAP, as defined in RFC 7252 [5], is one of them. It is a specialized web transfer protocol for use with constrained nodes (with small amounts of ROM and RAM) and constrained networks (e.g., low-power, lossy) [5]. Additionally, it is designed for M2M applications such as smart energy and building automation, as the M2M communication without human interaction will be a great part of the wireless networking in the future. Furthermore, CoAP keeps the possibility of being easily translated into HTTP to simplify the integration with the web.

This encourages current developers to use this protocol and combine it with existing web applications.

Resource monitoring and data collecting is a common application in the domain of IoT, which requires a long-term and continuous wireless networking. To deal with this kind of scenario, CoAP introduces the Observe extension [6] to provide a more flexible and energy-efficient working procedure. However, Observe only provides a limited interface to filter which information to subscribe to. Additionally, this interface is largely unspecified leading to many incompatibilities between different implementations. Hence, a client will likely receive more data then needed.

This thesis is intended to provide a new approach to deal with this situation by employing fog computing paradigm in CoAP. With the new method, CoAP endpoints are able to execute code provided by other devices and make the runtime manageable as normal CoAP resources.

In the next section, the idea and motivation of energy-saving using this method are comprehensively explained and the possible application scenarios are also introduced.

1.1 Motivation

Although sensing devices in IoT are usually small and cheap, the work they do can be complicated and varies widely from application to application. It is a tough job to design a communication protocol that meets every requirements under all kinds of scenarios.

Monitoring a resource over a period of time is a common task in smart environment applications. Transferring every single record collected by sensors is unnecessary, since the raw data may be noisy or it is not required by other devices immediately. The energy is wasted due to these redundant network traffic. With the observe option, client is able to retrieve a resource and keep the status of this resource updated by the server over a period of time [6]. Additionally, Uri-Query can be used to filter out uninteresting records or process collected data before sending it. This is an effective way to reduce the network traffic and save some power for other work. But it is difficult to deal with complex scenarios, as the available options for Uri-Query are limited to what the server can provide and a combination of these options is not flexible enough to carry out all kinds of tasks.

Another problem that could occur in a IoT application is the sustained high CPU usage. The complexity of tasks executed on embedded devices varies from each other. A computation-intensive task increases the drain of battery on those devices and has a negative effect on the battery lifetime. If it is possible to actively transfer this kind tasks to remote servers, which are resource-rich and insensitive to energy cost, this problem can be therefore avoided. Moreover, due to the huge advantage in processing speed, the results may still arrive in time or earlier, even with the overhead of transferring work taken into account.

This thesis aims to provide a new energy-saving method to solve the above mentioned problems by bringing in the fog computing paradigm [7] into CoAP. The idea behind the fog computing is to extend the computing power, storage and memory capacity closer to data source, which allows these devices to precise filter or process data before transferring it.

The concept of the new approach is to integrate another runtime into current CoAP implementation. Other CoAP endpoints should be able to use this runtime and execute code remotely on these devices. The code is independent to CoAP server and has the full control of reading sensors and processing data collected. As a result, this approach reduces wireless communications similarly as Observe extension, while filtering procedure is not limited by the server. In the case of dealing with energy-consuming tasks, the device acts as a client and make the use of the runtime provided by other resource-rich devices.

1.2 Goal

The goal of this thesis is to specify an API that allows CoAP endpoints to execute code written in a certain format and return the result when the code successfully returns. At the same time, other devices are able to inspect the status of the execution and control the progress remotely. The code is intended to precisely process raw data at the data source, in order to reduce necessary wireless communication.

Furthermore, this API will be used to dynamically migrate code to other devices, so that CoAP devices can avoid doing energy-consuming tasks locally. The performance after implementing this API will be tested in terms of response time of a certain request. The method aims to keep the CPU load of CoAP endpoints at a low level to extend its battery life without delay the response of the task.

1.3 Thesis Structure

The rest of the thesis is organized as follows:

In the next section, related work will be reviewed. Chapter 2 consists of three parts: Concept, Implementation and Experiment. Chapter 2.1 shows the main idea of the API design and possible operations of each component on the implemented endpoints. In the Chapter 2.2, implantation details, tools and libraries used in C and Lua are presented. Problems and challenges are also carefully discussed. Chapter 2.3 demonstrates two possible application scenarios and evaluates the performance based on the data collected. Finally, in Chapter 3 the outcome of the thesis is summarized.

1.4 Related Work

In the domain of IoT, Wireless Sensor Network (WSN) or Fog Computing, a considerable amount of work has been done to improve the battery life of mobile devices. The efforts are made in many directions.

While Gubbi et al. [2] present their vision of IoT and its architectural elements with their future directions, Wu et al. [8] analyze the possible developing directions of M2M communications and discuss importance of standardization progress. In the Fog computing conceptual model written by Iorga et al. [7], how fog and mist computing is related to cloud-based computing models for IoT is explained. Furthermore, they pointed out important properties and aspects of fog computing to give a guidance for later researchers. All

these contributions together build a firm foundation for future developments and light a clear path for later researchers.

In the exploration of employing the concept of fog computing in IoT applications, previous researchers have proposed many innovative approaches. An architecture called "Smart Gateway" is introduced by Aazam and Huh [9] in 2014. It is able to analyze and process data before uploading it to cloud. This method is efficient in terms of reducing the upload and synchronization delay and is helpful in alleviating communication overhead and lessening the load of cloud. Shi, Chen, and Deters [10] try to uniform mobile devices as a whole device cloud via CoAP, in order to provide distributed computation and shared resources among these devices. So that all devices are connected to each other and are able to interact with other members to provide a better user-experience.

In terms of reducing power consumption, there are a great number of significant contributions. In 2007 Rahmati and Zhong [11] have already made their attempts to make mobile devices actively switch between Wi-Fi and cellular networks based on network condition estimation. They formulated the problem as a statistical decision problem and provide algorithms to make selections based on the field-collected data. As a result, they improved the average battery time by 35%.

Nearly at the same time, Pering et al. [12, 13] have done a series of experiments to improve the battery lifetime of mobile devices. In an earlier research, they proposed a system called "CoolSpots", which enables a wireless mobile device to automatically switch between multiple radio interfaces, such as Wi-Fi and Bluetooth. Later, in another work, they considered sharing the usage of wireless channels among multiple clients in the newly introduced "SwitchR" framework to further reduce the energy consumption in a multi-radio architecture.

Likely, Shi, Bahl, and Sinclair [14] suggested an event driven energy saving strategy for battery operated devices. The work mainly focused on reducing the power a device consumes in idle mode with the concept called "wake-on-wireless". This goal finally is achieved by adding a second, low-power channel to the device. Thus, it is possible to shut down other interfaces when the device is not being used.

In recent researches, Kientopf et al. [15] proposed a service management platform inspired by the concept of fog computing. With the new platform, the client application is able to send requests for a service migration. The introduced service manager will migrate the service based on the communication cost. This method effectively reduces the transmission latency and achieved a better scalability for IoT applications.

This work learns from the previous work and attempts to propose a solution of extending embedded devices' battery lifetime from the perspective of software design.

CHAPTER 2

Thesis Contribution

In this chapter, the idea of the API design and the implementation details are discussed. Furthermore, two experiments are conducted to show the proof of the improvements.

Chapter 2.1 reviews important design decisions taken and outlines the structure of the program. The usage of each components and their available operations are also briefly introduced. At the end of this section, an example is given to show the possible working procedure.

Chapter 2.2 covers the tools and libraries used to achieve the proposed objectives. According to the design of the API, a server should be able to support multiple runtimes at the same time. Different runtimes are included like modules. Each modules can work separately. However, in order to let the amount of work to fit in the scope of this thesis, the implementation considers Lua as the only supported runtime module.

Chapter 2.3 consists of two parts. Experiment 1 compares the number of wireless transmissions needed for a single task before and after the API is implemented. And then the improvements are evaluated with a computation by referencing the power consumption of ESP8266. Experiment 2 measures the response time of a request with and without using the dynamic code migration technique. This experiment aims to find out if it is possible to execute tasks remotely without adding extra delays to responses.

2.1 Concept

Because of the high power consumption of the wireless communication module, reducing its necessary network traffic is an effective way to extend the battery life of a mobile device. An existing approach in CoAP is the Observe extension defined in RFC 7641 [6]. By using this option in the request, this endpoint can be notified whenever the status of interested resource changes. However, it is hard to cover all application scenarios, since the amount of available options is limited. In some cases, a long-term observation may be required and the data will actually be processed when all of them are arrived, which means they can be collected and processed at data source and be transmitted back when the final result is computed. This significantly reduces the number of messages needed to complete this kind

URI	Content Format	Resource Type	Interface
/fog/lua/simple	test/plain; charset=utf-8	task_lua	fog_computing
/fog/lua/	test/plain; charset=utf-8	task_manager	fog_computing

Table 2.1: Example: Payload of a discovery response

of task. The goal of this work is to provide an API that fulfills this requirement.

The main component of a CoAP server are CoAP resources. These resources can be manipulated by four pre-defined methods: `GET`, `PUT`, `POST` and `DELETE`. Taking the compatibility into consideration, this API should be designed under the scope of CoAP, i.e. without defining new components or methods. So that the new functionality can be used by other devices without this API immediately.

As a result, there are only two components in this API: The Task Manager and Tasks. Both of them are actually CoAP resources. The Task Manager is static and pre-configured on the server. It is responsible for managing Task resources and is initially discoverable. A Task is dynamically managed by the Task Manager and other devices can communicate with it via standard CoAP requests.

The implemented API can be detected by reading the discovery response of a device. Task Manager has a resource type `"task_manager"` and interface `"fog_computing"`. If multiple Task Managers are available, the resource type should be defined more specific, e.g. `"task_manager_runtime1"` and `"task_manager_runtime2"`. Similarly, a Task has a resource type `"task"` and the same interface as the Task Manager. If multiple script formats are supported, the resource type of each Task should be defined as `"task_runtime1"` and `"task_runtime2"`. An example payload of the discovery response is shown in Table 2.1.

Operations on Task Manager

Available operations on the Task Manager are `GET` and `POST`. `GET` is used to provide human-readable information such as usage instructions or code examples. The information is static. The method `POST` is used to upload runnable code. The code that to be executed in the defined runtime should consist of two parts: a configuration and a function. The configuration part provides the necessary information for creating and executing the Task resource, while the function part contains the code to be executed. As shown in Table 2.2, method `PUT` has a similar property as POST with the only difference in property idempotent. Scripts can be included both in the payload of a PUT request and a POST request. However, due to different content of payloads, the resulting status of the endpoint is not strictly the same. This violates the property of idempotent, thus makes POST the only option.

Operations on Task

Tasks, unlike other regular CoAP resources, are created dynamically by Task Manager. Possible operations on a Task are defined as follows:

	GET	PUT	POST	DELETE
payload	no	yes	yes	no
safe	yes	no	no	no
idempotent	yes	yes	no	yes

Table 2.2: Properties of four CoAP methods

- `GET`: returns the status of the corresponding task which can be one of: `INIT` | `RUNNING` | `ERROR` | `FINISHED`
- `DELETE`: deletes the corresponding task
- `POST`: performs the command appended in the payload. Possible commands are:
  - `start`: starts the task if the task has status `INIT` or `FINISHED`
  - `cancel`: terminates the existing runtime if the task has status `RUNNING`
  - `result`: returns results if the task has status `FINISHED`

The method `GET` is chosen to retrieve the status of the runtime instead of the results of the code. The status of the runtime is an important information of the resource, as it limits the allowed operations on the Task resource. Blindly sending commands to a Task may cause an inefficient way of communicating. Moreover, the possible operations on Tasks can be further extended. Due to the different requirements in different applications, new commands and new statues can be easily added in a backward compatible manner to allow more fine grained control of the runtime.

Procedure

A standard procedure of creating and deleting a Task is displayed in Figure 2.1. During the creation of Task, the code appended in the payload is loaded in the runtime and the configuration part is evaluated to extract necessary information for creating the Task resource. If no error occurs and no duplicate Task found, the script will be saved to the local storage and the newly created Task resource is then discoverable. Otherwise a `4.00 Bad Request` should be returned with a diagnostic payload to indicate the reason of failure.

As is displayed in Figure 2.2a, upon receiving `"start"` command, the function part should be loaded and executed in a separate runtime and the status of Task changes to `RUNNING`. After the function finishes its job and successfully exits, the status of Task should be changed to `FINISHED`. And then, the results can be retrieved by other devices via `POST` requests. At last, the Task resource can also be restarted or deleted.

If the task is configured to accept a separate response (see Figure 2.2b), the server will start the task immediately after the resource is successfully created and send back an empty `ACK` message. When the executed function returns, the server sends the result back directly in a new message without waiting for a `POST` request from the client.

If any error occurs during the execution, its status should be changed to `ERROR` and no request other than `DELETE` should be further handled. Upon receiving `DELETE` request, the

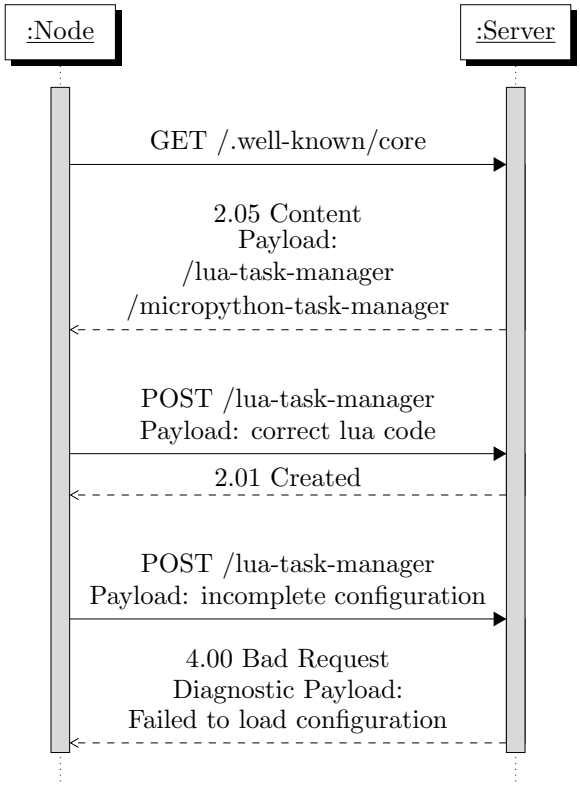


Figure 2.1: Sequence-Diagram: Creating a Task resource via Task Manager

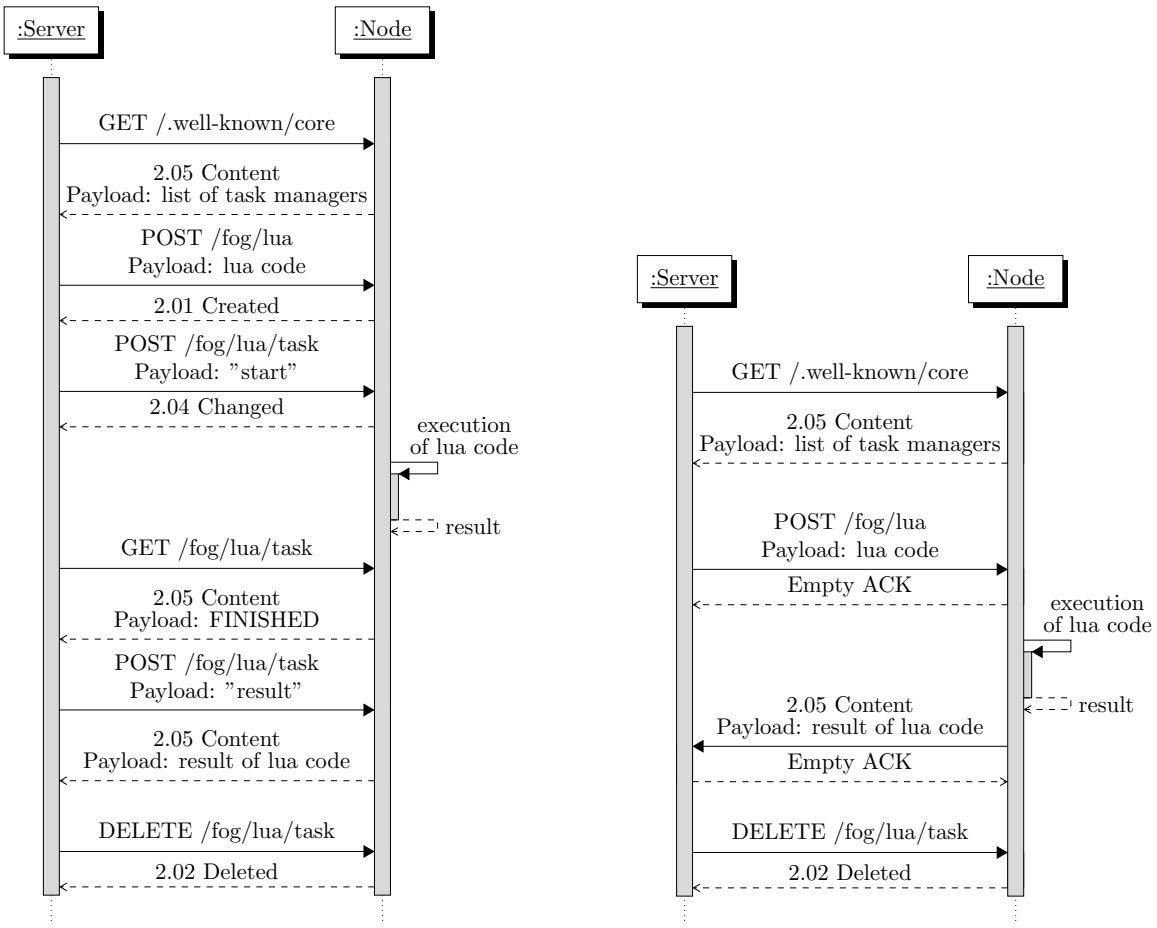
existing runtime should be stopped and all local files must also be cleared. Finally, the corresponding Task resource is no more visible on the server.

2.2 Implementation

This section details the implementation of the API and explains the additional libraries used to achieve the functionality of the separate response. Because the implementation of CoAP used in this thesis is written in C, further implementations of Task Manager and Task are also written in C. Although the runtime that can be integrated with the program is not fixed, due to the scope of the present work, only one is chosen and implemented. Among all available choices, Lua stands out for many reasons. Lua is designed to be a lightweight embeddable scripting language. It has a C API, which simplifies the work of integrating it into C program.

C API in Lua

One great advantage of Lua is its C API [16]. It can be included as libraries and let C code interact with Lua code in a separate environment called “state”. In the new environment, Lua manages a stack. This stack includes nearly all elements in a Lua program, and through this stack, values can be exchanged between C and Lua without worrying about



a) Task with a non-separate response

b) Task with a separate response

Figure 2.2: Sequence-Diagram: Running Tasks on remote server

type mismatch or potential memory leak. Moreover, Lua code can be executed in protected mode and all API functions are either safe or able to throw exceptions. These exceptions can be then handled in the C program. Hence, denial-of-service attacks (DoS attacks) on the server using the fog computing API are effectively prevented: The remotely executed Lua code will not be able to crash the C program in absence of bugs.

Pthread

Apart from building a new runtime in the C program, parallelism is another key in this API. It is common that a server must manage multiple Lua state at the same time, as in one Lua state only fits in one task. It would be a much less practical application, if a server can only do one job at a time. Meanwhile, the server must be able to handle other requests during the execution of these tasks. Threads and processes are both possible in this situation. However, threads have several advantages in this application compared to processes. The most important is the less overhead [17]. Creating new thread or terminating an existing thread consumes less time than doing so with a process. Switching between threads is also much faster than switching between processes. In this implementation, `pthread.h` [18] is used for multi-threading support. Pthread stands for POSIX Threads. `<pthread.h>` is an implementation adhered to the IEEE POSIX 1003.1c standard (1995). For the devices running a IoT operating system like RIOT, multiple threads are also supported by thread stacks and properly assigned priorities.

As a result, each Lua state will run in its own thread. In this thread, C code will load the Lua script, call the `run()` function in that script and wait until it returns. At the same time, the server holds the ID of the created thread, so that the whole execution can also be controlled from the outside.

Separate Response

As displayed in Figure 2.2, separate responses is a very practical functionality. It let the client get the result as soon as the task finishes, without sending extra query requests. Separate response is an already defined behavior in RFC7252 [5]. A server sends immediately an acknowledgment when receiving the request, and sends results back later in a new message.

There is already a event loop running on the server to watch over all incoming requests. Therefore, the server can be notified if there is a new I/O event. Pipe [19] is a common one-way communication method in UNIX operating system. The program can write on the one side and read on the other side. If the program finishes writing and close the write side of the pipe, an new I/O event will be raised. Thus it is used in this implementation as a notification system. `pipe()` is initialized when a task starts and the read side of the pipe is added to the event watch list, while the task thread holds the write side of the pipe. When the task finishes, the task thread writes to the pipe and exits, so that event loop will catch the read-ready signal and trigger the callback function to send the separate response.

```
conf = {
taskname = "a simple task",
separate = 1, autostart = 1
}
local script = {}
function script.run(): // do the work
return "a simple task done" end
return script
}
```

Listing 2.1: Lua Task Example

Example of a Lua task

In this section, a simple valid Lua script is displayed in Listing 2.1 to give a better understanding of this API. In global variable `conf` stores:

- `taskname`: to identify the resource under discovery
- `separate`: to check if the CoAP separate response should be used
- `autostart`: to check if the task should instantly start after the resource is successfully created.

The return value of the whole script must be a table including a `run()` function. Function `run()` should require no argument and its return value must be a string type. The returned string will be sent back as result in payload.

2.3 Experiments

This chapter describes two experiments conducted in this thesis. The goal of the experiments is to evaluate the potential improvements that this API can provide in the given two applications.

The first experiment aims to figure out how the wireless communication effects the total power consumption of a mobile device in a resource-monitoring task. The energy cost is computed based on the active time of each working mode of the device ESP8266, as well as its operating current. After a series of computation, three factors turn out to be significant: the number of messages, the length of each message and the operating current of the Wi-Fi module in TX mode.

In the second experiment, a simulation is performed to review the performance improvements by using the fog computing API to migrate code from the node to the cloud. The idea is to offload computation-intensive tasks from low power devices on to capable hardware.

2.3.1 Experiment 1: Analysis of Power Consumption

This Experiment assumes a common working environment of two CoAP devices. A client sends requests to a server periodically to collect sensor measurements. In this normal approach the server needs to send a response back for each measurement. However, not all data are valuable in the most of cases and it won't make any difference if the response comes immediately or not, since the collected information is often analyzed together.

With the API introduced in this thesis the client sends a piece of code to the server instead of a query request. The code is executed in the runtime provided by the server and returns the final result when it finishes. Hence, the server only needs to send one response back to deliver the final result.

The goal of this experiment is to analysis the power consumption of these two approaches.

Computation

In order to show the difference of power consumption in above mentioned two approaches more precisely, a computation is performed in this section.

Given a fixed period of time, the significant power consumption in this scenario consists of two parts: CPU work and wireless communication module, where the wireless communication can be further divided into two different types: working in Receive mode (RX) and working in Transmit mode (TX). The total power consumption of a device can be expressed as follows:

$$Q = T_{\mathrm{CPU}} \cdot I_{\mathrm{CPU}} + T_{\mathrm{RX}} \cdot I_{\mathrm{CPU+RX}} + T_{\mathrm{TX}} \cdot I_{\mathrm{CPU+TX}} \tag{2.1}$$

where

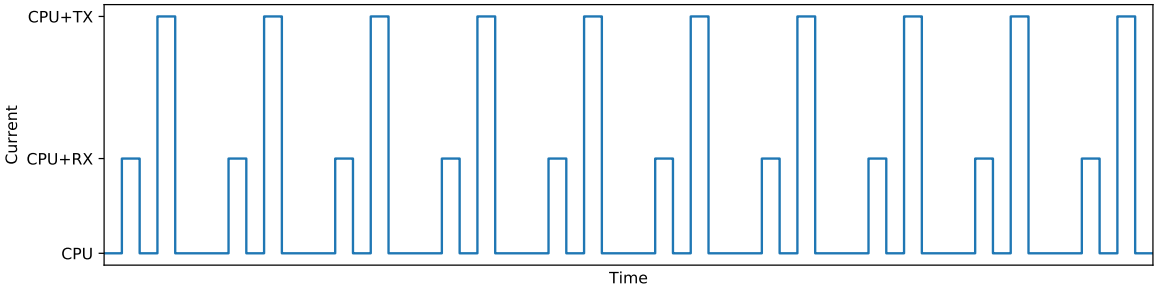
- Q is the total battery charge used
- T is the time elapsed in the corresponding power mode
- I is the working current of the corresponding power mode

The working mode of the device and the changes of its operating current during the task is shown in Figure 2.3. In the case a, the device switches to active mode frequently to send and receive data. In the case b, the device is able to work mostly in modem-sleep mode. In order to keep the availability, the device still needs to periodically switch to active mode to receive data. The additional charge used in the case b is the extra load on CPU due to the running code.

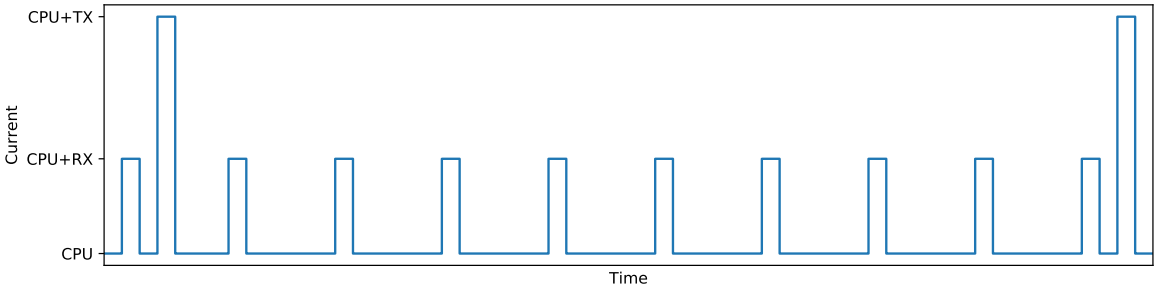
According to the equation 2.1, the charge required for both cases are displayed below:

$$\begin{aligned} Q_{\mathrm{a}} &= T_{\mathrm{CPU_a}} \cdot I_{\mathrm{CPU}} + T_{\mathrm{RX_a}} \cdot I_{\mathrm{CPU+RX}} + T_{\mathrm{TX_a}} \cdot I_{\mathrm{CPU+TX}} \\ Q_{\mathrm{b}} &= T_{\mathrm{CPU_b}} \cdot I_{\mathrm{CPU}} + T_{\mathrm{RX_b}} \cdot I_{\mathrm{CPU+RX}} + T_{\mathrm{TX_b}} \cdot I_{\mathrm{CPU+TX}} + T_{\mathrm{Task}} \cdot I_{\mathrm{Task}} \end{aligned} \tag{2.2}$$

Besides, the total time elapsed during this task is the sum of the time spent in all these three modes. The working current for RX/TX mode equals the current needed for the Wi-Fi module plus the necessary current for the CPU. Also, the time spent in RX mode



a) Task with a non-separate response



b) Task with a separate response

Figure 2.3: Current-Time-Diagram of monitoring a CoAP resource

for both cases are the same. As a result, there are the following relations between these variables.

$$\begin{aligned} T_{\text{total}} &= T_{\text{CPU}} + T_{\text{RX}} + T_{\text{TX}} \\ I_{\text{CPU+RX}} &= I_{\text{CPU}} + I_{\text{RX}} \\ I_{\text{CPU+TX}} &= I_{\text{CPU}} + I_{\text{TX}} \\ T_{\text{RX}_\text{a}} &= T_{\text{TX}_\text{b}} \end{aligned} \tag{2.3}$$

$$\begin{aligned} \Delta Q = Q_\text{a} - Q_\text{b} = (T_{\text{CPU}_\text{a}} - T_{\text{CPU}_\text{b}}) \cdot I_{\text{CPU}} \\ + (T_{\text{RX}_\text{a}} - T_{\text{RX}_\text{b}}) \cdot I_{\text{CPU+RX}} + (T_{\text{TX}_\text{a}} - T_{\text{TX}_\text{b}}) \cdot I_{\text{CPU+TX}} - \underbrace{T_{\text{Task}} \cdot I_{\text{Task}}}_{=Q_{\text{Task}}} \end{aligned}$$

As T_{Task} and I_{Task} are independent to the rest parts of the equation, the product can be simplified to Q_{Task} to indicate the additional power consumed by the running of the code. By combining the equation 2.2 with the equation 2.3, the difference of charge turns out to be:

$$\begin{aligned} \Delta Q = ((T_{\text{total}} - T_{\text{RX}_\text{a}} - T_{\text{TX}_\text{a}}) - (T_{\text{total}} - T_{\text{RX}_\text{b}} - T_{\text{TX}_\text{b}})) \cdot I_{\text{CPU}} \\ + (T_{\text{TX}_\text{a}} - T_{\text{TX}_\text{b}}) \cdot I_{\text{CPU+TX}} - Q_{\text{Task}} \end{aligned}$$

$$\begin{aligned}\Delta Q &= (T_{\mathrm{TX_b}} - T_{\mathrm{TX_a}}) \cdot I_{\mathrm{CPU}} + (T_{\mathrm{TX_a}} - T_{\mathrm{TX_b}}) \cdot I_{\mathrm{CPU+TX}} - Q_{\mathrm{Task}} \\ &= (T_{\mathrm{TX_a}} - T_{\mathrm{TX_b}}) \cdot (I_{\mathrm{CPU+TX}} - I_{\mathrm{CPU}}) - Q_{\mathrm{Task}} \\ &= (T_{\mathrm{TX_a}} - T_{\mathrm{TX_b}}) \cdot I_{\mathrm{TX}} - Q_{\mathrm{Task}}\end{aligned} \tag{2.4}$$

As T_{TX} can be expressed as the product of the number of packets N_{pkt} and the time needed to transfer a single packet T_{pkt}. So in case b, there are

$$T_{\mathrm{TX}} = N_{\mathrm{pkt}} \cdot T_{\mathrm{pkt}} \tag{2.5}$$

In the case a, the number of packets can be defined as N and the time needed for a normal response is T_{npkt}. In the second case, the required packets are constantly 2 and the transmission time of the result can be represented as T_{rpkt}. As a result, the final expression of the charge difference is:

$$\Delta Q = (N \cdot T_{\mathrm{npkt}} - 2 \cdot T_{\mathrm{rpkt}}) \cdot I_{\mathrm{TX}} - Q_{\mathrm{Task}} \tag{2.6}$$

Results

With equation 2.6 the charge difference for a certain task can be analyzed. The final influencing factors are:

- N: the number of messages needed to complete the task
- T_{npkt}: the transmission time for a normal packet
- T_{rpkt}: the transmission time for a result packet
- I_{TX}: the operating current of the wireless module in TX mode
- Q_{Task}: the additional charge required to run the code

The detailed evaluation will be performed in Chapter 3.1.

2.3.2 Experiment 2: Simulation of Computation-Intensive Scenario

This experiment is conducted to analyze the performance of this API in computation-intensive scenarios. One of the goals of this API is to allow mobile devices avoid doing energy-consuming tasks and make the use of the runtime provided by other resource-rich devices.

One important criterion of the performance is the response time. The expectation of this experiment is to determine the overhead of the code migrating process and compare it with the accelerated data-processing phase. As a result, this could be used as a reference in practical applications to decide if it is appropriate to execute a certain task remotely.

Environment Setup

It is assumed that there are two devices. One acts as a resource-constrained device with limited battery power. The other is a resource-rich cloud-based server. In this scenario, the pending task is expected to consume a large amount of power. Furthermore, the response time of this task is critical. The delay of the response is used a measurement of performance.

As Figure 2.4 illustrates, the communications between the client and the mobile device are the same. The difference appears during the data-processing phase. It is expected that the method with the API should not consume more time than the normal one. To find the turning point, a task with a variable complexity is required.

In this simulation, Lua function `cal_pi(n)` is used (see Appendix A.1). This function computes digits of pi with a for-loop for a given number of iterations. The higher iteration number is given, the more valid digits it produces. Although the output is irrelevant in this experiment, the variable execution time could be used to represent different levels of complexity. The number of iterations is set to 30, 100, 300, and 900.

Hardware

Two devices are used in the experiment.

A laptop is selected to act as the cloud-based server and a Raspberry Pi is selected to act the mobile device deployed near sensors. Technical specifications of the two devices are displayed as follows:

The laptop is a ThinkPad X1 Carbon (6th Gen) with:

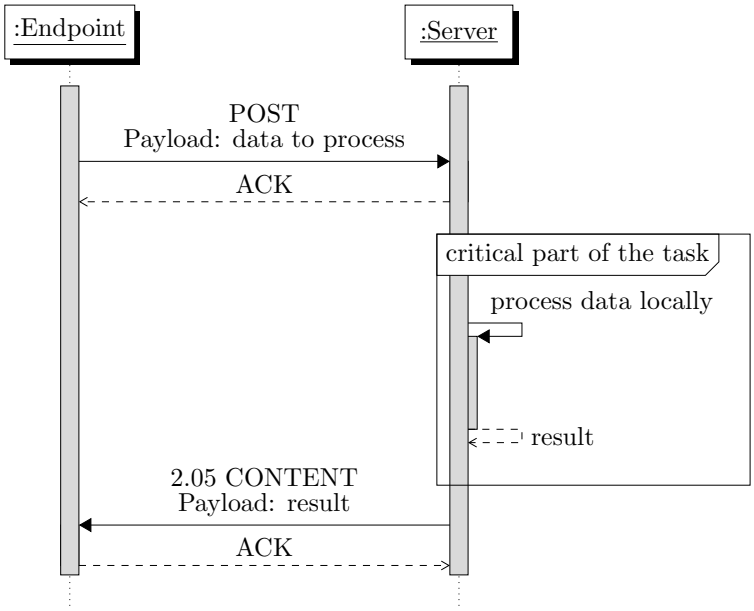
- OS: Arch Linux x86_64
- Processor Type: Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
- RAM: 8GB LPDDR3
- WLAN: Intel Dual Band Wireless-AC 8265, WiFi 2x2 802.11ac

The Raspberry Pi is a Raspberry Pi 3 Model B+ with:

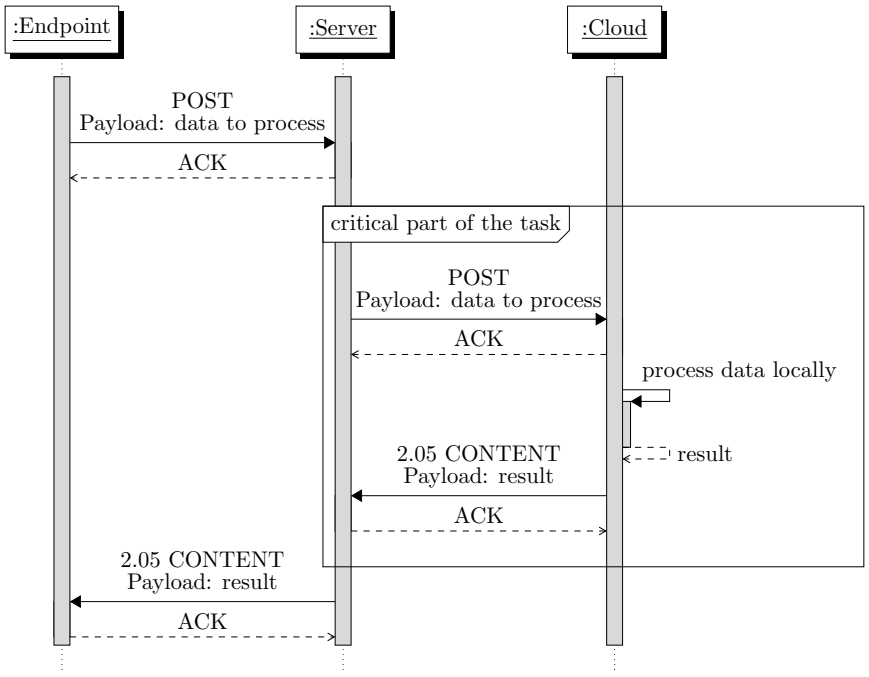
- OS: Debian Linux armv7l
- Processor Type: Broadcom BCM2837B0, Cortex-A53 (ARMv8) 64-bit SoC @ 1.4GHz
- RAM: 1GB LPDDR2
- WLAN: 2.4GHz and 5GHz IEEE 802.11.b/g/n/ac wireless LAN

Execution

In the first run, the goal is to measure the required time of the critical part in Figure 2.4a. A Lua script (see Appendix A.1) is used to load and execute the `cal_pi(n)` function. The time elapsed during the execution of the function is measured in nanosecond with the help of the Lua module `chronos`[20]. The script is executed on the Raspberry Pi 100 times for each level of complexity.



a) Processing data locally



b) Processing data remotely

Figure 2.4: Sequence-Diagram: Processing data on the server

Number of Loops	Location	Response Time [s]		
		Avg	Min	Max
30	local	0.013	0.011	0.023
	remote	0.022	0.014	0.077
100	local	0.073	0.066	0.141
	remote	0.036	0.023	0.100
300	local	0.571	0.547	0.614
	remote	0.096	0.083	0.161
900	local	5.054	4.485	5.313
	remote	0.571	0.516	0.614

Table 2.3: Response time of locally and remotely executed tasks

In the second run, to measure is the time needed for the critical part displayed in Figure 2.4b. A CoAP server is deployed on the laptop and the CoAP client on the Raspberry Pi sends a task-creating request to the server with the `cal_pi(n)` as the `run()` function. Also, the `separate` option in the configuration of the Task is enabled. The time required in this method is measured from the creating of the Task resource until the arrival of the separate response from the server. The program used to measure the time for this process is `coap_bench`. It is included in the CoAP implementation used in this thesis and also measures the time in nanosecond. After receiving the result, the client sends a `DELETE` request to delete the Task resource. Same as in the first run, these two steps are repeatedly performed 100 times for each level of complexity.

Results

Figure 2.5 shows a boxplot of response time for all tasks in four levels of complexity. Table 2.3 represents the average response time along with the maximum and minimum among all executions. The unit of all measurements is converted to second for a better readability. The tasks with same number of iterations are grouped in the same figure for the comparison. Abnormal measurements are marked out as outliers and will be ruled out in the final evaluation.

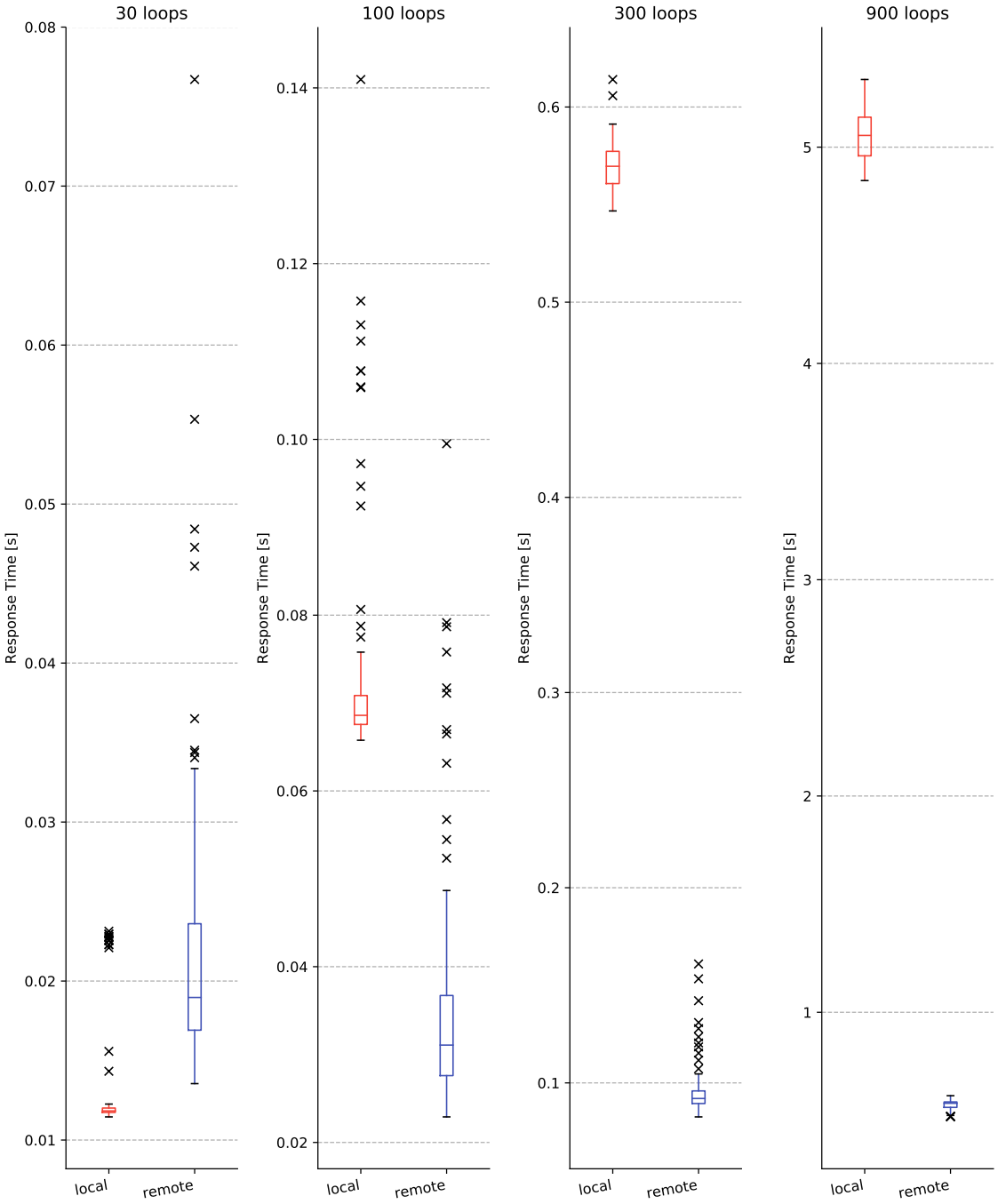


Figure 2.5: Boxplot: Response time of locally and remotely executed tasks

CHAPTER 3

Thesis Outcome

The main goals of the proposed API for code migrations are the reduction of power consumption and latency of low power servers. In this chapter the results of the experiments in Section 2.3 are evaluated in regard to these goals. Finally, a conclusion is drawn on how the concept of fog computing could be integrated into current CoAP applications. It states the improvements this API could provide, as well as the shortcomings it has. The last part of this thesis gives suggestions for future work.

3.1 Evaluation

3.1.1 Evaluation of Experiment 1

As shown in equation 2.6, there are several factors that determine the charge difference for a certain task: N, T_{npkt}, T_{rpkt}, I_{TX}, Q_{Task}.

First of all, if omitting the different length of the packets, the number of packets N needs to be greater than 2 in order to result in a positive ΔQ. It means that this API is more applicable in long-term tasks with frequent wireless communications.

Secondly, taking the length of the packets into account, this API could have its advantages with a small number N: If the code is able to reduce the length of the payload, so that $T_{\text{npkt}} \gg T_{\text{rpkt}}$. ΔQ could turn out to be a considerable amount of charge.

Furthermore, using different standards for the wireless transmission will also have an influence on T_{pkt} and I_{TX}. Transmissions using older versions of the IEEE 802.11 standard take longer, resulting in an increased T_{pkt}. Additionally the operating current for older versions of IEEE 802.11 is typically higher (see Table 3.1). As the new method limits the required packets to 2, it is more likely to achieve a longer battery life in such situations.

The last part of the equation Q_{Task} has a negative effect on the amount of power that could be be saved. This factor indicates that the uploaded code should avoid doing a computation-intensive task.

Power Mode	Description	Power Consumption
active	Wi-Fi TX 802.11b/g/n	170/140/120 mA
active	Wi-Fi RX 802.11b/g/n	50/ 56/ 56 mA
moderm-sleep	CPU is working	15 mA

Table 3.1: ESP8266: Required current by power mode [21]

3.1.2 Evaluation of Experiment 2

As illustrated in the Figure 2.5, the new approach outperforms the normal one in most of the cases. Processing data locally only has advantages if the process time is less than 0.05 seconds. Given a good network environment, the overhead of the code migrating is small, while the difference of computing power between cloud-based servers and portable devices is significant. In the case with 100 loops, the average response time for the locally executed tasks is already double of the remotely executed ones. This factor rises to 10, when the number of loops goes to 900.

Besides the reduced response time, the IoT node can enter low power mode after the successful creation of the task. If there are other incoming requests, the device is able to handle them immediately instead of being occupied by previous tasks.

It is noticeable that there are more outliers in the first two locally executed tasks, compared to the latter two with 300 and 900 loops. It can be caused due to the too short execution time of the first two tasks. If the scheduler preempts the process to allow the execution of other tasks, the cost of switching between processes may play a significant role here. The impact becomes smaller, as the time needed for the task grows.

In the case of remotely executed tasks, the number of outliers remains at a constant level for the first three kinds of tasks, while then number approaches zero in the last case. This phenomenon is likely to be caused by the congested Wi-Fi channel. The attempts of sending data and switching between channels add additional delay to the response.

3.2 Conclusion

This project is motivated by the consistent demand on longer battery life for mobile devices. In CoAP applications, the Observe extension has provided a data-filtering method to reduce the necessary wireless communication between mobile devices. However, there is still room for improvement in regard to energy-saving. Especially when dealing with resource-monitoring or computation-intensive tasks, CoAP devices are imperfect due to a lack of available methods.

The goal of this thesis is to provide a new approach for CoAP devices to improve the energy-efficiency when handling above mentioned tasks. An API is proposed to solve this problem. It enables CoAP servers to process scripts uploaded by other devices. These scripts are executed in a separate thread and has full control over how the raw data is processed. Compared to the Observe option, this method is more flexible in the capability of data-filtering. Moreover, it is able to support different runtimes according to the hardware

selected in CoAP applications. Finally, by making the use of separate responses, the client can get the result as soon as the script finishes.

The performance of the API is evaluated via two experiments.

In the first experiment, a computation is performed to show the difference of charge cost between the approaches with and without the introduced API. After analyzing the final expression, it turns out that the API has its advantages if the task requires frequent wireless communications. Additionally, reduced power consumption can also be achieved if the payload of the response can be compressed by the code uploaded. During the application of this API, extra attention should be paid to avoid executing computation-intensive tasks. As the required power of running the migrated code could be higher than the power required for transmission.

In the second experiment, the focus is on running computation-intensive tasks with mobile CoAP devices. Considering the lack of computing power on these devices, it would be unwise to executed this kind of tasks locally. In order to measure the overhead of code migration using this API, a laptop and a Raspberry Pi are used to run a set of simulations. During the simulations, the response time of the requests are measured as the reference of performance. By comparing the response time, it can be confirmed that the overhead of code transmission is smaller than the power required to completely process the data in most of cases. Besides, mobile devices can enter low power mode, if there are no other incoming requests to handle.

In conclusion, this API has its advantages in certain kinds of tasks: resource-monitoring and computation-intensive tasks. With the help of the first experiment, it is recognized that the number of wireless communications is a significant influence factor to the power consumption. Processing data at the data source is a feasible approach in terms of energy-saving, even if it comes at the cost of increased CPU time. It can also been concluded that computation-intensive tasks are better handled remotely with cloud-based servers, as the difference of computing power between IoT nodes and servers is significant. The response is more likely to come earlier, although the offloading approach requires additional work due to the code transmission phase.

In regard to the uncovered area of this thesis, problems are mainly caused by the lack of practical experiments. In this thesis, the experiments did not take network conditions into account. In practical applications, CoAP devices usually work in constrained environments with an unstable network connection. Different network conditions will result in different overhead of the code migration processes. This API could work in a more efficient way if it is optimized based on the data collected from real-world applications. Finally, the compatibility of this API with other existing approaches is not validated. Considering the versatility and complexity of smart environments in the domain of IoT, combing multiple approaches would be an effective way to achieve a better result.

Future Work

Due to the scope of this thesis, the proposed API is only tested on very limited hardware. Additionally, the implementation supports only one runtime. Adding more runtimes to various platforms is necessary to find out the most efficient one. Furthermore, different

standards of wireless communication may also be an influence factor. More experiments should be conducted to provide more precise optimizations. In order to combining existing methods together, the categorization of tasks is an indispensable step. Effectively recognize the type of the task and choose the right method to handle it is the key to maximize the battery life of mobile devices.

Bibliography

[1] Kevin Ashton. That 'internet of things' thing. `https://www.rfidjournal.com/articles/view?4986`, 2009. Accessed: 2019-01-03.

[2] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future generation computer systems*, 29(7):1645–1660, 2013.

[3] T. Pering, V. Raghunathan, and R. Want. Exploiting radio hierarchies for power-efficient wireless device discovery and connection setup. In *18th International Conference on VLSI Design held jointly with 4th International Conference on Embedded Systems Design*. IEEE Computer Soc.

[4] Yuvraj Agarwal, Curt Schurgers, and Rajesh Gupta. Dynamic power management using on demand paging for networked embedded systems. In *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, ASP-DAC '05, pages 755–759, New York, NY, USA, 2005. ACM.

[5] C Bormann, K Hartke, and Z Shelby. The constrained application protocol (coap). *RFC 7252*, 2015.

[6] K Hartke. Rfc 7641: Observing resources in the constrained application protocol, 2015.

[7] Michaela Iorga, Larry Feldman, Robert Barton, Michael J Martin, Ned Goren, and Charif Mahmoudi. Fog computing conceptual model. Technical report, mar 2018.

[8] Geng Wu, Shilpa Talwar, Kerstin Johnsson, Nageen Himayat, and Kevin D Johnson. M2m: From mobile to embedded internet. *IEEE Communications Magazine*, 49(4), 2011.

[9] Mohammad Aazam and Eui-Nam Huh. Fog computing and smart gateway based communication for cloud of things. In *Future Internet of Things and Cloud (FiCloud), 2014 International Conference on*, pages 464–470. IEEE, 2014.

[10] Heng Shi, Nan Chen, and Ralph Deters. Combining mobile and fog computing: Using CoAP to link mobile device clouds with fog computing. In *2015 IEEE International Conference on Data Science and Data Intensive Systems*. IEEE, dec 2015.

[11] Ahmad Rahmati and Lin Zhong. Context-for-wireless: Context-sensitive energy-efficient wireless data transfer. In *Proceedings of the 5th International Conference on Mobile Systems, Applications and Services*, MobiSys '07, pages 165–178, New York, NY, USA, 2007. ACM.

[12] Trevor Pering, Yuvraj Agarwal, Rajesh Gupta, and Roy Want. Coolspots: reducing

the power consumption of wireless mobile devices with multiple radio interfaces. In *Proceedings of the 4th international conference on Mobile systems, applications and services*, pages 220–232. ACM, 2006.

[13] Yuvraj Agarwal, Trevor Pering, Roy Want, and Rajesh Gupta. Switchr: Reducing system power consumption in a multi-client, multi-radio environment. In *Wearable Computers, 2008. ISWC 2008. 12th IEEE International Symposium on*, pages 99–102. IEEE, 2008.

[14] Eugene Shih, Paramvir Bahl, and Michael J. Sinclair. Wake on wireless: An event driven energy saving strategy for battery operated devices. In *Proceedings of the 8th Annual International Conference on Mobile Computing and Networking*, MobiCom '02, pages 160–171, New York, NY, USA, 2002. ACM.

[15] Kai Kientopf, Saleem Raza, Simon Lansing, and Mesut Güneş. Service management platform to support service migrations for iot smart city applications. In *Personal, Indoor, and Mobile Radio Communications (PIMRC), 2017 IEEE 28th Annual International Symposium on*, pages 1–5. IEEE, 2017.

[16] Roberto Ierusalimschy. Programming in lua. `https://www.lua.org/pil/24.html`, 2009. Accessed: 2019-01-21.

[17] Daniel Robbins. Posix threads explained. `https://www.ibm.com/developerworks/library/l-posix1/index.html`, 2000. Accessed: 2019-01-23.

[18] IEEE and The Open Group. <pthread.h>. `http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/pthread.h.html`, 2018. Accessed: 2019-01-15.

[19] Michael Kerrisk. Pipe(2). `http://man7.org/linux/man-pages/man2/pipe.2.html`, 2018. Accessed: 2019-01-21.

[20] ldrumm. chronos. `https://github.com/ldrumm/chronos`, 2018. Accessed: 2019-01-21.

[21] Espressif Systems. Esp8266ex datasheet. `https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf`, 2018. Accessed: 2019-03-16.

Appendix

A.1 Example Scripts used in Simulation

```
conf = {taskname = "pi", separate=1}
return {
    run = function()
        a = {}
        n = 300
    len = math.modf( 10 * n / 3 )
    for j = 1, len do
        a[j] = 2
    end
    nines = 0
    predigit = 0
    for j = 1, n do
        q = 0
        for i = len, 1, -1 do
            x = 10 * a[i] + q * i
            a[i] = math.fmod( x, 2 * i - 1 )
            q = math.modf( x / ( 2 * i - 1 ) )
        end
        a[1] = math.fmod( q, 10 )
        q = math.modf( q / 10 )
        if q == 9 then
            nines = nines + 1
        else
            if q == 10 then
                -- io.write( predigit + 1 )
                for k = 1, nines do
                    -- io.write(0)
                end
                predigit = 0
                nines = 0
            else
                -- io.write( predigit )
                predigit = q
                if nines ~= 0 then
                    for k = 1, nines do
                        -- io.write( 9 )
                    end
                    nines = 0
                end
            end
        end
    end
    -- print( predigit )
    return "Task: pi done!"
end
}
```

```
chronos = require("chronos")
print("# Benchmark_cal_pi")
print("#id,     time [ns]")
for i = 0,99 do
    start = chronos.nanotime()
    os.execute("lua -e \"dofile('cal_pi.lua'):run()\"")
    stop = chronos.nanotime()
  time = ("%s"):format((stop - start)*1e9)
  s, e = string.find(time, '.', 1, true)
  print(i .. ',     ' .. string.sub(time, 1, s-1))
end
```

I herewith assure that I wrote the present thesis titled *Extending Battery Life by Employing Fog Computing in CoAP* independently, that the thesis has not been partially or fully submitted as graded academic work and that I have used no other means than the ones indicated. I have indicated all parts of the work in which sources are used according to their wording or to their meaning.

I am aware of the fact that violations of copyright can lead to injunctive relief and claims for damages of the author as well as a pen alty by the law enforcement agency.

Magdeburg, March 18, 2019

_____

(Mo Shen)