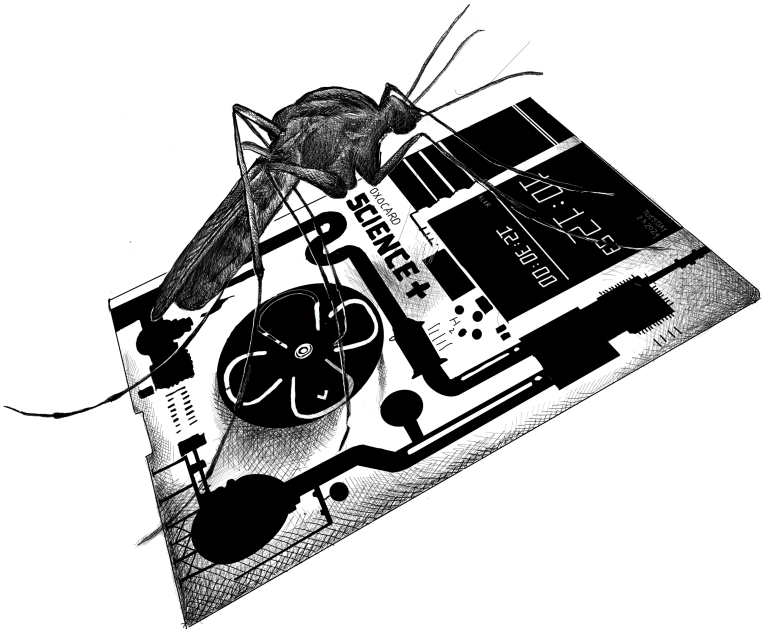
ETH zürich

Department of Computer Science
Algorithms and Didactics



Implementing the Oxocard for WebTigerPython

Josh Anderegg

Bachelor's Thesis

2024

Supervisors: Prof. Dr. Dennis Komm
Alexandra Maximova
Clemens Bachmann

Abstract

WebTigerPython is a web based Python IDE that teaches programming to students of all ages. This thesis explores the integration of Oxocards, a series of educational devices, into WebTigerPython, enabling them to be programmed directly from the platform. Since Oxocards do not natively support Python, an alternative method for code execution on these devices was developed.

The proposed solution leverages the MQTT messaging protocol to enable communication between the IDE and the Oxocards, allowing instructions to be sent and executed on device. This approach was rigorously tested, demonstrating both its strengths and limitations.

The final implementation effectively abstracts away the underlying complexity, making it seem to the user as if the Python code is running directly on the Oxocards.

Acknowledgment

I would like to express my gratitude to everyone who supported me throughout the development of this thesis.

First and foremost, I am grateful to my supervisors, Alexandra Maximova and Clemens Bachmann, for their guidance, constructive feedback, and appreciation for my efforts. Their support enhanced the quality of this work significantly and their encouragement provided me with the motivation to persevere.

I extend my sincere thanks to Prof. Dr. Dennis Komm for giving me the opportunity to conduct my thesis within his research group. His reassurance and feedback after the interim presentation were much appreciated.

Special thanks go to Andre Macejko for his technical assistance in setting up the MQTT server and to Giovanni Serafini for his kindness and openness when I first reached out to him about available research topics.

I am also deeply appreciative of the support from Thomas Garaiao and Tobias Meerstetter at Oxon AG, who generously provided access to their API, which was essential to this thesis. Thomas Garaiao offered continuous feedback and technical support, particularly when unexpected challenges arose with the Oxocard.

Lastly, I wish to thank Mariana Renteria for designing the cover page of this thesis. No work in computer science is complete without an animal on the front and her mosquito matches the content of this thesis perfectly.

Contents

Chapter 1

Introduction

1.1 Background

TigerJython [19] is a programming environment designed to teach Python in schools. It offers a user-friendly interface with a code editor to write Python code, buttons to start and stop the execution and error handling capabilities that highlight problematic lines with helpful suggestions in order to fix them. In addition to the execution of standalone Python code, TigerJython also enables to program external devices, further motivating students to learn programming.

Over the years, there were multiple iterations of TigerJython. In the current iteration WebTigerPython [9], the editor has become a web-app. This change made the editor more accessible, as it can be run on any device with a browser. However, this also lead to the removal of old features, such as the support for the Oxocard Blockly [1] device.

While this thesis was motivated by the idea to re enable the use of the Oxocard Blockly card, it turned out that Oxon [6], the manufacturer of Blockly, had also released new iterations of their educational cards [5]. As such, this thesis aimed to implement as many of them as possible for the WebTigerPython environment.

1.2 Main Contribution

This thesis explored various methods in which the Oxocards can be incorporated into the WebTigerPython environment. The procedures used can be reused for other devices and we aim to provide guidance for how to do it. Further, this thesis discusses other design decisions of WebTigerPython such as the trade-off between type inference and type enforcement, as well as the design of device interfaces in general.

1.3 Content

In Chapter 2 the specific problems and constraints for this thesis will be discussed, along with possible solutions. In Chapter 3 the different components of the final solution will be introduced and discussed, followed by an evaluation of their performance

and limitations in Chapter 4. Finally, in Chapter 5 we will conclude this thesis and discuss what future work will be important for WebTigerPython.

Chapter 2

Problem Statement

This chapter establishes the context for this thesis, formulates the goals of the project and discusses the various approaches to reach these goals.

2.1 Context

WebTigerPython

WebTigerPython [9] is the latest iteration in a series of integrated development environments (IDE) that allows for a user-friendly and interactive Python learning experience. It runs in the browser and supports the use of the majority of the Python standard library, as well as external pip packages.

It achieves this by utilizing Pyodide [13], a project that enables Python to run in browsers. Pyodide accomplishes this by running the CPython interpreter in WebAssembly, allowing Python to run outside of its traditional environment, namely in any browser that supports WebAssembly. Consequently, it runs purely in the browser, eliminating the need for a server to execute the Python code.

Transitioning from a desktop application to a web based IDE has resulted in the loss of compatibility to some devices [22]. This was due to browsers lacking access to the USB port, primarily for security reasons. In 2021, with the upcoming of WebUSB [23], devices such as the Calliope and the micro:bit were supported again on Chromium-based browsers, as done by Bachmann [9]. However, some originally supported devices, such as the Oxocards, are still not reenabled.

The method by which WebTigerPython makes these devices programmable involves flashing Python code onto them and executing that code on the device. Specifically, this is done by implementing an interface that is responsible for both the flashing and execution of the code.

Oxocard

Oxocards are a series of educational devices designed and produced by Oxon AG [5]. The different cards differ in their features; for example, the Oxocard Artwork is equipped with a screen that supports both 2D and 3D rendering, while the Oxocard Science adds scientific sensors, with the Oxocard Science+ incorporating even more sensors. A promotional image of the Oxocard Science+ can be seen in Figure 2.1.

Figure 2.1. Promotional image of the Oxocard Science+.

Therefore, the Oxocards can be used to render animations on the screen, collect data through the sensors, or even play and program games using buttons on the card.

These possible programs are all written in NanoPy [4], a lightweight, Python-like scripting language created by Oxon. The intent behind creating a custom scripting language is to achieve faster execution and compile time, compared to more widely used scripting languages like Python. To enhance code execution speed, all memory operations are stack-based. For this to work, the sizes of collections like strings, lists and dictionaries must be known at compile time. As a result, Python-like lists do not exist and are replaced by statically typed arrays of fixed size. Other collections, such as dictionaries, are removed from the language entirely. While these restrictions facilitate the aforementioned performance advantages, particularly on the limited hardware, they significantly constrain the development capabilities.

To write and deploy NanoPy code to the card, Oxon provides an online editor [3], as illustrated in Figure 2.2. The editor features sections dedicated to example scripts, comprehensive documentation and a debugger that allows for a line-by-line execution of scripts. Additionally, the editor provides code completion, displays error messages and includes a console for printing values at run time.

After sending the code to the Oxocard, an on-device compiler compiles to bytecode. In the end, this bytecode is interpreted by a custom-written interpreter on the device. By running a scripted language on the device, Oxon can implement user-friendly features, such as the aforementioned debugger, but at the same time they do not lose out on performance due to asserting control over the compilation and interpretation. Scripts that are transmitted can also be stored on the Oxocard and executed using the hardware buttons to navigate a graphical user interface (GUI) on the screen. Within this GUI, users can also configure settings, test the hardware and interact with the file system.

Overall, the Oxocards have a plethora of user-friendly functionality already in place, without suffering performance issues due to the limited hardware.

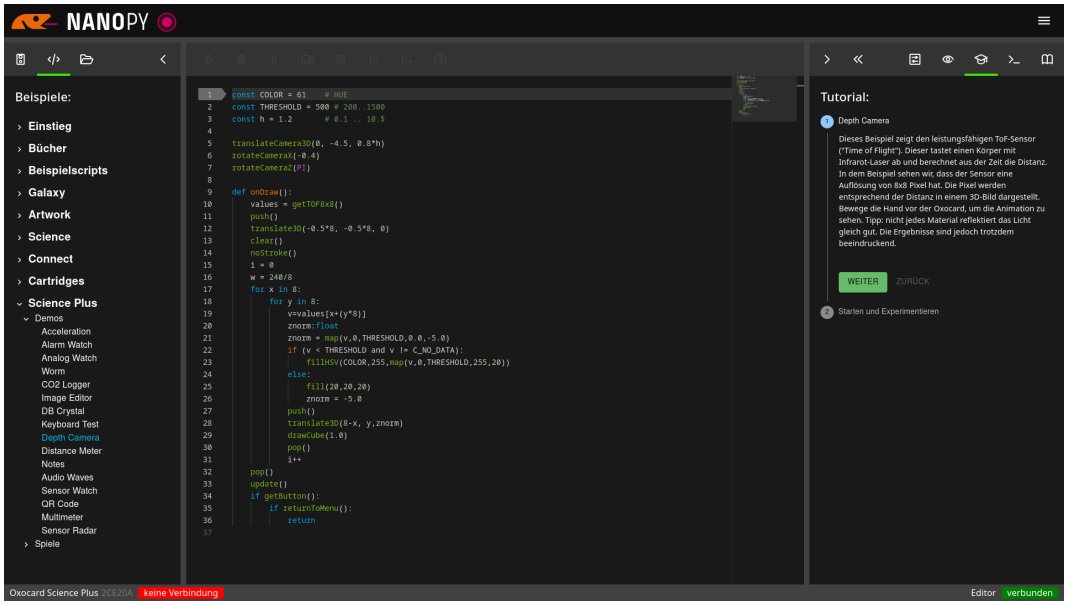


Figure 2.2. Screenshot of the NanoPy editor.

2.2 Goals

Within this context, the implementation should take into account both the functionality provided by WebTigerPython and the Oxocards. From WebTigerPython's perspective, the Oxocard should be implemented like any other device. The user simply selects the Oxocard as a device, establishes a connection and then executes code on it.

Whereas from the perspective of the Oxocards, the implementation should be compatible with the software already in place and not limit the ability to flash scripts from the NanoPy editor, rather than from WebTigerPython.

Independent of both perspectives, the implementation should be as performant as possible, leveraging the capabilities of the Oxocard, while providing the user-friendly experience associated with Python, as opposed to NanoPy.

2.3 Approaches

In order to achieve these goals, multiple approaches were considered and evaluated.

Flashing our Own Custom Firmware

This first approach is based on the previously used approach for implementing devices for WebTigerPython. Essentially, this involves writing custom firmware and flashing Python code onto the Oxocards.

This approach would grant full hardware control and full compatibility with the way WebTigerPython interacted with devices until now. The performance and functionality of this approach are only subject to the quality of the firmware developed. All Oxocards utilize the ESP32 controller [14], for which firmware can be written in a reasonable time frame.

What this approach ignores, is the work already accomplished by Oxon. Any GUI they have implemented is rendered useless, as our firmware will simply overwrite it. It will also be infeasible to have both Python and NanoPy support, resulting in incompatibility with the NanoPy editor. Another considerable downside is that anyone who uses an Oxocard will need to flash our firmware before using it with WebTigerPython, which limits accessibility. Furthermore, there is the remaining issue of WebUSB, as it is currently only supported by Chromium-based browsers and Opera [23], thereby excluding users of Firefox or Safari. Moreover, different types of Oxocards will require different firmware, complicating reusability.

Overall, this approach removes the existing GUI, is not backward compatible with NanoPy and makes the Oxocard inaccessible to users who are not comfortable with flashing their own firmware or changing browsers. However, it fits into the current device interface of WebTigerPython and also provides full hardware control.

Transpiling to NanoPy

One way to overcome those issues is to directly flash NanoPy code using the same application programming interface (API) as the NanoPy editor. To generate NanoPy code, we would use a Python to NanoPy transpiler. Finally, transmitting the code does not pose a problem, as we have been granted access to the Oxon API.

This approach resolves the backward compatibility issues and would not require the user to flash custom firmware onto the cards, allowing the cards to be used straight out of the box.

Transpiling from Python to NanoPy presents several challenges. The differences outlined between the two in Section 2.1 showcase the challenge of transpiling one into the other. The main issue comes from the aggregate structures like lists and dictionaries. Python lists, for example, would either require the user to only use single type lists with a fixed length, or would force us to find a way to have Python like lists within NanoPy. One would require the user to pretty much write NanoPy code in Python, the other would force us to write another type system on top of NanoPy. Even if lists worked, other Python features like switch statements, asynchronous function calls and the standard library would cause further issues.

Overall, this approach would provide full compatibility and excellent performance. However, it would at the same time either strictly limit the possible code that can be written or would require the implementation of a Python interpreter on top of NanoPy.

Using Remote Procedure Calls

The concept underlying this approach is based on remote procedure calls (RPC), sending requests over an interface to another device in order to call procedures on that other device. Instead of transpiling Python to NanoPy, a Python Oxocard API is provided that sends Oxocard function calls to the device and returns a value, if a return value is expected.

This approach combines the advantages of all previous ones. Notably, it does not require any form of transpiling, instead all Oxocard functions are implemented as Python functions that send and receive instructions to and from the card. Instructions are then interpreted on the card by a NanoPy script, which responds accordingly.

Thus, there is no interference with any Oxon functionality. Users of WebTigerPython can switch to a different tab to use the card with the NanoPy editor or exit the execution using the card's buttons to finally load a previously sent script. This approach also allows for plug-and-play functionality. Apart from a single linking operation, users do not need to flash anything to the Oxocard, they can instead link it once and start directly using it. Further, it does not require custom firmware per device, all Oxocards running on NanoPy can be controlled with this script. By having separate code executions in the editor and on the device, it is also possible to use previously implemented features for WebTigerPython, such as the GPanel [11], or matplotlib.

The primary drawback of this approach is the latency it introduces. With this approach the speed of the network dictates the speed of the program. The network might also introduce reliability issues and a general rise in complexity. Complexity increases because two devices operate in parallel and communicate through a protocol, which is considerably more complex than a single device executing commands sequentially. Furthermore, any offline capabilities also become impossible to implement. Without an internet connection, there is no way to use an Oxocard with this approach.

Although the disadvantages are not negligible, this was ultimately the selected approach. We argue that the major disadvantage is the introduced latency, rather than the increase in complexity, as the latter can be managed with sufficient care and cautious design. Chapter 4 provides further details on how detrimental this choice is to performance.

Chapter 3

Implementation

This chapter discusses the implementation details of this thesis, mainly how we achieve the RPC approach discussed in Section 2.3. To accomplish this, we will introduce the used messaging protocol in Section 3.1, describe our RPC protocol in Section 3.2, reference the Python library in Section 3.3, document the written NanoPy code in Section 3.4 and finally, showcase how everything was integrated into the existing application in Section 3.5.

3.1 MQTT

The underlying messaging protocol for our RPC protocol is Message Queuing Telemetry Transport (MQTT). MQTT is a lightweight, publisher-subscriber communication protocol, primarily used for Internet of Things (IoT) devices [8]. This protocol was selected as a lot of devices already provide an MQTT interface, including the current generation of Oxocards.

The publisher-subscriber scheme operates through designated topics, which clients can either subscribe to or publish over. Thus, topics can be interpreted as named communication channels. Topics can also be used in a hierarchical fashion. Given two topics `'oxocard/user1'` and `'oxocard/user2'`, we can subscribe to each of them separately. However, MQTT allows you to publish or subscribe to both channels simultaneously by using special characters like `'+'` or `'#'`. The `'+'` character serves as a single-level replacement character, `'oxocard/+'` therefore matches any topic that starts in `'oxocard'` and is followed by a single topic afterwards. The `'#'` character goes a step further and accesses all subtopics.

Topics do not need to be explicitly created. When a client is connected to the server, it may publish or subscribe to any topic, regardless of whether others are listening or providing messages. This, combined with the aforementioned special characters, makes MQTT very easy to work with. It is simple to monitor all communication, or create new communication channels.

MQTT is not a peer-to-peer protocol, it rather relies on a broker in the middle. The broker has to keep track of the different topics over which messages are published and is responsible for forwarding messages to subscribed clients. This centralized broker introduces a single point of failure, generally making MQTT less reliable compared to most peer-to-peer protocols. Most brokers use the Transmission Control Protocol (TCP) as the underlying transport protocol [16] and avoid adding any

overhead larger than a single byte. Consequently, the performance and memory overhead when using MQTT instead of TCP is nearly inexistent.

In conclusion, MQTT is an excellent choice for WebTigerPython. Most devices already support an MQTT interface, the performance overhead compared to TCP is minimal and the ability of creating topics on the fly makes it easy to work with. The potential single point of failure is further evaluated in Section 4.1.

Mosquitto Server

For this thesis, a Mosquitto server was set up. Mosquitto is an open source MQTT broker implementation provided by the Eclipse Foundation [21]. Written in C, it has the lowest memory and CPU footprint among the most common MQTT brokers [10]. It is also easy to deploy and configure. However, these benefits come at the cost of reliability and latency when compared to other brokers [10]. Such issues primarily arise if the broker has to handle a large number of clients with substantial message sizes. Fortunately, neither of these factors poses a problem in our case. Given its performance promise and low memory overhead, Mosquitto is a great choice for WebTigerPython.

Our configuration for the Mosquitto server differs little from the default configuration. To enable communication with the broker via the browser, the WebSocket protocol was enabled, as well as the `8080` port. For our use case, anonymous clients are permitted, meaning that clients do not need to authenticate themselves when establishing a connection. This decision was made with the assumption that WebTigerPython will eventually become open source and since the login credentials are visible in the code, any authentication becomes obsolete. For further reference of possible Mosquitto configurations, refer to [15].

Security in general was an afterthought when setting up the MQTT broker, as there is minimal incentive for hi-jacking or flooding an MQTT broker. If an adversary wished to use an MQTT server to send messages across, they could easily access any of the publicly available MQTT brokers. There is no sensitive information stored on the machine that runs the broker, nor is there any monetary risk, as the broker is hosted on a fixed amount of hardware.

PyoMQTT

With a functioning MQTT broker, two MQTT clients can now communicate seamlessly. This was easily achievable for the Oxocard, as Oxon provides an MQTT API. However, with Pyodide there were significant issues. The issue stems from the fact that Pyodide currently has no support for Python sockets [12]. As a result, it is impossible for Python MQTT libraries, like `paho-mqtt`, to work out of the box. Therefore, a custom MQTT library, called PyoMQTT, had to be developed to work with Pyodide.

The general idea behind PyoMQTT is to use an MQTT TypeScript package to manage all functionalities related to MQTT and to have an interface for accessing these functionalities from within Pyodide. Due to our specific use case, PyoMQTT simplifies the MQTT interface compared to packages like `paho-mqtt`. Rather than using handler functions to process incoming messages, PyoMQTT is designed to only send and receive messages over a predefined MQTT topic.

```
1 from pyomqtt import Client
2
3 device = 'oxocard'
4 sessionID = '00'
5
6 client = Client(device, sessionID)
7 await client.connect()
8
9 client.send('Hello World!')
10
11 answer = await client.receive() # answer = 'Hello back'
12
```

Figure 3.1. PyoMQTT "Hello World" program.

A PyoMQTT client is created from within a Pyodide Python instance. For this we need to provide an ID and a device string to the constructor. After awaiting the connection, we can send and receive values using the client. A simple `'Hello World'` program can be seen in Figure 3.1. After the construction, all communication happens over the topic `'{device string}/{ID}'`.

To understand how PyoMQTT works, imagine two threads: one responsible for executing the Python code with Pyodide and the other tasked with handling the communication with the MQTT broker. We will refer to these as the Pyodide thread and the MQTT thread, respectively. The task of PyoMQTT is to enable communication between these threads, ensuring that values declared in the Python code can be transmitted to the MQTT broker and that values received from the MQTT broker can be relayed back to the Pyodide thread. To analyze how this is accomplished, let us examine line 6 in Figure 3.1 in greater detail.

First, a PyoMQTT client is created in the Pyodide thread. This object contains a field for incoming messages called `answer`, as well as an MQTT session object. The MQTT session object is instantiated by the PyoMQTT client but operates within the MQTT thread, allowing them to run in parallel. As this session is a field of the PyoMQTT client, we can use the session to move messages from the Pyodide thread to the MQTT thread. In order to also receive messages from the MQTT session, we need to provide a way for it to write back values into the PyoMQTT client. To do this, the PyoMQTT client has a method called `pipe`, that simply writes to the value of the `answer` field. Now, when the MQTT session is created, we essentially provide this method as a pointer, enabling the MQTT session to write to the `answer` field. Figure 3.2 showcases how the pointers are set and how messages are forwarded to the client.

The MQTT session is connected to the Mosquitto server via a WebSocket [24]. To send and receive messages from the broker, the session publishes and subscribes to two distinct channels: `editor` and `device`. This is done for convenience sake, as if a client publishes to a channel that it is also subscribed to, it also receives the published message. This leads to boilerplate code, where the client needs to discard any messages it has published itself. By separating the channels, published messages do not have to be handled in code and as described in Section 3.1, such channels incur minimal overhead.

With everything initialized, let us revisit the example run of Figure 3.1, where

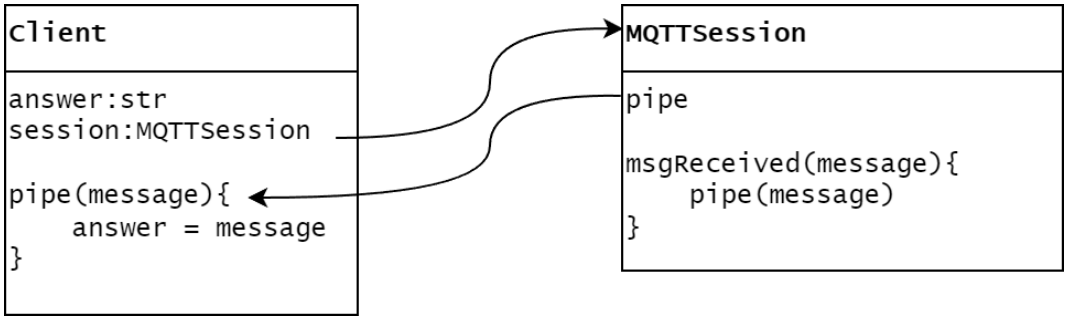


Figure 3.2. Pointers after the creation of the PyoMQTT client. `msgReceived` is the function that is automatically called if MQTT messages come in, in case this happens the MQTTSession will simply pipe it to the PyoMQTT client.

our Python code sends the message `'Hello World'` and receives the answer `'Hello back'`. Initially, the client is created and the connection awaited. When sending the message with the client, the MQTT session publishes it over the topic `'oxocard/00/editor'`. If the device functions correctly, it listens to this topic and afterwards publishes `'Hello back'` over the `'oxocard/00/device'` topic, to which the MQTT session is subscribed. Finally, as soon as the session receives the message, it pipes it into the client object, which ultimately assigns the message to the variable `answer`.

To ensure that a user communicates only with their connected card, we needed to devise a method for generating unique IDs that only the Oxocard and the editor are aware of. This was done by creating a pseudo-unique string known as a session ID. This string is stored in the cookies of the browser and is provided both to Pyodide, as well as the card. This ensures that both Pyodide and the Oxocard can communicate over a known channel as described above. As both the device and Pyodide need this session ID, its has to be created on the application level, allowing both Pyodide and the NanoPy script to receive the correct ID.

In practice, the user is unaware of the creation of clients, the awaiting of connections or the establishment of communication channels. When WebTigerPython runs with the Oxocard selected, a client is automatically created within Pyodide. This client is globally scoped inside the Python execution environment, allowing any section of the code to transmit messages. Sending and receiving messages is always asynchronous, necessitating the use of `await` statements in front of most lines that interact with the client. To avoid having these statements scattered throughout the finished code we will use asynchronous wrapping discussed in Section 3.3. Combined, this means that in the end the code resembles the code in Figure 3.3, without any establishment or `await` keywords needed.

3.2 Protocol

With the ability to communicate between Pyodide and the device, a communication protocol has to be devised for our use case. Essentially, it should be a protocol for RPC that executes functions on the Oxocard and if a return value is expected, returns the return value.

```
from pyomqtt import Client

client.send('Hello World!')

answer = client.receive()

```

Figure 3.3. PyoMQTT "Hello World" program with automatic client generation and asynchronous wrapping.

```
{
    "sequenceNr" : 0,
    "actionID" : 24,
    "params" : [12, "foo"]
}
```

Figure 3.4. JSON from the first protocol.

Single Instructions with Parameters

The first challenge to be solved by the protocol is to send over any kind of instruction. For this, we needed a way to identify the instruction we wanted to execute, a way to provide parameters to it and a way to have return values.

In order to achieve this, we sent over JavaScript Object Notation (JSON) strings. These consist of a sequence number, an action ID and lastly a field containing parameters, for the action to be performed, see Figure 3.4. JSON was chosen, as it is a human readable, widely supported serialization format. This choice was also taken under the assumption that NanoPy provides an built-in JSON parser; however this assumption proved to be incorrect. After reading through the documentation carefully, we discovered that the built-in JSON parser only works for specific JSON formats used by NanoPy. After execution, the called procedure sent back an answer JSON, which consisted of the sequence number of the executed request and a return value field containing the corresponding return value. If no return value was expected, we would simply insert the string `'None'` into the return field.

Batching Instructions

This scheme was effective and easily debuggable due to the readability of JSON. However, its performance did not meet expectations. Executing each instruction individually proved to be very slow for interactive workloads, this is mainly caused by the network latency, which is beyond our control. To minimize this network overhead, we needed to reduce the number of packets sent back and forth. To achieve this, we decided to batch the instruction whenever possible. This meant, that multiple instructions were collected, transmitted and finally executed on the card. Consequently, the bottleneck shifted to the number of instructions that could be sent at a time. The Oxocards have an upper limit of 1024 symbols that can be received from MQTT, making every symbol count. As a result, JSON was discarded in favor of a protocol that required fewer symbols to send the same number of instructions.

```
foo() # id = 0x78
bar(75, 42.0) # id = 0x0
foobar([0, 1, 2]) # id = 0x2D

```

```
25~78~0~75~42.0~2D~0~1~2~
```

Figure 3.5. Conversion of function calls to an execution string.

This version of the protocol produces a string of characters from a stream of instructions. This string is sent over without any additional sequence numbers or parameter fields. It works by separating action IDs and their parameters with a separation character, denoted as `'~'`. The only thing we declare before the stream of instructions is the number of total characters in the string. The rationale for this will be discussed in Section 3.4. Following the total number of characters, each instruction is appended as its action ID, with its parameters following after a separation symbol. An example of this can be seen in Figure 3.5.

With this upgrade to the protocol, we can send large batches of instructions to the card, which are then processed on the card itself. This minimizes network overhead by allowing more instructions to be transmitted simultaneously. However, not all instructions can be batched; specifically no instruction that produces a return value can be batched, as there is a possibility that this return value will be used by another instruction afterwards. How this is achieved in code is further discussed in Section 3.3.

Symbol Minimization

At this point, performance is determined by the number of instructions we can fit into a batch. Therefore, we sought methods to minimize the symbols needed for a single instruction. All floating-point numbers are rounded to a precision of two digits, as additional precision would waste space. All action IDs are sent over as hexadecimal numbers, which can save a character per instruction. For example, the highest action ID is 109, which is represented with two symbols in hexadecimal format instead of three in decimal notation. Strings are inserted as they are, which makes it impossible to send over strings with the separation character `'~'` included. This issue could be resolved by using some sort of string delimiter or escape character. However, this proved to be harder than expected; the Oxocard MQTT stack replaces built-in ASCII special characters with a null character. Since this null character also serves as the string delimiter in NanoPy, it could potentially lead to issues. Although we could have developed a work-around, we ultimately prioritized minimizing the symbol count and maintaining the script's simplicity over allowing the `'~'` character in strings.

Multiple Batches

By batching instructions, the performance of certain workloads improved drastically. In particular, animations could now be played more smoothly, as multiple frames could be sent over at a time. However, despite this improvement, a slight stutter persisted between the batches. Currently, the next batch is sent after the previous batch has completed execution and sent a message back to the editor. This causes

the stutter between the batches, as there are no frames that are available to be rendered by the card while waiting for the next batch.

To achieve smooth animations, we needed a sliding window algorithm that could transmit multiple instruction batches at a time. By sending multiple batches, there is no waiting time after a batch was processed, instead the next batch has already arrived and can immediately start execution. The general approach for this solution would involve reintroducing sequence numbers to order incoming batches and then execute them sequentially. However, this would prove challenging to implement with the available functionality on the Oxocard. We would need a way to efficiently collect and order incoming packets by their sequence numbers. Currently there is no built-in support for arrays of strings in NanoPy, as strings are just byte arrays and there is no support for two dimensional arrays. This combined with the memory restrictions, would make the collection of incoming messages cumbersome. Even if the collection could be accomplished easily, we would still need to find an efficient way to sort all the packets.

Instead, a simpler method was chosen. Once the Oxocard receives a batch, it sends a `send_next` message to the editor, indicating that it may send the next message already. As soon as the editor receives this message, it sends the next batch, which will most likely arrive during the execution of the previous batch, thereby eliminating the delay between them. As discussed, we would typically achieve this ordering using sequence numbers to verify if the incoming packet is the correct one to execute. Sequence numbers are needed, as the network does not guarantee that sent packets will arrive in order. However, with our approach, this issue is mitigated, as we only send over the next packet once the Oxocard confirms that it has received the previous one.

Other protocol messages like `send_next` exist, namely `send_previous` that sends the previous batch, this was done due to reliability issues discussed in Section 3.4. While the message `success` is sent in case a function did not have a return value to send back to the editor.

3.3 Python Library

With the RPC protocol described above, it was possible to focus on building a Python library that allows users to use the Oxocards. The design of the library followed the following general principles:

1. Allow the user to use all documented NanoPy functions.

2. Wherever possible, make using the functions more user-friendly.

3. Do not implement functions that do not need to be executed on device.

These principles are driven by user experience and performance. The first goal ensures that our solution is an extension to the work done by Oxon, not a reimplementation. The second leverages the advantages we have by using Python compared to NanoPy. The final goal allows us maintain optimal performance. By executing code inside of the editor, we omit using the network. Consequently, the more code that is executed locally, the better the performance. Overall, these

principles enable a backward compatible, user-friendly and performant interface for utilizing the Oxocard.

Implemented Functions

In general, if a function exists within the NanoPy documentation [2], it will work inside of WebTigerPython. Exceptions to this fall into six different categories.

Math Functions

All math functions of NanoPy are not supported for the simple reason that available Python packages like `math` implement the same functionality. As highlighted with the third goal, this `math` package is considerably faster than executing the math code on the card and exchanging values.

Vector Functions

NanoPy has its own vector class with its own class methods. These functions were not implemented as their usage within NanoPy is very sparse at the moment. The only function that takes in a vector is `rotateAroundAxis`. In cases where vectors are used, the user may simply provide a Python list instead. This list is type and length checked at runtime.

String Functions

String functions are already substantially covered in the Python standard library. Additionally, NanoPy strings lack a lot of features compared to Python strings, making the implementation of NanoPy string functions obsolete.

Time and Datetime Functions

Apart from the `delay` function, none of the time functions have been implemented. Time functions either get or set Unix time values, or start timers. The Unix time values are much easier to work with from within Python libraries, while implementing NanoPy timer functionalities would be difficult. In NanoPy, timers operate by defining an `onTimer` function, that gets executed as soon as a timer expires. This is hard to model with our approach and is not essential for most applications. Datetime functions are methods of an object called `dateTime`, allowing for time-related functions in a object oriented way. Everything `dataTime` can accomplish, the Python time package can also perform. The `delay` function is the sole exception, as it is frequently used for drawing animations.

File, Log, IO and Network Functions

In general, any function that directly reads from or writes to files is not implemented. For example, the `log` function, with which users can log sensor data to files on the file system. This also includes Input/Output (IO) functions, that would allow you to send and receive data over interfaces, as well as network functions, that let you send and receive HTTP requests as well as MQTT operations.

File manipulation functions were not implemented because WebTigerPython already offers the same file functionalities, with the only difference being that files are not stored on the device itself. Implementing network functionality would have likely caused issues, as our implementation already relies heavily on them. As a result, everything involving network and IO was not implement.

Other

Other not supported functions include any functions that require a very large parameters in terms of symbols. E.g. the `drawSprite` function that takes in 576 bytes and draws a sprite with each byte being equal to a certain color. The possible range of symbols exceeds the limit of the MQTT message size supported by the Oxocard. This is also true for `drawImageMono`, `drawImageMonoCentered` and `drawShape`. The other exceptions are `setRotation3D` and `getRotation3D`. Both are barely used in the example codes and apart from themselves, `rotation3D`, the object returned by `getRotation3D`, is never used for any 3D functions.

Type Inference

NanoPy is a statically typed language that enforces type rules for input parameters of functions, assignments and other operations. In contrast, Python is dynamically typed and does not enforce these rules, only raising errors once an undefined operation occurs. To bridge these two different paradigms, this implementation uses type inference, allowing for Python's liberal approach to types, while also enforcing sensible type checks for the correctness of the NanoPy portion of the code.

All type inference is done within Python. The general approach involves clamping and converting variables. For instance, if a value needs to be an integer in the range of $[0, 240]$, the implementation converts any given value to an integer and then clamps it to this range. For example, if the floating-point number 321.5 is provided it would be converted to the integer 240.

Additionally, our implementation allows for the use of more suitable parameters for functions. Consider Figure 3.6, where NanoPy requires users to provide constants that represent the properties of notes. In contrast, our type inferring environment enables users to provide more expressive arguments, like strings. The general approach was to maintain backward compatibility; the functions are designed interpret both the NanoPy constant, as well as the strings. This principle also applies for functions related to fonts, where fonts are represented as constants in NanoPy and as strings in our implementation.

Type inference, as described above, does have its downsides. For one, it has to be slower due to the executions of additional code, even if the slowdown is not impactful. It also leads to more unpredictable code. In a strictly typed environments, users are made aware of type errors, in our inferred environment, they are not. This lack of awareness may result in bugs that are harder to observe and track. This issue is a common problem in weakly typed languages, a potential solution will be discussed further in Section 5.3.

On the other hand, as long as the written code is correct and sufficiently simple, a type inferring environment is more convenient and forgiving for minor type errors.

```
beginSong()
note(C_NOTE_A, C_DURATION_1_4)
note(C_NOTE_B, C_DURATION_1_4)
note(C_NOTE_C, C_DURATION_1_2)
endSong()
playSong(false)
```

```
beginSong()
note('a', 'q')
note('b', 'q')
note('c', 'h')
endSong()
playSong(false)
```

Figure 3.6. Comparison between NanoPy and Python code for recording and playing a Song. Note `'q'` is chosen as abbreviation of quarter and `'h'` for half.

Users are not required to explicitly cast values for function calls and can also opt for more expressive ways to write programs, like in Figure 3.6.

Overall, type inference provides greater convenience than it causes problems, provided that it is widely and consistently enforced throughout WebTigerPython.

Design Guide

All implemented functions adhere to a style guideline. Before defining the function's code, we attach the previously mentioned `actionID` to the function. For testing purposes, we also define the `testParam` field, which specifies the types of parameters of the function, enabling them to be randomly generated during tests. Lastly, the `hardwareNotSupported` field lists the different Oxocards for which the function is not defined. This assignment of values is accomplished using Python decorators, which enhance readability compared to declarations made within the function's scope.

All functions are defined as asynchronous because the response of the Oxocard has to be awaited before continuing the execution. Additionally, documentation strings are used to provide a description for each function. These strings resemble the NanoPy documentation for the same functions but are occasionally altered to be more uniform. The input values are clamped and converted as described in Section 3.3. Afterwards, we verify whether the connected Oxocard supports the called function by calling the `cardCheck` function. Finally, depending on whether the resulting instruction can be batched, as discussed in Section 3.2, we set the `batchExec` parameter to true or false. An example of such an implementation can be seen in Figure 3.7.

Points

Some functions have different parameters than their NanoPy counterparts, an example of these parameters are points. For geometric functions in NanoPy, users must always supply the x and y coordinates of a point as separate arguments. To draw a pixel at the coordinate $(0, 0)$, you would call `drawPixel(0, 0)`.

We changed our approach by treating all points as tuples of two elements. To draw a pixel with our method, the user would call `drawPixel((0, 0))`, passing a tuple as a parameter. The rationale behind this is that when functions accept multiple points as parameters, it is hard to follow where these points lie in the coordinate space. Figure 3.8 illustrates this issue and demonstrates how using tuples allows us to abstract these points into separate variables. This tuple representation also aligns with the mathematical notation for points in most Cartesian systems, as

```python
@actionID(69)
@testParam([randbool])
@hardwareNotSupported(["galaxy", "science", "science+", "connect"])
async def playSong(autorepeat:bool):
    """plays the recorded song

    Args:
        autorepeat (bool): should the song be repeated
    """
    if not isinstance(autorepeat, bool):
        raise ValueError("autorepeat should be a boolean")
    cardCheck(playSong.notSupported)
    params = [autorepeat]
    await client.request(playSong.actionID, params, batchExec=True)


```

Figure 3.7. Example for the standard implementation of a function.

```
drawTriangle(0,0,120,0,120,120)

```

```python
x = (0, 0)
y = (120, 0)
z = (120, 120)
drawTriangle(x, y, z)

```

Figure 3.8. Comparison between NanoPy code, where we supply six integers vs. Python code where we can abstract the points out.

shown in Figure 3.9. We believe that the representation of points in code should correspond to the representation that users learn in school. These two reasons led us to prefer tuples over the approach used by NanoPy.

Asynchronus Wrapping

As previously mentioned, all functions are asynchronous. However, to replicate the NanoPy experience, users should not be forced to insert `await` statements throughout their code. This issue was addressed already by Csurgay [11] when the similar problems arose. The solution involved wrapping all functions as asynchronous prior to the execution. Consequently, no asynchronous function call needs to be explicitly awaited, instead it is handled automatically. The result is a slightly slower execution for a more uniform and user-friendly code writing experience. The slowdown is more pronounced if many non-asynchronous functions are converted to asynchronous ones. In our case, this is not an issue as all Oxocard functions are asynchronous anyway.

One alternative approach that was considered is to preprocess programs by inserting `await` statements wherever they are needed. However, the downsides would include skewed error handling, as inserted keywords change the line positions of errors, as well as the occurring error. This will be discussed further in Section 5.3.

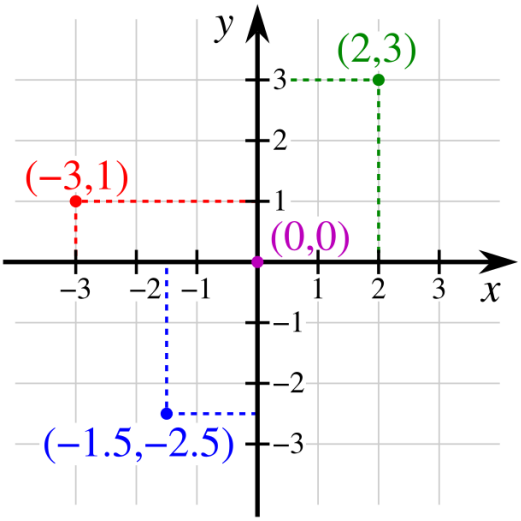


Figure 3.9. Standard representation for points in a Cartesian coordinate system, from K. Bolino.

Object Oriented Patterns

The initial version of this Python library heavily relied on object oriented patterns. When establishing a connection to the Oxocard, the user had to create an Oxocard object, with all implemented functions being methods of this object.

This approach was swiftly replaced by a more procedural one. The procedural code style is facilitated by the global PyoMQTT client discussed in Section 3.1. Procedural patterns were favored because it is unclear if users of WebTigerPython are already familiar with the semantics of object oriented patterns. Without objects that handle the connection, the code closely resembles the original NanoPy code, allowing users to use most NanoPy code, with some exceptions discussed in Section 3.3.

The primary concern that may arise from the departure from the object oriented approach, is the scoping of functions. Some of the functions inside the Oxocard module have very common names such as `fill`, `restart` and `update`. If someone defined a function with the same name inside of WebTigerPython by accident, unexpected behavior and failures would follow.

3.4 NanoPy Library

While the Python portion of the code was designed to be as reusable and comprehensible as possible, the NanoPy code focuses purely on performance. This emphasis on performance is due to the fact that NanoPy operates on embedded hardware, whereas the Python code runs on much more powerful machines. Additionally, NanoPy is less stable and predictable than Python, which further complicated the final code. This section will outline the general flow of the NanoPy code, highlight the optimizations that were implemented and address the technical challenges that we encountered.

```
elif action == 82:
    radius82 = strToFloat(getArg(0, params))
    height82 = strToFloat(getArg(1, params))
    drawCylinder(radius82, height82)
    return none

```

Figure 3.10. NanoPy function for fetching two arguments and drawing a cylinder.

Structure of the Code

This section outlines the implementation, while the following section will give more motivation for certain design decisions. The NanoPy code has three major sections: the initialization section, the messaging section and lastly the execution section.

A program will first execute the initialization code; which starts by establishing a connection to the MQTT broker and flushing previously received MQTT messages. Afterwards, a brief animation plays on the Oxocard's screen, which displays the WebTigerPython logo. Finally, the type of the Oxocard is sent to the editor, which is used to perform the hardware checks described in Section 3.3.

After establishing the connection, the Oxocard will continuously check for incoming instructions. Once the Oxocard receives a batch, the execution stack is parsed in by reading the fields separated by `'~'`. The first read field corresponds to the number of total symbols in the stack, while the subsequent fields represent action IDs accompanied by their arguments, as described in Section 3.2.

This action ID and its corresponding arguments are processed in a large if-else statement, as switch statements are not available in NanoPy. Depending on the action ID, the corresponding parameters are fetched and converted to the appropriate type before the function is executed. This process is repeated for all instructions on the stack, resulting in a sequential execution of the received batch. An example for such an execution branch can be seen in Figure 3.10.

Optimizations and Technical Issues

As hinted many times before, NanoPy does have its fair share of quirks that lead to unexpected optimization and workarounds. This subsection is a collection of some of them.

Binary Search for Conditions

As the number of implemented functions increased, so did the number of action IDs. With this increase, it became evident that actions with higher IDs performed worse than actions with lower IDs. This issue arose because NanoPy does not generate a jump table from the consecutive if conditions, resulting in longer execution times for lower branches. The workaround was to perform a three-level binary search for the action IDs, effectively halving the range of possible action IDs three times, as one would do in standard binary search.

With this approach, the most significant improvement is achieved for action 109, which required 110 comparisons before optimization and 19 afterwards. However, there is also a minimal overhead of three comparisons. For example, the action ID

0 previously only needed one comparison and now needs four. To further optimize this, one could consider using the frequency in which functions are used, assigning the functions with most usage the IDs with least comparison operations. However, for the sake of this project, this additional optimization was not deemed necessary.

MQTT Bugs

We previously mentioned that the length of the stack is always prepended before the actual execution stack. This was done, as the MQTT implementation of the Oxocard behaved unexpectedly at times, by forwarding only parts of the original MQTT message. As a consequence, we were forced to implement some form of reliable messaging, that lets the Oxocard request a retransmission of corrupted packets.

Other packet corruptions occurred, such as randomly receiving packets without any content or containing random noise. These packets could not be distinguished from valid messages, which is why all valid messages are prefixed with an `[Exec]` tag. Although this increases the number of symbols used for each instruction stack, it is necessary to solve the issue of packet corruption.

The last occurring bug was caused by the way MQTT messages are received by the Oxocard. The Oxocard stores all incoming messages in a queue, which is not emptied once the execution stops. As a result, if an execution was stopped and another was started, the new execution may receive old messages at the start. To resolve this issue, the MQTT queue is flushed during initialization by discarding all packets that are already in the queue.

Ultimately, all issues with the MQTT stack of the Oxocard were resolved with the inclusion of the packet length, the ability to receive the previous instruction stack again, the implementation of tags and flushing the MQTT queue.

Scoping

While implementing the portion of the code that processes stack instruction a peculiar type bug occurred. When converting a string to a floating-point value, the resulting value was incorrectly treated as an integer, leading to unexpected behavior. This bug was caused by the way NanoPy manages scope, specifically by the number of allowed scopes.

NanoPy has only three levels of scope: global, function and object internal. In contrast, most modern programming languages have a dynamic number of scopes; for example, if-clauses and for-loops create their own scope. To create these scopes, an abstract syntax tree (AST) is utilized. While the creation of an AST is not inherently a computationally hard problem, the Oxocard compiles on device, leading to significantly quicker compile times if the AST is not generated.

Consequently, typing issues can arise. Consider Figure 3.11, where we expect two distinct scopes to exist: the first between lines $3-6$ and the second between lines $8-13$. As discussed, these scopes do not exist in NanoPy, all variables in Figure 3.11 reside in the same scope of the function `process`. During compilation, the compiler attempts to infer the types of all variables. If a value appears multiple times, the type of the first occurrence determines the type of the variable in the whole scope. In this case, `x` is assigned the type integer on line 3. Issues arise, if the execution enters the second branch, where `x` should be treated as a floating-point number for the

```
def process(action:int) -> byte[120]:
    if action == 20:
        x = strToInt(getArg(0, params))
        y = strToInt(getArg(1, params))
        drawPixel(x, y)
        return none
    elif action == 21:
        x = strToFloat(getArg(0, params))
        y = strToFloat(getArg(1, params))
        z = strToFloat(getArg(2, params))
        # Throws an Error as x and y are thought to be integers
        scale3D(x, y, z)
        return none

```

Figure 3.11. Type bug in NanoPy.

function `scale3D`. Instead, the compiler has already assigned `x` the type of integer. As a result, during execution, `x` is automatically converted to an integer, effectively rounding the floating-point value down.

Sometimes the compiler catches the issue and provides an error message indicating that `x` is expected to be an integer rather than a float. For certain function calls, NanoPy performs automatic type conversions, specifically between floating-point numbers and integers. This hidden control flow meant that the execution continued with the rounded-down floating-point number, which altered the program's behavior. Ultimately, the simplest solution is illustrated in Figure 3.10. We simulate a scope by appending the corresponding action ID of a branch to all variables. If this rule is kept throughout code, the compiler will encounter exactly one type per variable, thereby preventing the type error from occurring.

3.5 Application

Until now, we have established the inner workings of all the moving parts. We discussed the MQTT broker, the way instructions are sent, the Python interface and the code executing on the Oxocard. This section demonstrates how all these elements work together to provide a coherent user experience for someone using WebTigerPython.

When opening WebTigerPython, the Oxocard can be selected like any other device. Upon selection an import statement is added to the text editor to indicate that the Oxocard is currently being used, see Figure 3.12.

Like with the other devices, the next step is to connect the Oxocard using the connect button. The user will be prompted to enter a code, see Figure 3.13. This code can be read from the Oxocard by creating a new pairing with a device. This process is identical to the one used when connecting the Oxocard to the NanoPy editor.

Once this code is entered in the prompt window, the NanoPy code described in Section 3.4 is sent over to the Oxocard and begins execution. A brief animation of the WebTigerPython logo shows the user that the editor is now connected. At this moment, the Oxocard also transmits its hardware number, which indicates to the

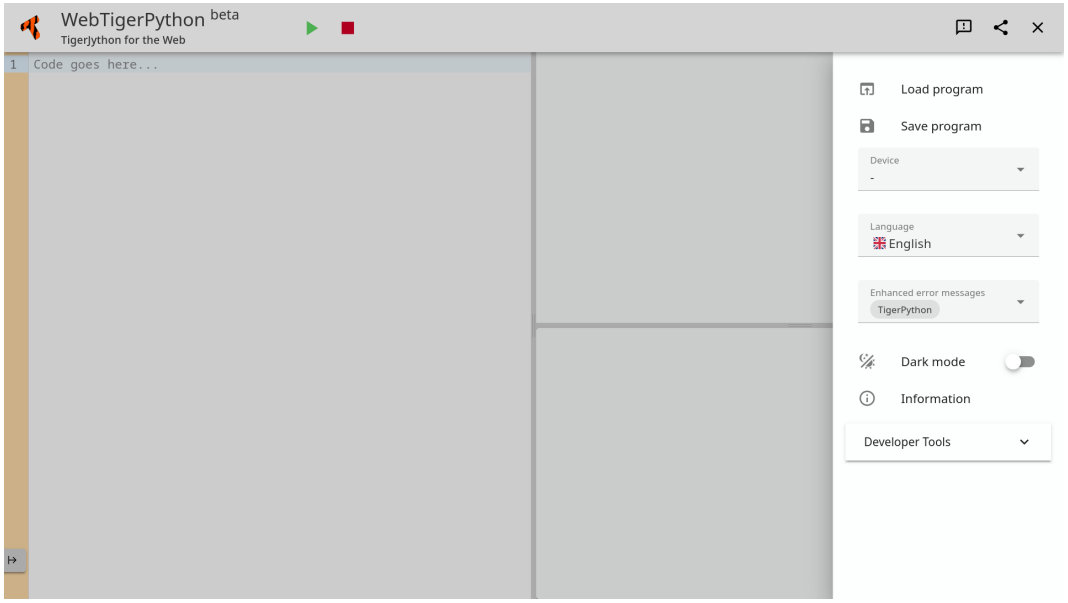


Figure 3.12. Device selection window. Under 'Device' the Oxocard can be selected.

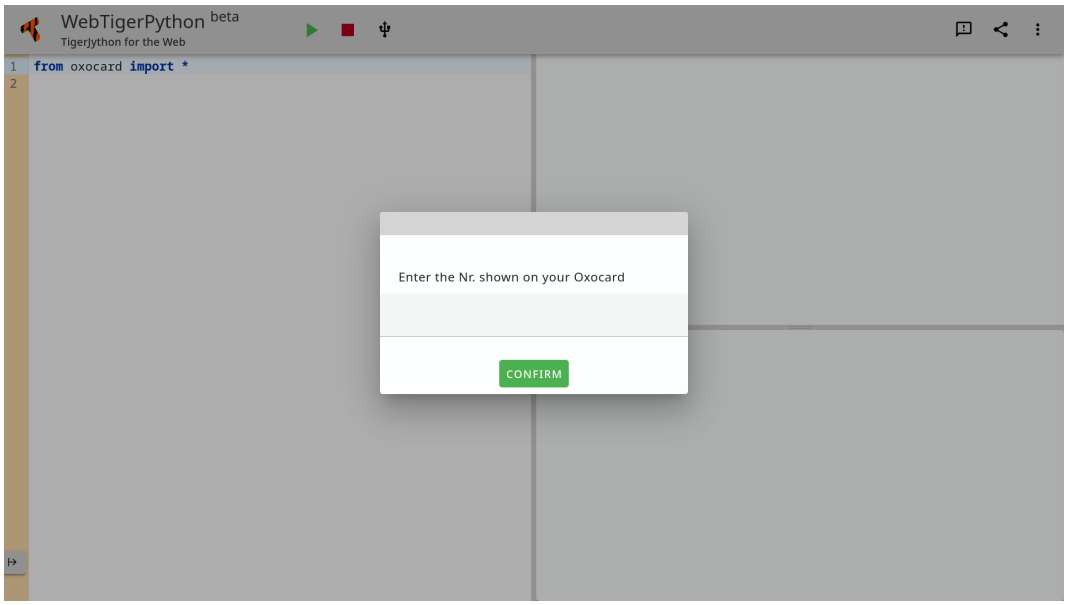


Figure 3.13. The prompt that appears when the Oxocard was selected and the connection button was pressed (next to the stop button). After entering the prompt the button will indicate that the connection was successful and code can be written and executed.

application which functions are available for use. This hardware number is used by the editor to perform the hardware checks described in Section 3.3.

The pairing between the Oxocard and the editor is now permanent. To remove the pairing, the device must be disconnected like any other device. This pairing is not limited to a single editor, the NanoPy editor can be opened in another tab and used to program the Oxocard in NanoPy. To switch back to the WebTigerPython program, our NanoPy script has to be sent over again. The easiest way to achieve this is by selecting the Oxocard as a device once more, which causes a retransmission of the script.

To execute code, a user writes it within the WebTigerPython editor, just like any other Python code. The written code can be executed and interrupted the same as normal Python code, with some difficulties to correctly interrupt at times. This is caused by the extensive use of asynchronous code, which may continue to run even after the interrupt button is pressed. In such cases refreshing the page is the simplest solution to restart the execution.

Overall, the Oxocard appears to the user like any other device, such as the Calliope. However, the underlying stack is much more complex and necessitates all the components discussed in this chapter.

Chapter 4

Performance

This chapter will examine the performance of different aspects of this thesis. To achieve this, we will conduct benchmarks and discuss the observed results.

4.1 MQTT Server

As mentioned in Section 3.1, the MQTT broker represents a single point of failure. If the MQTT broker ceases to function, or performs subpar, the entire implementation is affected. Therefore, it is essential to thoroughly evaluate the performance of our Mosquitto broker. This can be accomplished by benchmarking our broker.

For the duration of this section, assume that the broker runs on a shared Akamai-hosted virtual machine [7]. This virtual machine is equipped with a single core, `1GB` of memory, `25GB` of storage, network bandwidth of `1Gb`/sec and it runs the latest release of Alpine Linux.

Description

The used benchmark is written in Go and can be found under [20]. The benchmark essentially creates multiple clients that each publish messages to the broker and verify if the sent messages are correctly published. Since the clients are created locally on the machine running the benchmark, the measured performance depends as much on our own machine as it does on the MQTT server. For example, the throughput of the benchmark may be worse, if it is run over a low-bandwidth network. Therefore, the benchmark was conducted on a machine with a network bandwidth of `2.5Gb`/sec, exceeding the network bandwidth of the server.

The benchmark allows the specification of several parameters: the number of clients communicating with the broker, the number of messages each client exchanges, the time interval within which they send messages and the size of each message. All of the mentioned parameters have to be integers, for the message interval this corresponds to an interval in seconds. Consequently, we cannot accurately emulate sub-second message intervals. This limitation arises because MQTT is typically used in IoT environments, where message intervals on the order of seconds are the norm. If a message interval of 0 is given, the clients will send their messages as soon as they have verified the publication of the previous one.

Message count	Mean Latency	Messages/second	Fail rate
2 500	4.2 ms	133 271.05	0.096 ppm
5 000	5.4 ms	105 212.78	0.184 ppm
10 000	7.5 ms	89 392.27	1.238 ppm

Table 4.1. Results of the MQTT benchmark.

We run a series of benchmarks, with a fixed message interval of 0, message size of `1KB`, 500 clients and a variable message count, the results can be seen in Table 4.1.

Results

The first observation, is that the throughput (Messages/sec × Message size) approaches the achievable throughput of `1Gb`/sec ≈ `125MB`/sec. For instance, in the second row of the table, we sent `1KB` of data in every message and achieved about 105 212 messages per second. Consequently, the resulting throughput would be ≈ `105MB`/sec. This is a good sign, especially if you look at the small amount of introduced latency. The numbers also suggests that the current implementation is quite robust, with a low fail rate across all three tests.

One has to keep in mind that this benchmark is quite artificial; it is highly unlikely that 500 clients would independently from one another decide to flood our broker with `1KB` sized messages. For our use case, we expect smaller messages on average, fewer clients and a more balanced request distribution. Additionally, this benchmark does not allow to measure whether the failures were introduced by the broker or the machine that ran the benchmark. Further, during the benchmark tests, the Akamai monitoring tool did not register high CPU usage, suggesting that the bottleneck is either the bandwidth or the available memory.

Overall, it is fair to assess that the chosen MQTT broker is capable of supporting the presented implementation. Should this no longer be the case in the future, the MQTT broker can easily be redeployed on more powerful hardware.

4.2 MQTT Clients

After observing the low latencies measured previously, we were quite surprised to find that sending instructions and receiving answer between editor and card took around 400 ms. In this section we will refer to one such exchange as the round trip time (RTT) between the editor and the Oxocard. This RTT is effectively composed of two separate RTTs: one between the editor and the broker, the other between the broker and the Oxocard. Assuming that the overall RTT is symmetric, we would arrive at 200 ms RTTs between one client and broker, which is still significantly higher than the latencies measured in the benchmark presented in Section 4.1.

There are several possible explanations for these differences. First, the benchmark used Go, a compiled language that is arguably designed for concurrency and network functionality [17]. In contrast, our TypeScript Pyodide stack is likely to be significantly slower and the same may be true for the Oxocards. Additionally, the Pyodide component of our implementation employs WebSockets [24], while the

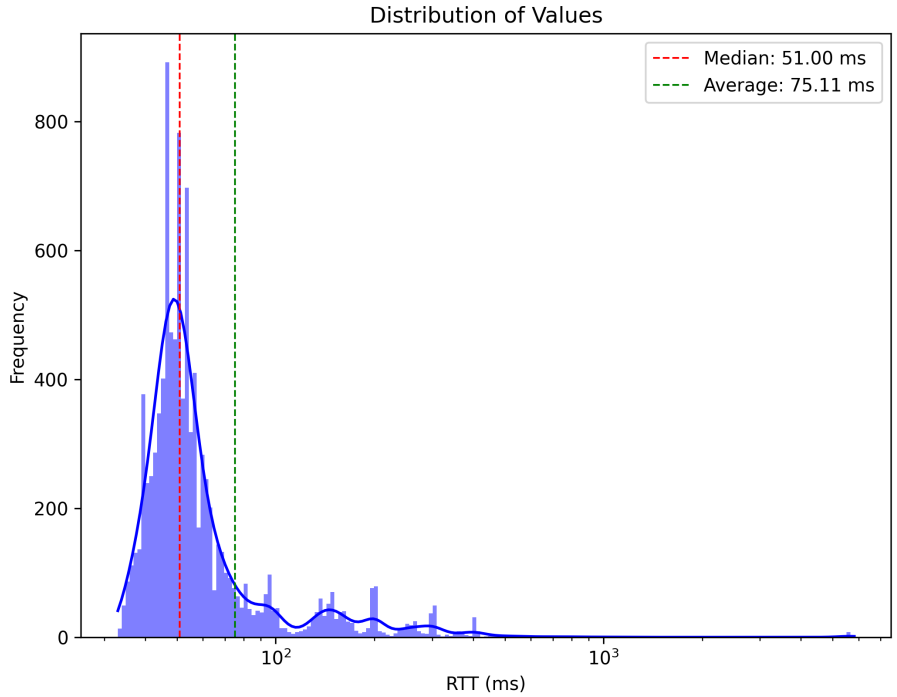


Figure 4.1. Distribution of MQTT RTTs for the Oxocard Science+, scale is logarithmic.

benchmark relies on standard transport layer sockets. By definition, WebSockets are slower than standard sockets because they are built on top of them. Finally, the assumption that the RTT is symmetric may be incorrect, suggesting that either the Oxocard or WebTigerPython could be significantly slower than the other. While it is not feasible to change the technology stack of the Oxocard or WebTigerPython, it is both possible and interesting to test their MQTT performance respectively.

Description

To test this, we performed another benchmark. In this benchmark, both WebTigerPython and the Oxocard Science+ made 10 000 requests to our MQTT server and awaited a response. To make the results comparable, both devices were connected to the same router and published the exact same `100B` message to the broker.

Results

With the described test, we obtained an average RTT of 75.11 ms for the Oxocard Figure 4.1. In comparison, WebTigerPython averaged a RTT of 20.56 ms Figure 4.2. A side-by-side comparison of both can be found in Figure 4.3.

The measurements indicate that the MQTT stack of the Oxocard has a significant impact on the total RTT. As shown in Figure 4.3, nearly all ($\approx 98.8\%$) of the WebTigerPython MQTT messages exhibited a faster RTT than the minimum RTT recorded on the Oxocard. This imbalance is further exacerbated by the approximately 10.8% of Oxocard RTTs that exceeded 100 ms, as illustrated in Figure 4.1.

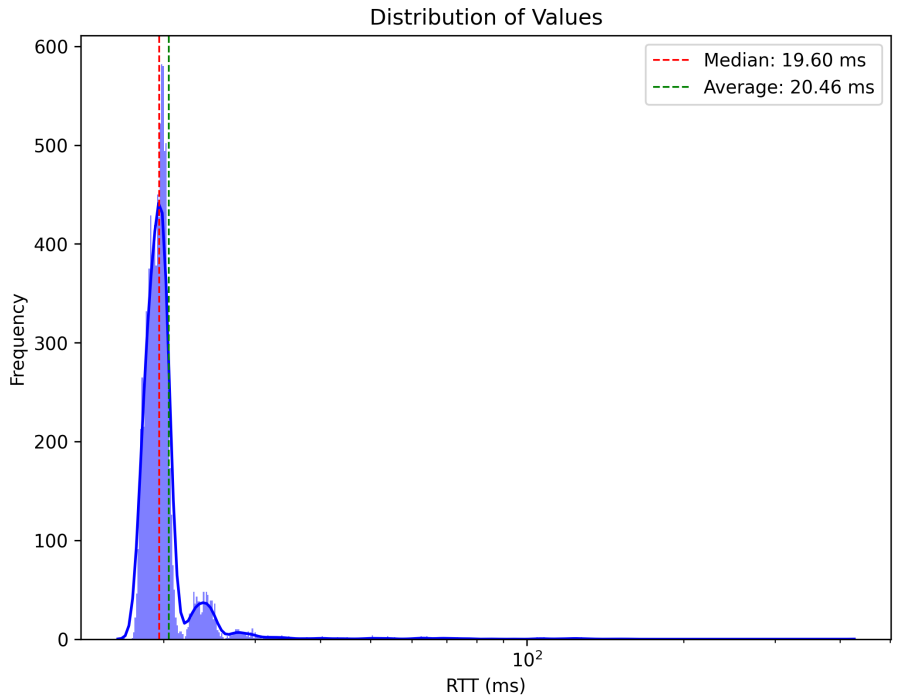


Figure 4.2. Distribution of MQTT RTTs for WebTigerPython, scale is logarithmic.

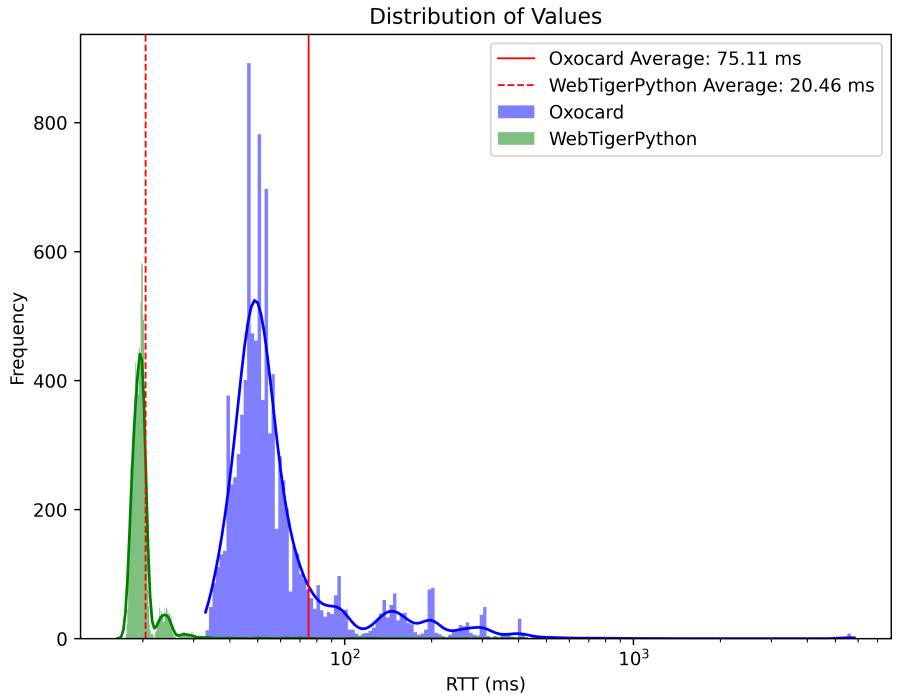


Figure 4.3. OxocardScience+ and WebTigerPython RTT distributions in comparison, scale is logarithmic.

Additionally, the average RTT is also heavily influenced by large outliers; During the testing, the largest Oxocard RTT was around 6000 ms.

It is unclear what causes these issues with the Oxocard. The most likely explanation is a limitation inherent the way the Oxocard is intended to work. WebTigerPython can utilize multiple concurrent threads to manage incoming messages, whereas the Oxocard interprets NanoPy on embedded hardware. It is fair to conclude, that with the use of MQTT in such a way, the Oxocard has reached its performance limits.

In comparison, it is surprising to see how well our PyoMQTT stack performs. Initially, we suspected that the workarounds necessary for MQTT to work inside of Pyodide would introduce larger amounts of latency. With an average RTT of 20.46 ms, it is still considerably slower than the benchmark, but considering the environment it is run in 20.46 ms are acceptable.

By adding both averages, we arrive at around 100 ms, which is considerably lower than the 400 ms measured for a single instruction exchange. The difference is likely due to a combination of the time required to execute the instructions and the fact that the benchmark is highly isolated. It is probable that both MQTT stacks perform slightly worse when more instructions are scheduled beyond just sending and receiving messages.

4.3 Execution of Programs

The previous benchmarks provide a useful performance estimate. However, they do not accurately reflect real usage of the implementation. This section aims to offer an overview of actual usage performance.

To achieve this, we tested three Python programs, each with a different type of workload. The first program polls values from the brightness sensor of the Oxocard and plots them in WebTigerPython. The second program executes the animation of a pendulum, utilizing the ability to transmit instructions in batches. The final program combines the two, by polling values from the brightness sensor and animating a spinning sphere on the Oxocard, with the sphere's color depending on the polled value. Each of the programs was executed 100 times and the time from the beginning to the end of each execution was measured.

Polling the Brightness Sensor

The program Figure 4.4 collects 50 lumen measurements from the Oxocard and subsequently generates a plot of the data.

It is important to notice here, that there is exactly one function called that operates on the Oxocard, namely `getAmbientLux()`. Apart from this, all lines of code execute within Pyodide.

From the measurements, we see, that the execution took ≈ 20 seconds on average. For easier comparison with the other programs, we assume that the duration of the program only relies on the number of called Oxocard functions. Therefore, we reason that 50 remote functions take 20 seconds to execute. This implies an average RTT of ≈ 400 ms, resulting in about 1.5 instructions per second.

```
1  from oxocard import *
2  import matplotlib.pyplot as plt
3  values = []
4  n = 50
5  time_points = list(range(n))
6  for _ in range(n):
7      val = getAmbientLux()
8      values.append(val)
9
10
11 plt.figure(figsize=(10, 5))
12 plt.plot(time_points, values, marker='o', linestyle='-', color='b')
13 plt.title('Brightness values')
14 plt.xlabel('Measurements')
15 plt.ylabel('Brightness (Lumen)')
16 plt.grid(True)
17 plt.show()
18
```

Figure 4.4. Lumen polling program.

This subpar instructions per second count was the main reason for redesigning our RPC protocol, as discussed in Section 3.2. Despite the improvements made to the protocol, this program will not perform any better. This is because `getAmbientLux` has a return value and as previously mentioned, functions with return values are never executed in batches. However, to showcase the improvements achieved, let us examine the pendulum animation.

Pendulum Animation

The program in Figure 4.5 draws a total of 250 frames of a pendulum animation. The code written for this animation comes from the example NanoPy script `pendulum.npy`, which can be found in the NanoPy editor. As outlined in Section 3.3, some changes had to be made to the code, in order to work properly. Specifically, lines $14 - 16$, where points and RGB values had to be put in tuples and lists to match our new type system.

This program is significantly more interesting in terms of performance, as the animation frames can now be sent in batches. Consequently, instead of executing a single instruction per RTT as before, we can execute multiple.

Per frame, we execute a total of six remote functions. Given that we animate 250 frames, this results in $6 \times 250 = 1\,500$ remote calls in total. It is important to note, that the batches are not sent over once every loop, instead they can accumulate over multiple iterations until a maximum number of symbols is reached. Resulting in batches that look like the one from Figure 4.6.

Working with the expected execution time from the previous section, that is 400 ms per instruction, we would arrive at a $1\,500 \times 400\text{ms} = 600\,000\text{ms} = 10\text{min}$ long execution time. However, a fully executed animation is measured at ≈ 7 seconds on average. Therefore, the number of instructions per second turns out to be $\frac{1500}{7} \approx 214.2$.

This is a considerable increase from the 1.5 instructions we achieved with the

```
from oxocard import *
from math import sin, cos, pi
a = pi/3
aVel = 0.0
aAcc = 0.0

for _ in range(250):
    clear()

    l = 160
    y = sin(a+pi/2)*l
    x = 120+cos(a+pi/2)*l

    drawLine((120,0),(x,y))
    fill([0,0,0])
    drawCircle((x,y),20)

    aAcc = -0.04 * sin(a)
    a+=aVel
    aVel+=aAcc

    aVel = aVel * 0.94

    update()
    delay(20)

```

Figure 4.5. Pendulum animation program.

[Exec]937~23~21~126~20~63~6a~20~3c~1e~120~0~21~126~6b~0~0~0~23~21~126~20~63~6a~20~3c~1e~120~0~25~128~6b~0~0~0~23~25~128~20~63~6a~20~3c~1e~120~0~31~133~6b~0~0~0~23~31~133~20~63~6a~20~3c~1e~120~0~40~138~6b~0~0~0~23~40~138~20~63~6a~20~3c~1e~120~0~52~144~6b~0~0~0~23~52~144~20~63~6a~20~3c~1e~120~0~66~150~6b~0~0~0~23~66~150~20~63~6a~20~3c~1e~120~0~83~155~6b~0~0~0~23~83~155~20~63~6a~20~3c~1e~120~0~101~158~6b~0~0~0~23~101~158~20~63~6a~20~3c~1e~120~0~119~159~6b~0~0~0~23~119~159~20~63~6a~20~3c~1e~120~0~137~159~6b~0~0~0~23~137~159~20~63~6a~20~3c~1e~120~0~154~156~6b~0~0~0~23~154~156~20~63~6a~20~3c~1e~120~0~168~152~6b~0~0~0~23~168~152~20~63~6a~20~3c~1e~120~0~181~147~6b~0~0~0~23~181~147~20~63~6a~20~3c~1e~120~0~190~143~6b~0~0~0~23~190~143~20~63~6a~20~3c~1e~120~0~197~139~6b~0~0~0~23~197~139~20~63~6a~20~3c~1e~120~0~201~137~6b~0~0~0~23~201~137~20~63~6a~20~3c~1e~120~0~202~137~6b~0~0~0~23~202~137~20~63~6a~20~3c~1e~120~0~200~138~6b~0~0~0~23~200~138~20~63~

Figure 4.6. Large animation batch from the pendulum execution.

```
from oxocard import *

W = 1.5
t = 0.0

colors = [
    [255, 255, 255],  # White
    [255, 0, 0],      # Red
    [0, 255, 0],      # Green
    [0, 0, 255],      # Blue
    [255, 255, 0],    # Yellow
    [0, 255, 255],    # Cyan
    [255, 0, 255],    # Magenta
    [128, 0, 128],    # Purple
    [255, 165, 0],    # Orange
    [0, 128, 128]     # Teal
]

for _ in range(50):
    lumen = getAmbientLux()
    color = colors[(lumen)%10]
    clear()

    fill(color)
    push()
    rotate3D((t, t, 0))
    drawSphere(5.0)
    pop()

    t = t + 0.05
    update()

```

Figure 4.7. Sphere drawing program.

previous program. This indicates, that the performance of our approach depends significantly on the executed program and strongly favors programs that can be executed in batches.

It is important to note that the minimum possible execution time for the above animation would be $250 \times 20\text{ms} = 5\text{sec}$, based on the number of frames and the delay between them. By taking 7 seconds on average the execution time is considerably close to the optimal duration.

It is also important to consider that the speedup is not only the result of batching the instructions, but also arises from sending multiple batches at once as discussed in Section 3.2.

Brightness Sphere Animation

The program in Figure 4.7 allows us to evaluate a combination of animation and sensor polling. For each frame of the animation, the program polls the current luminosity before rendering a rotating sphere in a color that corresponds to the last digit of the polled brightness value.

We can count 8 remote function calls within a loop. Combining this with 50

```
[Exec]2~07~
[Exec]46~3c~6b~0~255~255~3e~55~1.3~1.3~0.0~50~5.0~3d~63~
```

Figure 4.8. The two typical batches of the sphere drawing program.

frames, we get 400 remote calls in. According to the measurements, one execution took an average of 27 seconds, resulting in $\frac{400}{27} \approx 15$ instructions per second.

As anticipated, the result falls within the range of the two previous tests. This is due to the fact that while the animation can be executed in bulk, the polling of the data cannot. The fact that of the 8 remote procedures, 7 can be executed in bulk, still speeds up the execution considerably.

In Figure 4.8, we can observe what allows the speed up. During each execution we poll the luminosity, corresponding to the batch on the first line. The animation based on the polled value results in a batch as seen on the second line.

This result is interesting to compare with the measurements of the first program, where we simply polled the brightness values. We poll the exact same amount of values, with the only difference being that these values are now used to transmit another batch of instructions. One might expect that the execution of this program would take exactly double the time, but the actual increase is slightly below 50%. The intuition behind this is, once again, the multiple batches that are transmitted.

Every iteration after polling the brightness requires us to wait until the value is returned. All instructions from line 22 to 31 are effectively accumulated and sent together when we poll the luminosity again. Once we confirm that the Oxocard has begun executing the animation instructions, the next polling request is sent over. As demonstrated, this approach leads to a noteworthy increase in the number of instructions transmitted per unit of time.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

This thesis began with the simple goal of developing a method to program the Oxocards using WebTigerPython. While several approaches were discussed, the final method utilizes the MQTT messaging protocol to send requests back and forth between the Oxocards and our editor.

For this, we needed to determine which MQTT server to use, how to set it up and consider its scalability. We chose Mosquitto, an open source, high-performance and well-studied MQTT broker.

Having established the MQTT server, we needed to devise a protocol, with which the Oxocards and the editor communicate. The primary focus was the performance of the protocol, which we achieved by batching instructions, minimizing the number of symbols needed to represent instructions and sending over multiple batches simultaneously.

After having the communication in place, we developed a Python library that enables users to program the Oxocard from within WebTigerPython. This was done with considerations of usability, performance and code maintainability. Finally, we integrated everything into the existing WebTigerPython iteration, ensuring that it can be interacted with like any other device.

With the final approach, we can support most of the functionality of the Oxocard while remaining backward compatible with its intended usage. The proposed solution is also extensible to other devices and scalable to accommodate a large number of users.

5.2 Missed Opportunities and Shortcomings

Blockly

The thesis initially aimed to implement all Oxocards, including Blockly, but ultimately, Blockly is not compatible with our solution. Blockly does not run NanoPy and therefore can not be targeted with the proposed solution. However, there is a silver lining behind not implementing Blockly and focusing on a good NanoPy compatible solution. Oxon will likely continue to produce NanoPy compatible cards, which then can be targeted by this solution. Thus, by choosing this approach more Oxocards will

be available in the future. That is, as long as NanoPy is not replaced by something else.

RPC Protocol

While the current RPC protocol works very well, it would have been interesting to go further with the symbol minimization. One idea that was considered, would have been to not send over values as their text representation, but instead as their bit-representation. Inspired by protobufs [18], this approach could further decrease the symbol count, thus improve performance. It is unclear whether this approach would have been possible, as it requires the ability to perform bit operations with NanoPy. Additionally we would have to assume, that the Oxocard MQTT stack allows for bit strings to be received.

Buttons

At the moment there is no way to use the buttons of the Oxocard. This is mainly due to the fact, that introducing buttons would have further complicated the communication between the Oxocards and the editor. Consider the question of where the code is located that reacts to a button press. Do we continuously poll the button as if it was a sensor and then react once a button is pressed from within the editor? Or do we make the buttons programmable by defining a batch of instructions, that is executed on the card once a button is pressed? Are there any intrinsic limitations of both approaches?

While all interesting questions, they would have gone beyond the scope of this thesis. After all, the buttons would have become an input device and maybe there is a uniform way WebTigerPython would like to handle input devices?

Interactivity

The benchmarking section gave an overview over the difficulties of the current implementation. With the remaining issue of latency, it will likely never be possible to have truly interactive workloads, like video games, without considerable delay. This would only be possible by transpiling Python, as discussed in Section 2.3, but apart from that there is probably no solution to solve the remaining latency.

5.3 Future Work

First Class Error Handling

One of the things that became apparent while working on the Python portion of the code, was that it would be nice to have a universal way of handling errors. As discussed in Section 3.3, the great thing about scripted languages, is that sometimes slightly incorrect code still works. If I provide a Python function with an integer instead of a floating point value, it will cast the values itself, executing my code at least partially as expected.

While this is great, alerts would be far better. Instead of throwing a type error, the program should execute but alert me that it had to do some extra steps in order

```
1 foo(42)
2 bar()
3
```

(a) Original code.

```
1 establishConnection()
2
3 neededForFoo(42)
4 await foo(42)
5
6 bar()
7 teardownConnection()
8
```

(b) Pre-processed code.

Figure 5.1. Preprocessing mapping.

to run correctly. After all WebTigerPython is about learning how to program and understanding types is part of that journey. It is not helpful to force people to type correctly all the time. However, showing them where they could have been more cautious, may be beneficial. Additionally, these improved error messages can also easily be localized in any language.

Universal Preprocessing

While working on the Python interface, I wished for a universal way to preprocess Python code. Universal preprocessing refers to the ability to prepend and append instructions to the written code at will. This approach would take the replaced code into consideration, thus be universal. The main concern with preprocessing is the fact that new lines of code are inserted and hidden control flow is introduced.

A possible solution could maintain a mapping between the preprocessed code and the actually written code. An example can be seen in Figure 5.1, where two simple function calls require the establishment and consequential tear down of a connection, as well as `foo` having to be executed asynchronously.

While the current implementation allows for similar functionality, the usability and capabilities could be expanded. If the function call `neededForFoo(42)` on line 3 of Figure 5.1 (b) throws an error, the error message will refer to `neededForFoo(42)`, while from the users point of view the faulty code should be `foo(42)` somehow. This could be done by the aforementioned mapping, which in this case would link back to `foo()` if any of the code generated by it is faulty.

Bluetooth Support and other Devices

Since the Oxocard already had an MQTT interface, this thesis solely focused on using MQTT and as an extension the internet for transmitting instructions. However, future devices may rely on other communication methods like Bluetooth or the USB port. This thesis has demonstrated several advantages of an RPC protocol compared with flashing the code directly on device. Thus, motivating the extension of the implemented RPC protocol for other devices and communication channels in the future.

Bibliography

[1] Oxon AG. Blockly description. https://oxocard.ch/blocklycard/. Accessed 2024-08-29.

[2] Oxon AG. Nanopy documentation. https://nanopy.io/en/referenz/. Accessed 2024-08-29.

[3] Oxon AG. Nanopy editor. https://editor.nanopy.io/. Accessed 2024-07-02.

[4] Oxon AG. Nanopy overview. https://nanopy.io/overview/. Accessed 2024-08-29.

[5] Oxon AG. Oxocard website. https://oxocard.ch/. Accessed 2024-07-02.

[6] Oxon AG. Oxon home page. https://oxon.ch/. Accessed 2024-08-29.

[7] Akamai. Akamai shared cpi resources. https://techdocs.akamai.com/cloud-computing/docs/shared-cpu-compute-instances. Accessed 2024-08-26.

[8] Amazon. What is mqtt? https://aws.amazon.com/what-is/mqtt/. Accessed 2024-07-11.

[9] Clemens Bachmann. WebTigerJython 3 A Web-Based Python IDE Supporting Educational Robotics. Master's thesis, ETH Zurich, 2023.

[10] Melvin Bender, Erkin Kirdan, Marc-Oliver Pahl, and Georg Carle. Open-source mqtt evaluation. In *2021 IEEE 18th Annual Consumer Communications & Networking Conference (CCNC)*, pages 1–4, 2021.

[11] Mark Csurgay. Implementing GPanel: A Coordinate Graphics Library for WebTigerPython. Master's thesis, ETH Zürich, 2024.

[12] The Pyodide development team. Pyodide Python compatibility. https://pyodide.org/en/stable/usage/wasm-constraints.html. Accessed 2024-08-25.

[13] The Pyodide development team. pyodide/pyodide. https://doi.org/10.5281/zenodo.7570138, August 2021.

[14] Espressif. Esp32 controller overview. https://www.espressif.com/en/products/socs/esp32. Accessed 2024-08-25.

[15] Eclipse Foundation. mosquitto.conf man page. https://mosquitto.org/man/mosquitto-conf-5.html. Accessed 2024-07-12.

[16] Eric Gamess, Trent N. Ford, and Monica Trifas. Performance evaluation of a widely used implementation of the mqtt protocol with large payloads in normal operation and under a dos attack. In *Proceedings of the 2021 ACM Southeast Conference*, ACM SE '21, page 154–162, New York, NY, USA, 2021. Association for Computing Machinery.

[17] Google. Go use cases, cloud & networking. https://go.dev/solutions/cloud. Accessed 2024-08-29.

[18] Google. Protobufs overview. https://protobuf.dev/overview/. Accessed 2024-08-28.

[19] Tobias Kohn. *Teaching Python Programming to Novices: Addressing Misconceptions and Creating a Development Environment*. Doctoral thesis, ETH Zurich, Zürich, 2017.

[20] krylovsk. MQTT benchmark. https://github.com/krylovsk/mqtt-benchmark. Accessed 2024-08-26.

[21] Roger A. Light. Mosquitto: server and client implementation of the mqtt protocol. *Journal of Open Source Software*, 2(13):265, 2017.

[22] Nicole Trachsler. Webtigerjython - a browser-based programming ide for education. Master thesis, ETH Zurich, Zurich, 2018.

[23] Can I Use. Webusb browser compatibility. https://caniuse.com/webusb. Accessed 2024-08-25.

[24] WHATWG. Websocket standard. https://websockets.spec.whatwg.org/. Accessed 2024-08-29.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Eigenständigkeitserklärung

Die unterzeichnete Eigenständigkeitserklärung ist Bestandteil jeder während des Studiums verfassten schriftlichen Arbeit. Eine der folgenden drei Optionen ist in Absprache mit der verantwortlichen Betreuungsperson verbindlich auszuwählen:

- (●) Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben, namentlich, dass mir niemand beim Verfassen der Arbeit geholfen hat. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuungsperson. Es wurden keine Technologien der generativen künstlichen Intelligenz[1] verwendet.
- ( ) Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben, namentlich, dass mir niemand beim Verfassen der Arbeit geholfen hat. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuungsperson. Als Hilfsmittel wurden Technologien der generativen künstlichen Intelligenz[2] verwendet und gekennzeichnet.
- ( ) Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben, namentlich, dass mir niemand beim Verfassen der Arbeit geholfen hat. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuungsperson. Als Hilfsmittel wurden Technologien der generativen künstlichen Intelligenz[3] verwendet. Der Einsatz wurde, in Absprache mit der Betreuungsperson, nicht gekennzeichnet.

Titel der Arbeit:

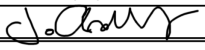
Implementing the Oxocard for WebTigerPython

Verfasst von:
Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich.

Name(n):	**Vorname(n):**
Anderegg	Josh

Ich bestätige mit meiner Unterschrift:
- Ich habe mich an die Regeln des «Zitierleitfadens» gehalten.
- Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu und vollständig dokumentiert.
- Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Eigenständigkeit überprüft werden kann.

Ort, Datum	**Unterschrift(en)**
04/09/2024	

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich. Durch die Unterschriften bürgen sie grundsätzlich gemeinsam für den gesamten Inhalt dieser schriftlichen Arbeit.

[1] z. B. ChatGPT, DALL E 2, Google Bard
[2] z. B. ChatGPT, DALL E 2, Google Bard
[3] z. B. ChatGPT, DALL E 2, Google Bard